String Algorithms

15CSE211: Design and Analysis of Algorithms

Amrita School of Engineering
Amrita Vishwa Vidyapeetham

Introduction

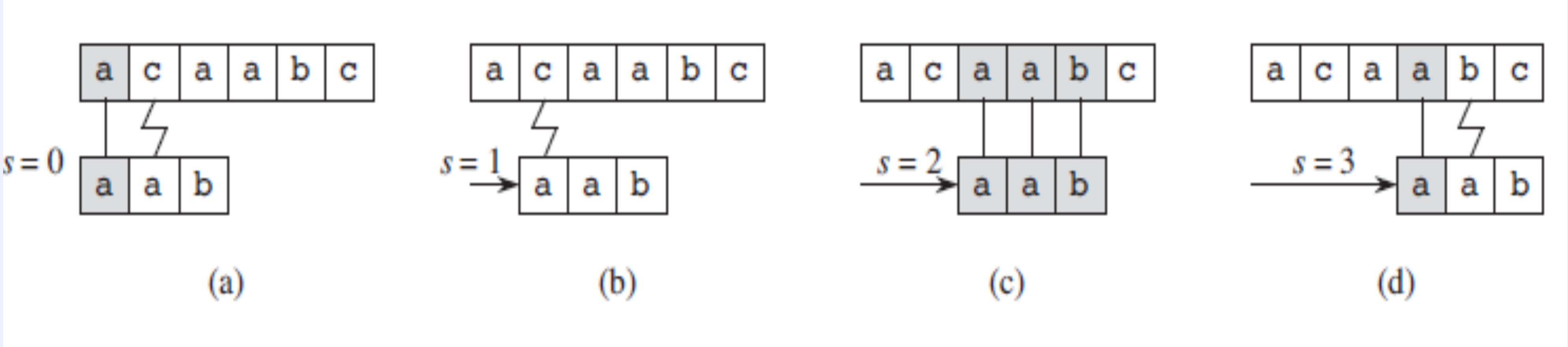
- Let P be a string of size m
  - A substring P[i .. j] of P is a contiguous sequence of P consisting of the characters with ranks between i and j
  - A prefix of P is a substring of the type P[0 .. I]
  - A suffix of P is a substring of the type P[i ..m − 1]
- Given strings T (text) and P (pattern), the pattern matching problem consists of finding a substring of T equal to P

Applications

- Spam detection
- Computer Forensics
- Gene Sequencing
- Screen Scraping

Brute-Force Algorithm

- Compare pattern P with text T starting with first position in T
  - Shift P w.r.t T by one position
  - Repeat till end of T reached
- O(nm) time



Src: CLRS 32.1

Rabin Karp Algorithm

- Tries to reduce the comparisons by using hashing for shifting substring search
  - Two strings are compared iff their hash values are equal
- Hashing scheme
  - Each symbol in alphabet Σ can be represented by an ordinal value { 0, 1, 2, ..., d }
- Hash a pattern P into a numeric value
  - Let a string be represented by the sum of these digits
    - e.g BAN → 1 + 0 + 13 = 14

Rabin Karp Algorithm

- Let T[1..n] be text of length n, and P[1..m] be pattern
  - Let t_s denote the decimal value of the length-m substring T[s+1.. s+*m*], and p decimal value of pattern
  - t_s = p if and only if T[s+1.. s+*m*] = P[1..*m*]
  - S is a valid shift iff t_s = p ie their hash functions are equal
- Compute t_{s+1} from t_s in constant time
  - $t_{s+1} = 10(t_s - 10^{m-1} T[s+1]) + T[s+m+1]$

