(1) Explain Naïve string matching algorithm.

- The string-matching problem is defined as follows.
- We assume that the text is an array \( T[1...n] \) of length \( n \) and that the pattern is an array \( P[1...m] \) of length \( m \leq n \).
- We further assume that the elements of \( P \) and \( T \) are characters drawn from a finite alphabet \( \Sigma \). For example, we may have \( \Sigma = \{0, 1\} \) or \( \Sigma = \{a, b, ..., z\} \).
- The character arrays \( P \) and \( T \) are often called strings of characters.
- We say that pattern \( P \) occurs with shift \( s \) in text \( T \) (or, equivalently, that pattern \( P \) occurs beginning at position \( s + 1 \) in text \( T \)) if \( 0 \leq s \leq n - m \) and \( T[s + 1...s + m] = P[1...m] \) (that is, if \( T[s + j] = P[j] \), for \( 1 \leq j \leq m \)).
- If \( P \) occurs with shift \( s \) in \( T \), then we call \( s \) a valid shift; otherwise, we call \( s \) an invalid shift. The string-matching problem is the problem of finding all valid shifts with which a given pattern \( P \) occurs in a given text \( T \).
- The naive algorithm finds all valid shifts using a loop that checks the condition \( P[1...m] = T[s + 1...s + m] \) for each of the \( n - m + 1 \) possible values of \( s \).

**NAIVE-STRING-MATCHER**(\( T, P \))

1. \( n \leftarrow \text{length}[T] \)
2. \( m \leftarrow \text{length}[P] \)
3. \( \text{for } s \leftarrow 0 \text{ to } n - m \text{ do} \)
   4. \( \text{if } P[1,..., m] = T[s + 1,..., s + m] \)
   5. \( \text{then print "Pattern occurs with shift" } s \)

- The naive string-matching procedure can be interpreted graphically as sliding a "template" containing the pattern over the text, noting for which shifts all of the characters on the template equal the corresponding characters in the text, as illustrated in Figure.
- The for loop beginning on line 3 considers each possible shift explicitly.
- The test on line 4 determines whether the current shift is valid or not; this test involves an implicit loop to check corresponding character positions until all positions match successfully or a mismatch is found.
- Line 5 prints out each valid shift \( s \).
- Example

![Diagram](image)

- In the above example, valid shift is \( s = 3 \) for which we found the occurrence of pattern \( P \) in text \( T \).
- Procedure **NAIVE-STRING-MATCHER** takes time \( O((n - m + 1)m) \), and this bound is tight in the worst case.
The running time of NAIVE-STRING-MATCHER is equal to its matching time, since there is no preprocessing.

(2) Explain Rabin-carp method for string matching and also give the algorithm.

