String Matching

String Matching

- Given <u>a text T</u> and <u>a pattern P</u>, check whether P occurs in T
- Goal: Find pattern of length *M* in a text of length *N*.(typically N >> M)

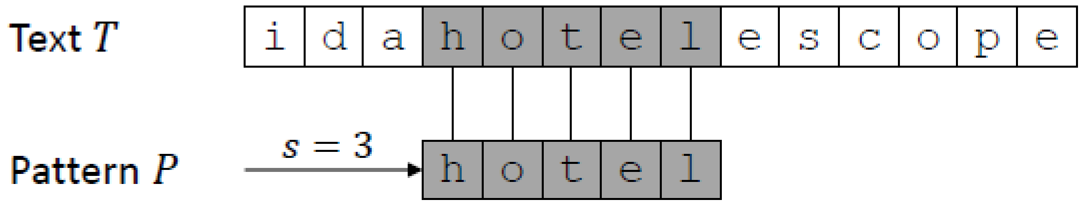
String Matching Examples

- T = {aabbcbbcabbbcbccccabbabbccc}
  Find all occurrences of pattern P = bbc

- *T* = AGCATGCTGCAGTCATGCTTAGGCTA
- *P* = GCT
- *P* appears three times in *T*

Formalization of the String Matching Problem

- **Text** is an array $T[1..n]$ of length n
- **Pattern** is an array $P[1..m]$ of length $m \le n$
- T and P are drawn from a finite alphabet Σ, they are often called **strings** of characters
- P **occurs with shift** $\boldsymbol{s}$ in T if $0 \le s \le n-m$ and $T[s+1..s+m]=P[1..m]$



Applications of String Matching

- Finding patterns in documents
  - Word processing
  - Web searching
  - Desktop search Parsers.
- Spam filters.
- Virus detection, intrusion detection
- Electronic surveillance.
- Natural language processing.
- Bioinformatics.
- FBIs Digital Collection System 3000.
- Feature detection in digitized images.
- …..

Notation and Terminology

- Strings
  - Σ^* set of all finite-length strings with characters from Σ
  - ϵ zero-length empty string also belongs to Σ^*
  - $|x|$ length of a string x
  - xy concatenation of strings x and y has length $|x|+|y|$
- Prefix and suffix
  - string w is a prefix of a string x, denoted as $w \sqsubset x$, if $x=wy$ for some string $y \in \Sigma^*$
  - string w is a suffix of a string x, denoted as $w \sqsupset x$, if $x=yw$ for some string $y \in \Sigma^*$
  - S_k denotes the k-character prefix $S[1..k]$ of the string $S[1..n]$ and thus $S_0=\epsilon$ and $S_n=S=S[1..n]$.

Observations

- Strings
  - $|\epsilon| = 0$
- Prefix and suffix
  - for any string x, $\epsilon \sqsubset x$ and $\epsilon \sqsupset x$
  - if $w \sqsubset x$ or $w \sqsupset x$, then $|w| \leq |x|$
  - for any two strings x and y and any character a, $x \sqsubset y \rightarrow ax \sqsubset ay$ and $x \sqsupset y \rightarrow xa \sqsupset ya$
  - both $\sqsubset$ and $\sqsupset$ are transitive relations
- Reformulated string-matching problem
  - finding all shifts s in the range $0 \leq s \leq n - m$ such that $P \sqsupset T_{s+m}$

Examples

- Assume $\Sigma=\{a,b,c\}$
  - $\Sigma^*=\{\epsilon$, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc,…$\}$
  - x= ab and y= ba
  - $|x|=|y|=2$
  - xy= abba
  - $|xy|=|x|+|y|=4$
  - $\epsilon \sqsubset$ abba and $\epsilon \sqsupset$ abba
  - a $\sqsubset$ abba and a $\sqsupset$ abba
  - ab $\sqsubset$ abba and ba $\sqsupset$ abba
  - abb $\sqsubset$ abba and bba $\sqsupset$ abba
  - abba $\sqsubset$ abba and abba $\sqsupset$ abba

