String Matching

Problem formulation: Assume that

Text *is an array* ***T[1..n]*** *of length* ***n*** *of elements from a finite alphabet* Σ.

Pattern *P is an array* ***P[1..m]*** *of length* ***m ≤ n*** *of elements from* Σ.

Characters array P and T are often called **strings** of characters

Pattern P occurs with shift ***s*** *in text T if* ***T[s+1..s+m] = P[1..m]*** *i.e. Pattern P occurs beginning at position s+1 in text T*

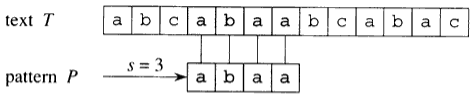
***If P occurs with shift s in T, then s is a* valid shift otherwise invalid shift.**

String matching problem *is the problem of finding all valid shifts with which a given pattern P occurs in a given text T.*

Point to note: *Valid shifts must be in the range 0 ≤ s ≤ n-m, so there are n-m+1 possible different values of valid shifts*

An examples of string matching

Example 1*:*



The string-matching problem. The goal is to find all occurrences of the pattern P = abaa in the text T = abcabaabcabac. The pattern occurs only once in the text, at shift $s = 3$. The shift $s = 3$ is said to be a valid shift. Each character of the pattern is connected by a vertical line to the matching character in the text, and all matched characters are shown shaded.

Example 2*:* P= abab, T= abcababababbc

P occurs at s=3 and s=5.

one occurrence begins within another one

Applications of Matching

- **In Computational Biology—DNA Sequence Analysis**
  - DNA sequence is a long word (or text) over a 4-letter alphabet
  - GTTTGAGTGGTCAGTCTTTTCGTTTCGACGGAGCCCCCAATTAATAAACTCAT AAGCAGACCTCAGTTCGCTTAGAGCAGCCGAAA…..
  - Find a Specific pattern P
- **Finding occurrences of patterns in documents formed using alphabet**
  - Word processing
  - Web searching
  - Text editing programs
  - Desktop search (Google, MSN)
- **Computer security**
- **Matching strings of bytes containing**
  - Graphical data
  - Machine code
- **grep in unix**
  - grep searches for lines matching a pattern.

Notation and terminology

- *The set of all finite-length strings using characters from an alphabet Σ is denoted Σ^*.*
- *The length of a string x is denoted |x|.*
- *The zero-length empty string is denoted ε.*
- *The concatenation of two strings x and y is denoted by xy having length |x|+|y|*
- *A string w is a **prefix** of a string x, denoted $w \sqsubset x$ if x = wy, for some string $y \in \Sigma^*$.*
- *A string w is a **suffix** of a string x, denoted $w \sqsupset x$ if x = yw, for some string $y \in \Sigma^*$.*
- *Empty string is both suffix and a prefix of every string*

- *Denote the **k-character prefix** P[1..k] of a string P[1..m] by $\boldsymbol{P_k}$.*

String Matching Algorithms

Algorithm	Preprocessing time	Matching time
Naive	0	$O((n-m+1)m)$
Rabin-Karp	$\Theta(m)$	$O((n-m+1)m)$
~~Finite automaton~~	~~$O(m\|\Sigma\|)$~~	~~$\Theta(n)$~~
Knuth-Morris-Pratt	$\Theta(m)$	$\Theta(n)$

Figure 32.2 The string-matching algorithms in this chapter and their preprocessing and matching times.

Overlapping shift lemma

Suppose x,y,z are strings such that $x \sqsupset z$ and $y \sqsupset z$, then

if $|x| \leq |y|$, then $x \sqsupset y$;

if $|x| \geq |y|$, then $y \sqsupset x$; and

if $|x| = |y|$, then $x=y$.