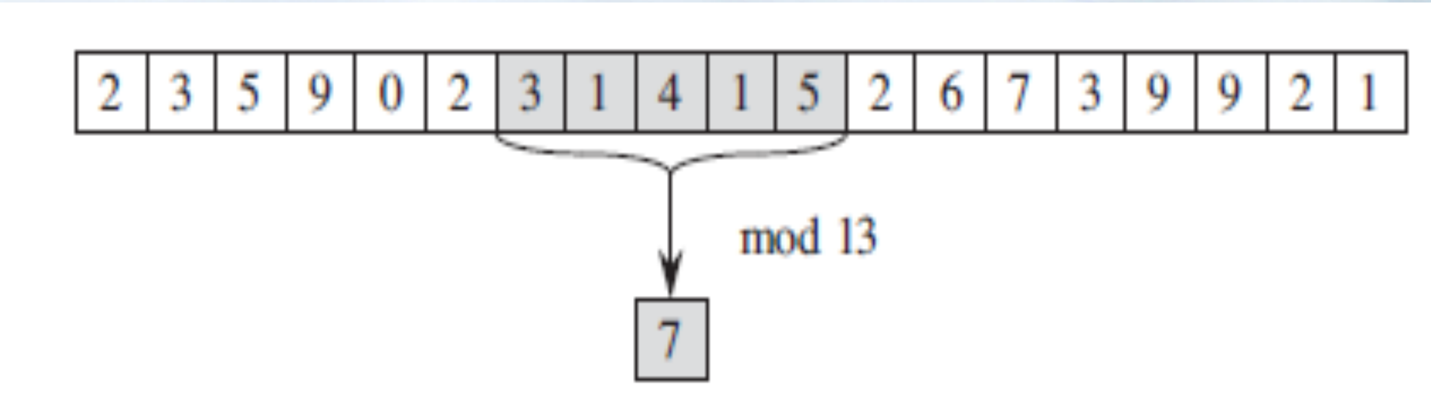
Choice of Hash Function

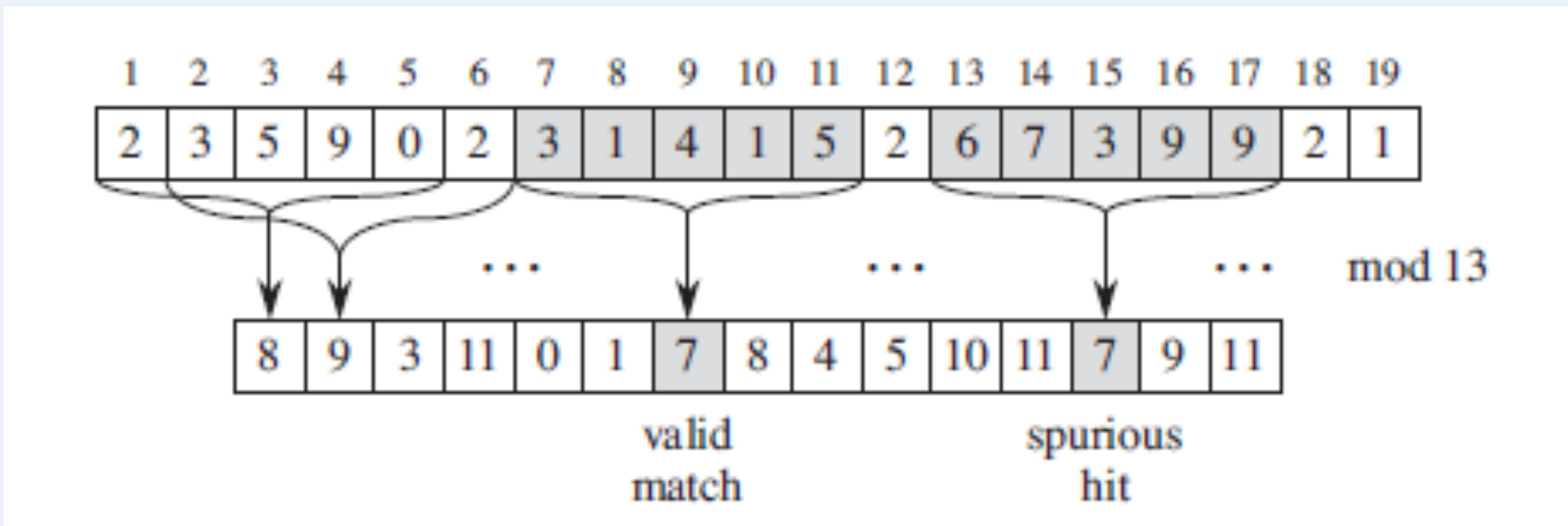
- The number of false positives induced by the hash function should be similar to that achieved by a "random" function
    - Two different strings may have same hash value
- It should be easy to compare two hash values
- Easy to compute t_{s+1} from t_s
    - e.g if t_s = 31415, t_{s+1} is
    - 10(31415-10000.3)+2 = 14152
        - Assuming 2 is the next digit

Possible Hash Function

- Use MOD operation
  - When MOD q, values will be < q
- Usually q is a prime number
- Spurious hits
  - Hash value match does not mean patterns match
  - Hash value mismatch definitely means shift is invalid
  - Any shift s for which t_s = p.mod q must be tested further to see whether s is really valid or we just have a spurious hit.