Brute Force (Naïve) Approach

<u>Brute-force algorithm</u>

Step 1 Align pattern at beginning of text

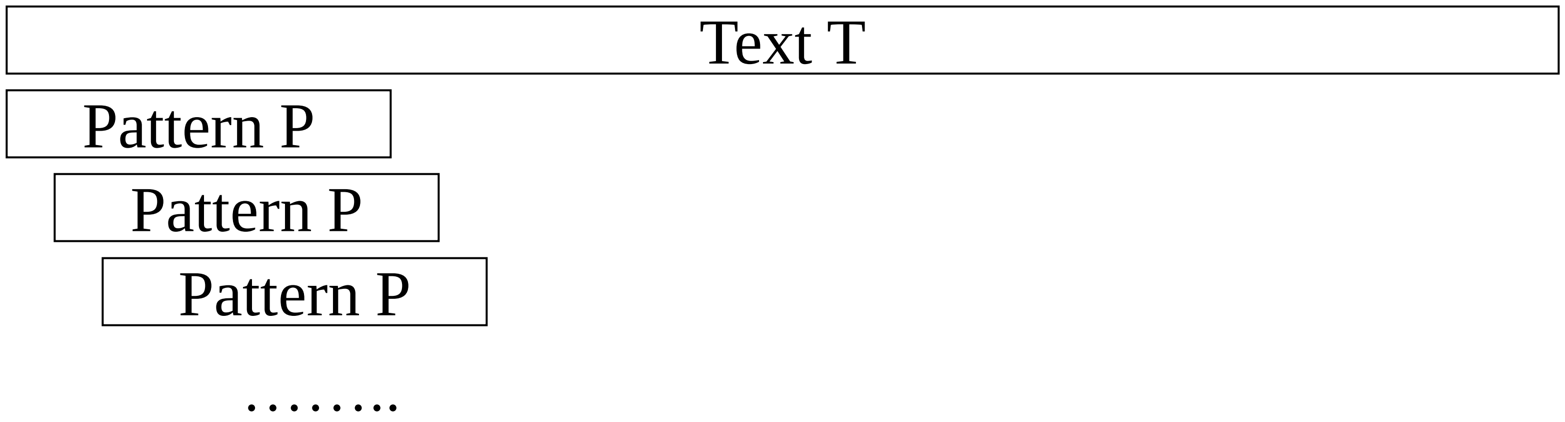
Step 2 Moving from left to right, compare each character of pattern to the corresponding character in text until

- all characters are found to match (successful search); or
- a mismatch is detected

Step 3 While pattern is not found and the text is not yet exhausted, realign pattern one position to the right and repeat Step 2

Brute Force (Naïve) Approach

- Assume |T| = n and |P| = m



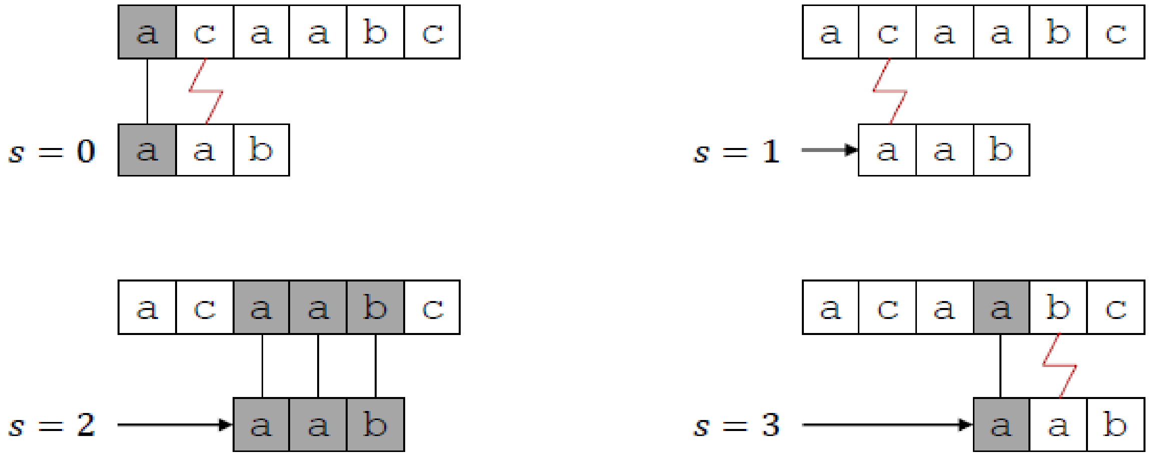
Compare until a match is found.
If so return the index where match occurs
else return -1