Graphical Proof of Lemma

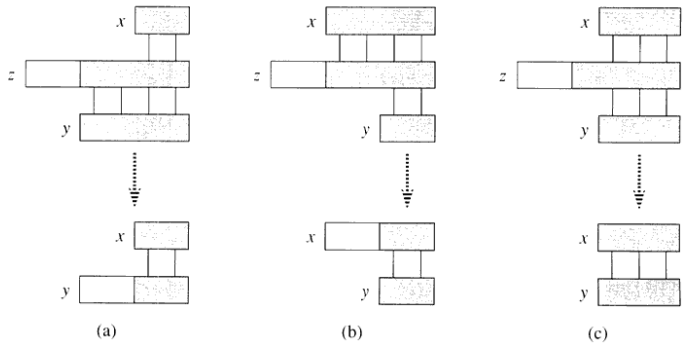


Figure 32.3 A graphical proof of Lemma 32.1. We suppose that $x \sqsupset z$ and $y \sqsupset z$. The three parts of the figure illustrate the three cases of the lemma. Vertical lines connect matching regions (shown shaded) of the strings. **(a)** If $|x| \leq |y|$, then $x \sqsupset y$. **(b)** If $|x| \geq |y|$, then $y \sqsupset x$. **(c)** If $|x| = |y|$, then $x = y$.

Naïve string matching

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 ← length[T]
2  m ← length[P]
3  for s ← 0 to n − m
4       do if P[1..m] = T[s+1..s+m]
5            then print "Pattern occurs with shift" s
```

Running time: ***O((n-m+1)m)***

Operation of Naïve string matching

Naïve string matching procedure can be interpreted graphically as sliding a ***"template"*** *containing pattern over text, noting for which shifts all of the characters on template equal the corresponding characters in text*

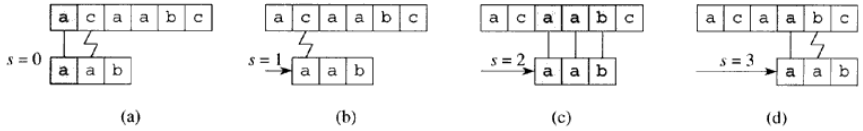


Figure 32.4 The operation of the naive string matcher for the pattern P = aab and the text T = acaabc. We can imagine the pattern P as a "template" that we slide next to the text. **(a)–(d)** The four successive alignments tried by the naive string matcher. In each part, vertical lines connect corresponding regions found to match (shown shaded), and a jagged line connects the first mismatched character found, if any. One occurrence of the pattern is found, at shift $s = 2$, shown in part (c).

Naïve string matching

- Naïve string matcher is not optimal procedure
- Whenever a character mismatch occurs after matching of several characters, the comparison begins by going back in *T* from the character which follows the last beginning character
- Inefficient because information gained about text from one vale of *s* is entirely ignored in considering other value of *s* ; which can be very valuable
- If known to us how the pattern matches against itself, we can slide the pattern more characters ahead than just one character as in the naïve algorithm
- ***is it possible to improve the length of the shifts ??? !***

Knuth-Morris-Pratt (KMP) algorithm

- Shift more than one position
- Preprocessing of pattern to avoid trivial comparisons
- Avoids re-computing matches
- Keeps information that naïve algorithm wasted gathered during the scan of the text
  - By avoiding this waste of information ,achieved a linear running time
- KMP implementation is efficient because it minimizes the total number of comparisons of pattern itself against the text string

□

Knuth-Morris-Pratt (KMP) algorithm

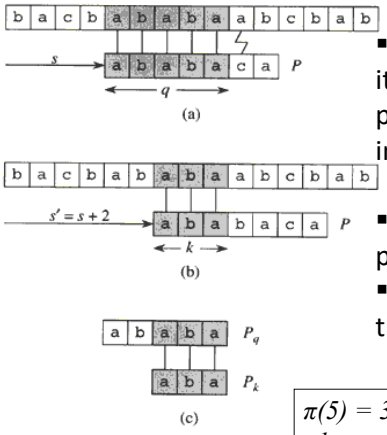
- reuse of the work, after a failure of valid shift
- After some character (say q) matches of P with T and then a mismatch,