The idea





Src: CLRS 32.5

The algorithm

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

$n \leftarrow$ length[T] ; $m \leftarrow$ length[P]

$h \leftarrow d^{m-1}$ mod q; $P \leftarrow 0$; $t_0 \leftarrow 0$

for $i \leftarrow 1$ **to** m   ► Preprocessing

  do $p \leftarrow (d*p + P[i]) \bmod q$

    $t_0 \leftarrow (d*t_0 + T[i]) \bmod q$

for $s \leftarrow 0$ **to** $n - m$ ► Matching

  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 $t_{s+1} \leftarrow (d*(t_s - T[s + 1] *h)+T[s + m + 1])$

mod q

Performance Analysis

- Preprocessing (determining each pattern hash)
  - $\Theta(m)$
- Worst case running time
  - $\Theta((n-m+1)m)$
  - No better than naïve method
- Expected case
  - If we assume the number of hits is constant compared to n, we expect $O(n)$
  - Only pattern-match “hits” – not all shifts

Exercise

- 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?
- How would you extend the Rabin-Karp method to the problem of searching a text string for an occurrence of any one of a given set of k patterns? Start by assuming that all k patterns have the same length. Then generalize your solution to allow the patterns to have different lengths.

Boyer Moore Algorithm

- Based on two heuristics
  - Looking-glass heuristic: Compare P with a substring of T moving backwards
  - Character-jump heuristic: When a mismatch occurs at T[i] = c
    - If P contains c, shift P to align the last occurrence of c in P with T[i]
    - Else, shift P to align P[0] with T[i + 1]

Example

a		p	a	t	t	e	r	n		m	a	t	c	h	i	n	g		a	l	g	o	r	i	t	h	m
r	i	t	h	m																							
		r	i	t	h	m																					
							r	i	t	h	m												r	i	t	h	m
												r	i	t	h	m						r	i	t	h	n	
																	r	i	t	h	n						