- This algorithm makes use of elementary number-theoretic notions such as the equivalence of two numbers modulo a third number.
- Let us assume that \( \Sigma = \{0, 1, 2 \ldots 9\} \), so that each character is a decimal digit. (In the general case, we can assume that each character is a digit in radix-\( d \) notation, where \( d = |\Sigma| \)).
- We can then view a string of \( k \) consecutive characters as representing a length-\( k \) decimal number. The character string 31415 thus corresponds to the decimal number 31,415.
- Given a pattern \( P[1 \ldots m] \), let \( p \) denote its corresponding decimal value.
- In a similar manner, given a text \( T[1 \ldots n] \), let \( t_i \) denote the decimal value of the length-\( m \) substring \( T[s + 1 \ldots s + m] \), for \( s = 0, 1 \ldots n - m \).
- Certainly, \( t_i = p \) if and only if \( T[s + 1 \ldots s + m] = P[1 \ldots m] \); thus, \( s \) is a valid shift if and only if \( t_i = p \).
- We can compute \( p \) in time \( \Theta(m) \) using Horner's rule:
  \[
p = P[m] + 10(P[m-1] + 10(P[m-2] + \cdots + 10(P[2] + 10P[1])\ldots))
\]
  The value \( t_0 \) can be similarly computed from \( T[1 \ldots m] \) in time \( \Theta(m) \).
- To compute the remaining values \( t_1, t_2, \ldots, t_{n-m} \) in time \( \Theta(n - m) \), it suffices to observe that \( t_{i+1} \) can be computed from \( t_i \) in constant time, since
  \[
t_{i+1} = 10(t_i - 10^{m-1}T[s + 1]) + T[s + m + 1]
\]
  Subtracting \( 10^{m-1}T[s + 1] \) removes the high-order digit from \( t_i \), multiplying the result by 10 shifts the number left one position, and adding \( T[s + m + 1] \) brings in the appropriate lower order digit.
  For example, if \( m = 5 \) and \( t_i = 31415 \) then we wish to remove the high order digit \( T[s + 1] = 3 \) and bring in the new lower order digit (suppose it is \( T[s + 5 + 1] = 2 \)) to obtain
  \[
t_{i+1} = 10(31415 - 10000 \times 3) + 2 = 14152
\]
  The only difficulty with this procedure is that \( p \) and \( t_i \) may be too large to work with conveniently.
- There is a simple cure for this problem, compute \( p \) and the \( t_i's \) modulo a suitable modulus \( q \).
  \[
t_{i+1} = (d(t_i - T[s + 1]h) + T[s + m + 1]) \mod q
\]
  Where \( h = d^{m-1} \mod q \) is the value of the digit "1" in the high-order position of an \( m \)-digit text window.
- The solution of working modulo \( q \) is not perfect, however \( t_i = p \mod q \) does not imply that \( t_i = p \) but if \( t_i \neq p \mod q \) definitely implies \( t_i \neq p \), so that shift \( s \) is invalid.
- Any shift \( s \) for which \( t_i = p \mod q \) must be tested further to see whether \( s \) is really valid or it is just a spurious hit.
- This additional test explicitly checks the condition \( T[s + 1 \ldots s + m] = P[1 \ldots m] \).
Example

pattern P

3 1 4 1 5

mod 13

7

text T

2 3 5 9 0 2 3 1 4 1 5 2 6 7 3 9 9 2 1

mod 13

7

valid match

spurious hit

Algorithm RABIN-KARP-MATCHER(T, P, d, q)

n ← length[T];
m ← length[P];
h ← d^{m-1} mod q;
p ← 0;
t_{0} ← 0;
for i ← 1 to m do
   p ← (dp + P[i]) mod q;
   t_{0} ← (dt_{0} + P[i]) mod q
for s ← 0 to n – m do
   if p == t_{s}, then
      if P[1..m] == T[s+1..s+m] then
         print “pattern occurs with shift s”
   if s < n-m then
      t_{s+1} ← (d(t_{s} – T[s+1]h) + T[s+m+1]) mod q

Analysis

• RABIN-KARP-MATCHER takes Θ(m) preprocessing time and it matching time is Θ(m(n – m + 1)) in the worst case.

(3) Explain finite automata for string matching.

• Many string-matching algorithms build a finite automaton that scans the text string T for all occurrences of the pattern P.
We begin with the definition of a finite automaton. We then examine a special string-matching automaton and show how it can be used to find occurrences of a pattern in a text. Finally, we shall show how to construct the string-matching automaton for a given input pattern.

Finite automata
A finite automaton $M$ is a 5-tuple $(Q, q_0, A, \Sigma, \delta)$, where
- $Q$ is a finite set of states,
- $q_0 \in Q$ is the start state,
- $A \subseteq Q$ is a distinguished set of accepting states,
- $\Sigma$ is a finite input alphabet,
- $\delta$ is a function from $Q \times \Sigma$ into $Q$, called the transition function of $M$.

The finite automaton begins in state $q_0$ and reads the characters of its input string one at a time. If the automaton is in state $q$ and reads input character $a$, it moves ("makes a transition") from state $q$ to state $\delta(q, a)$. Whenever its current state $q$ is a member of $A$, the machine $M$ is said to have accepted the string read so far. An input that is not accepted is said to be rejected.