  - the matched q characters allows us to determine immediately that certain *shifts are invalid*. So directly go to the shift which is potentially valid.
- Matched characters in T are in fact a prefix of P
  - so just from P, determine whether a shift is invalid or not.
- Compute in advance how far to jump in P when a match fails without any knowledge of T
- Define a ***prefix function*** π, which encapsulates the knowledge about how the pattern P matches against shifts of itself
  - Function π :$\{1,2,\ldots,m\}\rightarrow\{0,1,\ldots,m\text{-}1\}$ such that
  - $\pi[q]=\max\{ k : k<q \text{ and } P_k \; P_q \}$,
    - length of the longest proper prefix in the (sub) pattern P that matches a suffix in the same (sub)pattern P_q

Prefix function π

- Preprocess the pattern to find matches of prefixes of the pattern with the pattern itself
  - Indicates how much of the last comparison can be re-used if it fails
- Defined as size of largest prefix of P[0….j-1] that is also a suffix of P[1....j]

  π[q] = the largest k < q such that
  (P[1], P[2], ..., P[k-1]) = (P[q-k+1], P[q-k+2], ..., P[q-1])
- Prefix function computes shifting of pattern matches within itself
  - if we know that how pattern matches shifts against itself
  - then we can slide pattern more characters towards right than just one character
- Thought is that slide the pattern towards the right among the string such that the longest prefix of P that we have matched; matches the longest suffix of T that we have already matched
- If longest prefix of P that matches suffix of T is nothing, then slide whole pattern towards right

Prefix function



- If we precompute prefix function of P (against itself), then whenever a mismatch occurs, the prefix function can determine which shift(s) are invalid and directly ruled out
- So move directly to the shift which is potentially valid.
- However, there is no need to compare these characters again since they are equal.

π(5) = 3 indicates that a shift of +1 to the right invalid ; however, +2 is potentially valid.

Figure 32.10 The prefix function π. **(a)** The pattern P = ababaca is aligned with a text T so that the first $q = 5$ characters match. Matching characters, shown shaded, are connected by vertical lines. **(b)** Using only our knowledge of the 5 matched characters, we can deduce that a shift of $s + 1$ is invalid, but that a shift of $s' = s + 2$ is consistent with everything we know about the text and therefore is potentially valid. **(c)** The useful information for such deductions can be precomputed by comparing the pattern with itself. Here, we see that the longest prefix of P that is also a proper suffix of P_5 is P_3. This information is precomputed and represented in the array π, so that $\pi[5] = 3$. Given that q characters have matched successfully at shift s, the next potentially valid shift is at $s' = s + (q - \pi[q])$.