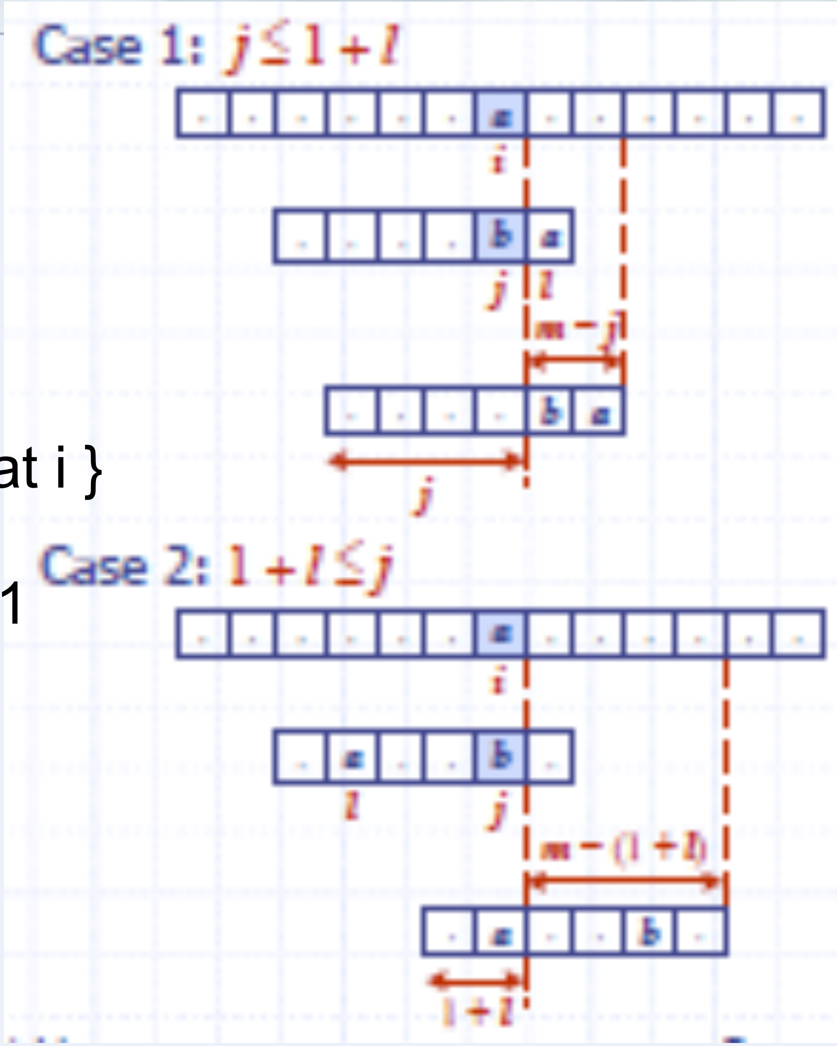
Last Occurrence Function

- The algorithm preprocesses the pattern P and the alphabet Σ to build a last-occurrence matrix L, mapping Σ to integers, where L(c) is defined as
  - the largest index i such that P[i] = c or
  - −1 if no such index exists
- Example : Σ = {a, b, c, d} and P = abacab

C	a	b	c	d
L(C)	4	5	3	-1

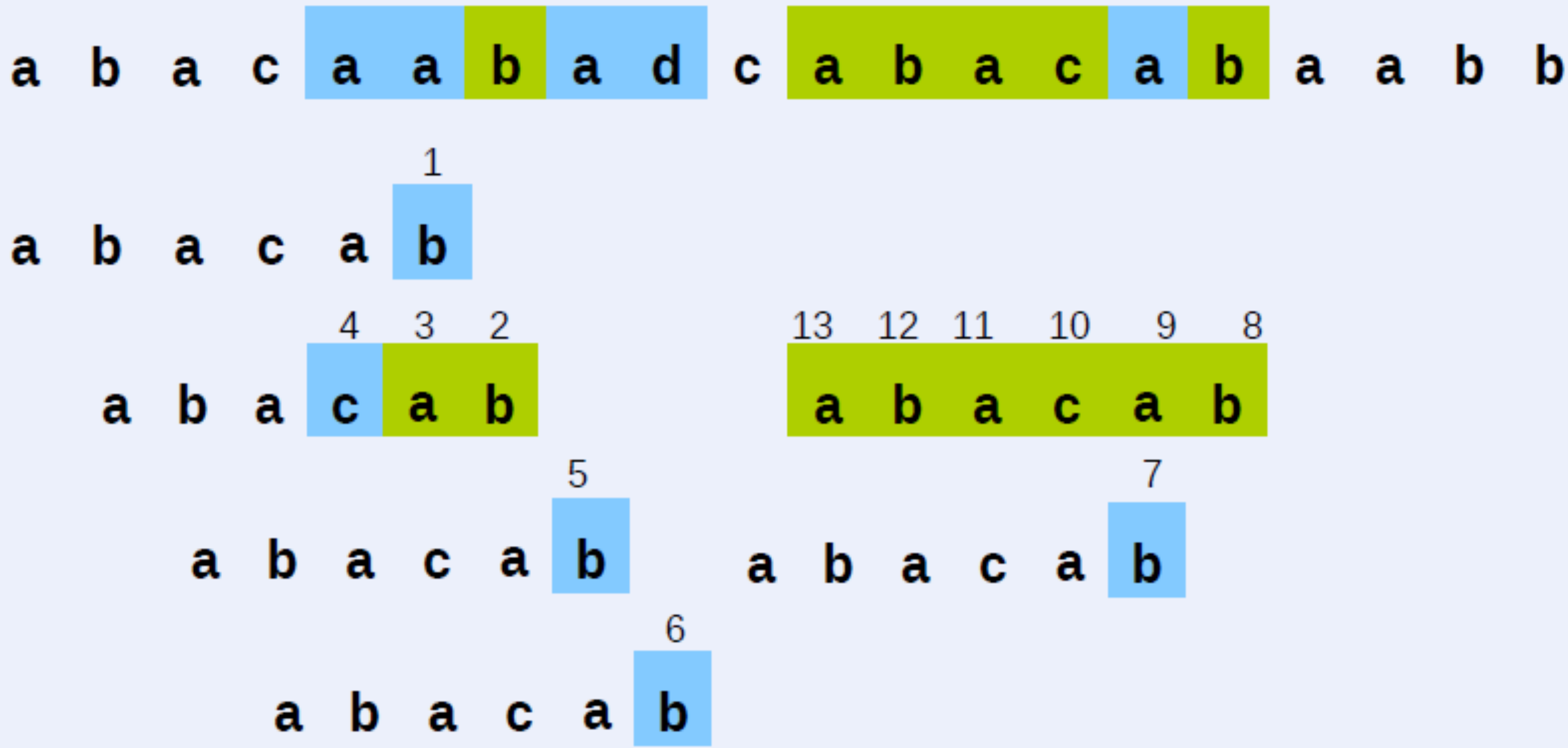
The Boyer Moore Algorithm

```
Algorithm BoyerMooreMatch(T, P, Σ)
    L ← lastOccurenceFunction(P, Σ )
    i ← m − 1; j ← m − 1
    repeat
            if T[i] = P[j]
                    if j = 0
                            return i { match at i }
                    else
                            i ← i − 1; j ← j − 1
            else
                    { character-jump }
                    l ← L[T[i]]
                    i ← i + m – min(j, 1 + l)
                    j ← m − 1
    until i > n − 1
return −1 { no match }
```