Brute Force (Naïve) Algorithm

```
NAÏVE-STRING-MATCHER$(T, P)$
  $n \leftarrow length[T]$
  $m \leftarrow length[P]$
  **for** $s \leftarrow 0$ **to** $n - m$
      **do if** $P[1..m] = T[s+1..s+m]$
          **then** print "Pattern occurs with shift" $s$
```

Brute Force (Naïve) Example

- Graphical interpretation
  - sliding pattern over text in steps of length 1
  - Note for which shifts all of pattern characters equal the corresponding text characters



Brute Force (Naïve) Algorithm

- Brute force worst case
  - O(mn)
  - Expensive for long patterns
- How to improve on this?

Rabin-Karp Algorithm(Hash Table)

- Motivation
  - comparing numbers is “cheaper” than matching strings
  - represent text and pattern as numbers
  - use number-theoretic notions to match strings
- Assumptions and notation
  - $\Sigma_{10}=\{0,1,2,\ldots,9\}$, but in the general case each character will be a digit in radix-d notation where $d=|\Sigma|$
  - p denotes the value corresponding to $P_{1..m}$
  - given T,1..n-, ts denotes the value of the length-m substring $Ts+1..s+m$, for $s=0,1,\ldots,n-m$
  - $–ts=p \Leftrightarrow Ts+1..s+m=P1..m$

Rabin-Karp Algorithm

- Main idea: preprocess *T* to speedup queries
  - Hash every substring of length *k*
  - *k* is a small constant
- For each query *P*, hash the first *k* letters of *P* to retrieve all the occurrences of it within *T*

- Don’t forget to check collisions!

Rabin-Karp Algorithm

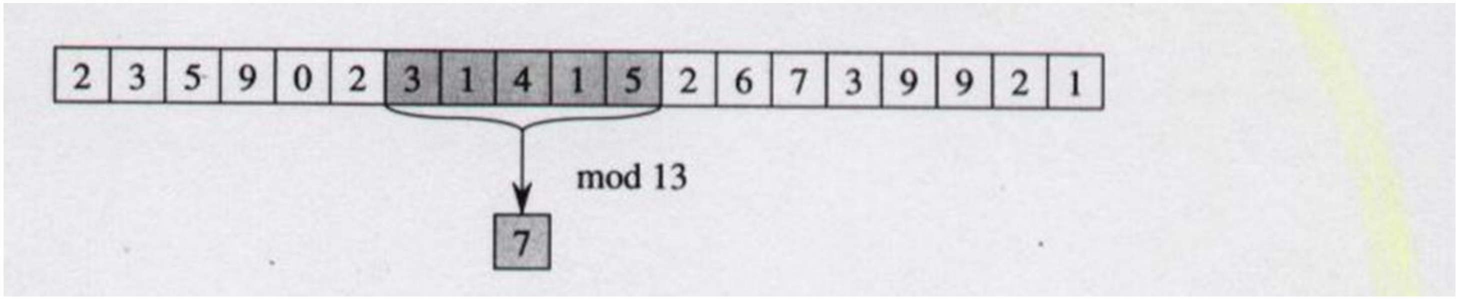
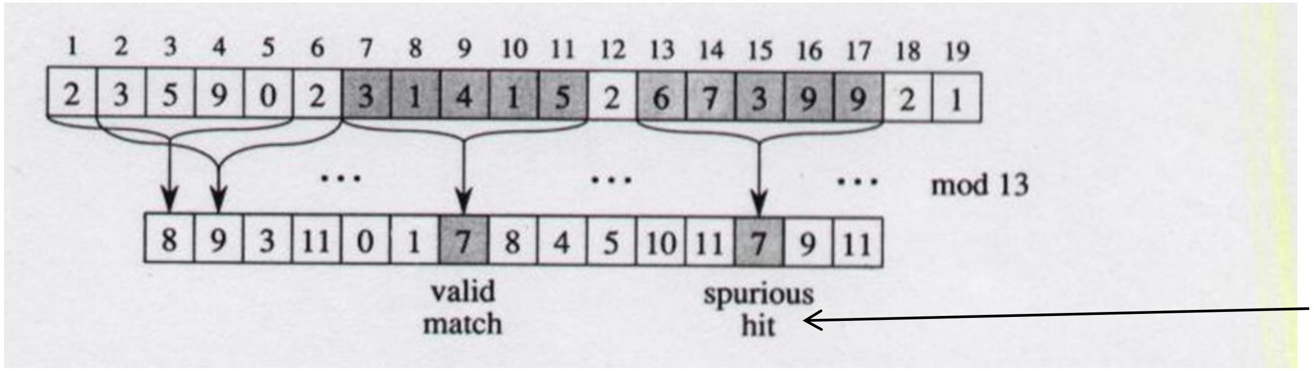