```
KMP-MATCHER(T, P)
1  n ← length[T]
2  m ← length[P]
3  π ← COMPUTE-PREFIX-FUNCTION(P)
4  q ← 0                                  ▷ Number of characters matched.
5  for i ← 1 to n                         ▷ Scan the text from left to right.
6      do while q > 0 and P[q + 1] ≠ T[i]
7          do q ← π[q]                    ▷ Next character does not match.
8      if P[q + 1] = T[i]
9        then q ← q + 1                   ▷ Next character matches.
10     if q = m                           ▷ Is all of P matched?
11       then print "Pattern occurs with shift" i − m
12           q ← π[q]                     ▷ Look for the next match.
```

KMP-MATCHER→ Θ(m)+ Θ(n).

```
COMPUTE-PREFIX-FUNCTION(P)
 1  m ← length[P]
 2  π[1] ← 0
 3  k ← 0
 4  for q ← 2 to m
 5       do while k > 0 and P[k + 1] ≠ P[q]
 6            do k ← π[k]
 7          if P[k + 1] = P[q]
 8            then k ← k + 1
 9          π[q] ← k
10  return π
```

COMPUTE-PREFIX-FUNCTION → Θ(m)

Complete Prefix function

i	1	2	3	4	5	6	7	8	9	10
$P[i]$	a	b	a	b	a	b	a	b	c	a
$\pi[i]$	0	0	1	2	3	4	5	6	0	1

(a)

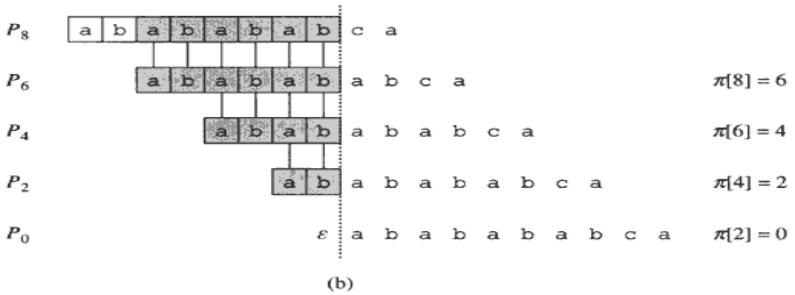


Figure 32.11 An illustration of Lemma 32.5 for the pattern P = ababababca and $q = 8$. **(a)** The π function for the given pattern. Since $\pi[8] = 6$, $\pi[6] = 4$, $\pi[4] = 2$, and $\pi[2] = 0$, by iterating π we obtain $\pi^*[8] = \{6, 4, 2, 0\}$. **(b)** We slide the template containing the pattern P to the right and note when some prefix P_k of P matches up with some proper suffix of P_8; this happens for $k = 6$, 4, 2, and 0. In the figure, the first row gives P, and the dotted vertical line is drawn just after P_8. Successive rows show all the shifts of P that cause some prefix P_k of P to match some suffix of P_8. Successfully matched characters are shown shaded. Vertical lines connect aligned matching characters. Thus, $\{k : k < q \text{ and } P_k \sqsupset P_q\} = \{6, 4, 2, 0\}$. The lemma claims that $\pi^*[q] = \{k : k < q \text{ and } P_k \sqsupset P_q\}$ for all q.

Rabin-Karp Algorithm

- compares a string's hash values, rather than the strings themselves
- hash value of the next position in the text is easily computed from the hash value of the current position for efficiency
- perform some pre-processing to eliminate locations that could not possibly match, before spending a lot of time comparing chars of a match

Rabin-Karp Strategy

- Choose a **hash function H**:Σ^* → integers.
- Compute ***H (P[1..m])***.
- For each successive **s**, from **0 to n-m**:
  - Compute ***H(T[s+1..s+m])***
  - Compare ***H(T[s+1..s+m]***) with ***H(P[1..m])***.

  i) If H(T[s+1..s+m]) ≠ H(P[1..m]), then T[s+1..s+m] ≠ P[1..m], so that there is **no match**

  ii) If H(T[s+1..s+m]) = H(P[1..m]), then there is **possibly** a ***match*** (hash values of two different elements may collide)

  In this case, ***explicitly check for a match*** T[s+1..s+m] = P[1..m] by comparing character by character as in the naïve matcher

  iii) If H(T[s+1..s+m]) = H(P[1..m]) but T[s+1..s+m] ≠ P[1..m], then we have a **spurious hit**
- For the hash function H:
  - possible to compute H(T[s+2..s+m+1]) from H(T[s+1..s+m]) efficiently

    i.e., starting from H(T[1..m]) it should be possible to efficiently calculate the successive hash values H(T[2..m+1]), H(T[3..m+2]), …, each ***with the help of the previous one.***
  - It should be possible to efficiently compare H(T[s+1..s+m]) with H(P[1..m])

Example: Suppose $\Sigma = \{0, 1, \ldots, 9\}$, the set of decimal digits.

Define the hash function $H:\Sigma^* \rightarrow$ integers by H(w) = decimal number represented by w.

if w = 23903 → ***H(w) = 23,903***;

if w = 02858 → H(w) = 2,858

Note: a decimal character string is simply *a* representation of an integer, but they are *not* the same thing

Given a pattern P[1..m], the ***value H(P[1..m])***, call it **p**, can be computed via

Horner's rule: $\mathbf{p} = P[m] + 10(P[m-1] + 10 (P[m-2] + \ldots + 10 (P[2] + 10P[1]) \ldots))$

Let $\mathbf{t_s}$ denote the ***value H(T[s+1..s+m])***. Then, t_0 can be computed using Horner's rule (as **p** above).