Following Figure illustrates these definitions with a simple two-state automaton.

String-matching automata
- There is a string-matching automaton for every pattern $P$; this automaton must be constructed from the pattern in a preprocessing step before it can be used to search the text string.
- In our example pattern $P = ababaca$.
- In order to specify the string-matching automaton corresponding to a given pattern $P[1, \ldots, m]$, we first define an auxiliary function $\sigma$, called the suffix function corresponding to $P$.
- suffix of a string $x$ is denoted as $w \supseteq x$ if $x = yw$.
- The function $\sigma$ is a mapping from $\Sigma^*$ to $\{0, 1, \ldots, m\}$ such that $\sigma(x)$ is the length of the longest prefix of $P$ that is a suffix of $x$:

$$\sigma(x) = \max\{k : P_k \supseteq x\}$$

- The suffix function $\sigma$ is well defined since the empty string $P_0 = \varepsilon$ is a suffix of every string.
- As examples, for the pattern $P = ab$, we have $\sigma(\varepsilon) = 0$, $\sigma(ccaca) = 1$, and $\sigma(ccab) = 2$. 

![Finite automaton diagram](attachment:image.png)
For a pattern \( P \) of length \( m \), we have \( \sigma(x) = m \) if and only if \( P \supseteq x \). It follows from the definition of the suffix function that if \( x \supseteq y \), then \( \sigma(x) \leq \sigma(y) \).

We define the string-matching automaton that corresponds to a given pattern \( P[1,...,m] \) as follows.

The state set \( Q \) is \( \{0, 1 \ldots m\} \). The start state \( q_0 \) is state 0, and state \( m \) is the only accepting state.

The transition function \( \delta \) is defined by the following equation, for any state \( q \) and character \( a \):

\[
\delta(q, a) = \sigma(P_qa)
\]

The following procedure computes the transition function \( \delta \) from a given pattern \( P[1,...,m] \).

**Algorithm Compute-Transition-Function(\( P, \Sigma \))**

1. \( m \leftarrow \text{length}[P] \)
2. \( q \leftarrow 0 \)
3. for each character \( a \in \Sigma \) do
4. \( k \leftarrow \min(m + 1, q + 2) \)
5. repeat \( k \leftarrow k - 1 \)
6. until \( P_k \supseteq P_qa \)

Computing the transition function

The following procedure computes the transition function \( \delta \) from a given pattern \( P[1,...,m] \).

**Algorithm Finite-Automaton-Matcher(\( T, \delta, m \))**

1. \( n \leftarrow \text{length}[T] \)
2. \( q \leftarrow 0 \)
3. for \( i \leftarrow 1 \) to \( n \) do
4. \( q \leftarrow \delta(q, T[i]) \)
5. if \( q = m \) then print "Pattern occurs with shift" \( i - m \)
7. \( \delta(q, a) \leftarrow k \)
8. \( \text{return} \ \delta \)

- This procedure computes \( \delta(q, a) \) in a straightforward manner according to its definition.
- The nested loops beginning on lines 2 and 3 consider all states \( q \) and characters \( a \), and lines 4-7 set \( \delta(q, a) \) to be the largest \( k \) such that \( P_k \supseteq P_q a \). The code starts with the largest conceivable value of \( k \), which is \( \min(m, q + 1) \), and decreases \( k \) until \( P_k \supseteq P_q a \).
- Time complexity for string matching algorithm

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Preprocessing time</th>
<th>Matching time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive</td>
<td>0</td>
<td>( O((n - m + 1)m) )</td>
</tr>
<tr>
<td>Rabin-Karp</td>
<td>( \Theta(m) )</td>
<td>( O((n - m + 1)m) )</td>
</tr>
<tr>
<td>Finite automaton</td>
<td>( O(m</td>
<td>\Sigma</td>
</tr>
</tbody>
</table>