Figure 34.4 The Rabin-Karp algorithm. Each character is a decimal digit, and we compute values modulo 13. **(a)** A text string. A window of length 5 is shown shaded. The numerical value of the shaded number is computed modulo 13, yielding the value 7. [redacted]

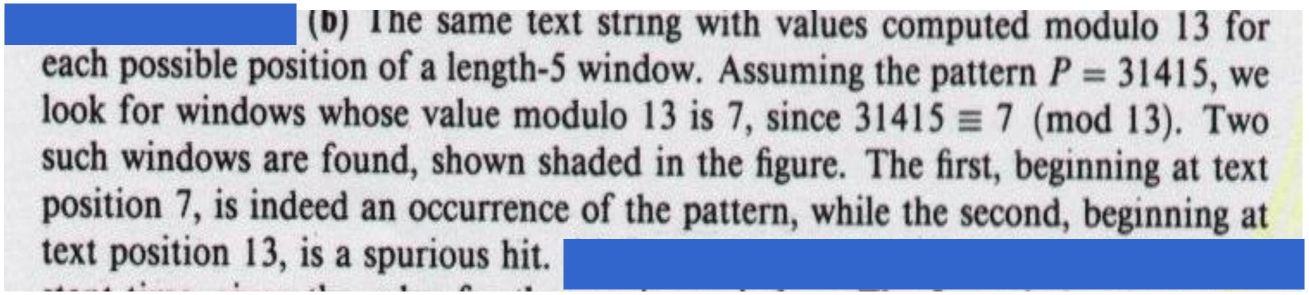
Rabin-Karp Algorithm

$$t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q \,, \qquad (34.2)$$

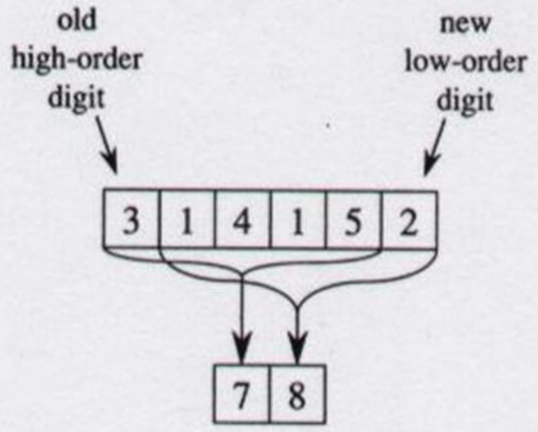


p = 31415

Spurious hit


[redacted] (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. [redacted]

Rabin-Karp Algorithm



$$
\begin{aligned}
14152 &\equiv (31415 - \overset{\substack{\text{old}\\\text{high-order}\\\text{digit}}}{\overset{\downarrow}{3}}\cdot 10000)\cdot \overset{\text{shift}}{\overset{\downarrow}{10}} + \overset{\substack{\text{new}\\\text{low-order}\\\text{digit}}}{\overset{\swarrow}{2}} \pmod{13}\\
&\equiv (7 - 3\cdot 3)\cdot 10 + 2 \pmod{13}\\
&\equiv 8 \pmod{13}
\end{aligned}
$$

[redacted] (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 Algorithm

- Pros:
  - Easy to implement
  - Significant speedup in practice
- Cons:
  - Doesn’t help the asymptotic efficiency
    - Can still take (nm) time
  - A lot of memory consumption