Moreover, $\mathbf{t_{s+1}}$ can be computed from $\mathbf{t_s}$ using the recurrence:

$$\mathbf{t_{s+1}} = 10(t_s - 10^{m-1} T[s+1]) + T[s+m+1] \qquad (32.1)$$

Efficiency: If the constant 10^{m-1} is pre-computed, and if 10^{m-1}, p and t_s, for all s, can each be contained in a single computer word, then each execution of the above equation takes a constant number of arithmetic operations and comparing p and t_s is a single word comparison operation as well.

Therefore, total time: $\Theta(n-m+1)$.

Assumption that $\boldsymbol{10^{m-1}}$, ***p*** and $\boldsymbol{t_s}$ fit into a single computer word is not always feasible.

Working of Rabin-Karp

- Let characters in both arrays T and P be digits in radix- Σ notation. (Σ = (0,1,...,9)
- Let ***p*** be the value of the characters in P
- Choose a prime number ***q*** such that ***10q*** fits within a single computer word to speed computations
- Compute (p mod q)
  - The value of p mod q is what we will be using to find all matches of the pattern P in T.

So in this case the operations involved in the modified recurrence (32.1)

- Compute $t_{s+1} = (10(t_s - 10^{m-1}T[s+1]) + T[s+m+1]) \ mod\ q$

  can each be executed as a one single-precision arithmetic operation.
- Test against P only those sequences in T having the same (mod q) value
- (T[s+1, .., s+m] mod q) can be incrementally computed by subtracting the high-order digit, shifting, adding the low-order bit, all in modulo q arithmetic.

Working of Rabin-Karp

- spurious hits are now an issue and the worst case running time is $\Theta(n-m+1)m$), like the naïve matcher,
  - because every valid shift has to be checked character by character, and there are potentially n-m+1 valid shifts.
- In practice, though, we expect only a few (possibly constant number) of valid shifts, and only a few spurious hits (which also have to be verified character by character), in which case performance is much better than the worst case.

- In general, if the alphabet Σ is of size d, then it is interpreted as $\Sigma = \{0, 1, \ldots, d-1\}$ and a string in Σ^* interpreted as a ***radix-d*** integer.

  Correspondingly, (32.1) which was modified earlier to

  $t_{s+1} = (10(t_s - 10^{m-1}T[s+1]) + T[s+m+1]) \bmod q$

modified now to: avoid computation of 10^{m-1} *on every iteration*

$$t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q \qquad (32.2)$$

(where $h = d^{m-1} \bmod q$)

Text T =

Pattern P = 31415

31415 mod 13= 7

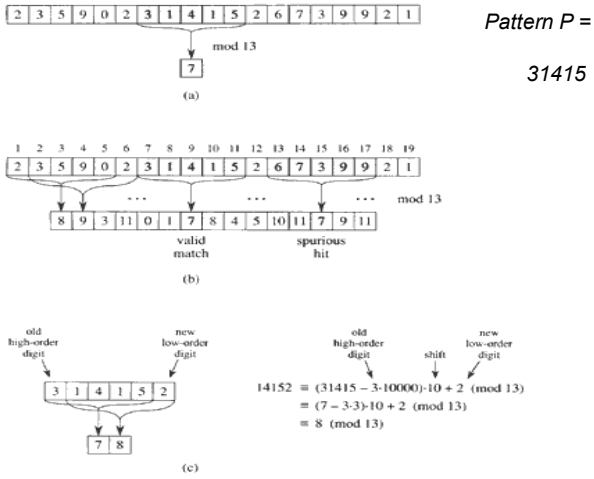


Figure 32.5 The Rabin-Karp algorithm. Each character is a decimal digit, and we compute values modulo 13. **(a)** A text string. A window of length 5 is shown shaded. The numerical value of the shaded number is computed modulo 13, yielding the value 7. **(b)** The same text string with values computed modulo 13 for each possible position of a length-5 window. Assuming the pattern $P = 31415$, we look for windows whose value modulo 13 is 7, since $31415 \equiv 7 \pmod{13}$. Two such windows are found, shown shaded in the figure. The first, beginning at text position 7, is indeed an occurrence of the pattern, while the second, beginning at text position 13, is a spurious hit. **(c)** Computing the value for a window in constant time, given the value for the previous window. The first window has value 31415. Dropping the high-order digit 3, shifting left (multiplying by 10), and then adding in the low-order digit 2 gives us the new value 14152. All computations are performed modulo 13, however, so the value for the first window is 7, and the value computed for the new window is 8.

```
RABIN-KARP-MATCHER(T, P, d, q)
 1  n ← length[T]
 2  m ← length[P]
 3  h ← d^(m−1) mod q
 4  p ← 0
 5  t_0 ← 0
 6  for i ← 1 to m                          ▷ Preprocessing.
 7       do p ← (dp + P[i]) mod q           } Horner's method
 8          t_0 ← (dt_0 + T[i]) mod q       }
 9  for s ← 0 to n − m                      ▷ Matching.
10       do if p = t_s
11             then if P[1..m] = T[s + 1..s + m]
12                       then print "Pattern occurs with shift" s
13          if s < n − m
14             then t_{s+1} ← (d(t_s − T[s + 1]h) + T[s + m + 1]) mod q
```