Src: Goodrich ch 9.1

Example

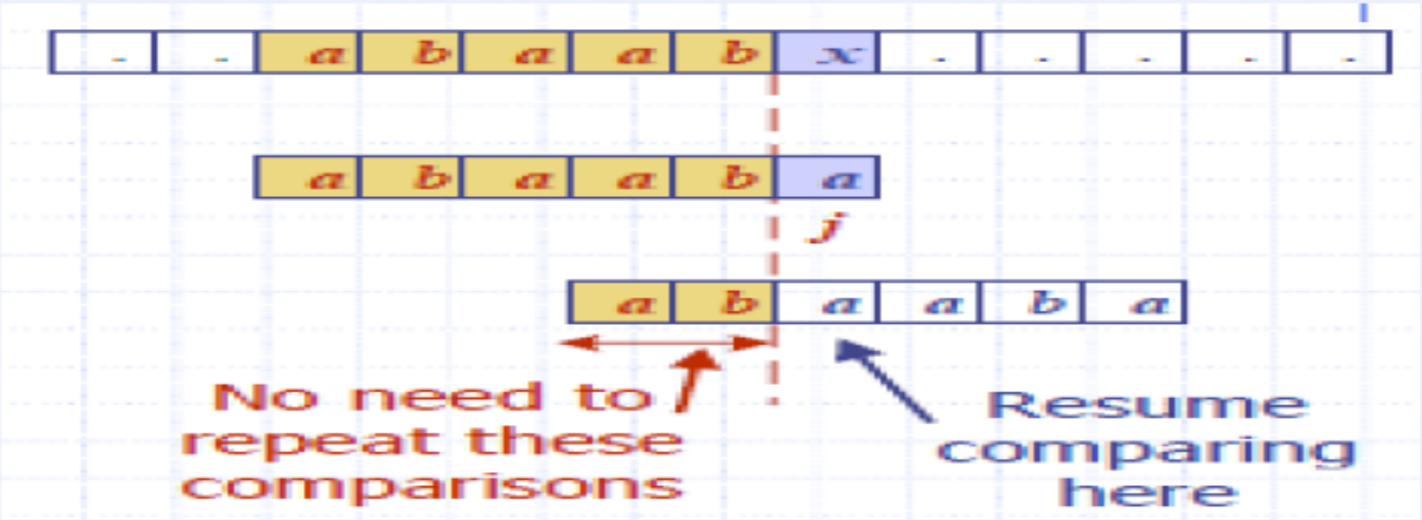

a b a c a a b a d c a b a c a b a a b b
1
a b a c a b
4 3 2
a b a c a b
13 12 11 10 9 8
a b a c a b
5
a b a c a b
7
a b a c a b
6
a b a c a b

Analysis

- Works poorly for certain cases
  - Worst case O(mn)
  - This occurs in images etc
- Works very well for English text

Knuth Morris Pratt's (KMP) Algorithm

- KMP algorithm compares the pattern to the text in left-to-right, but shifts the pattern more intelligently than the brute-force algorithm.
  - When a mismatch occurs, what is the most we can shift the pattern so as to avoid redundant comparisons?



Src: Algorithm Design : Goodrich and Tamasssia

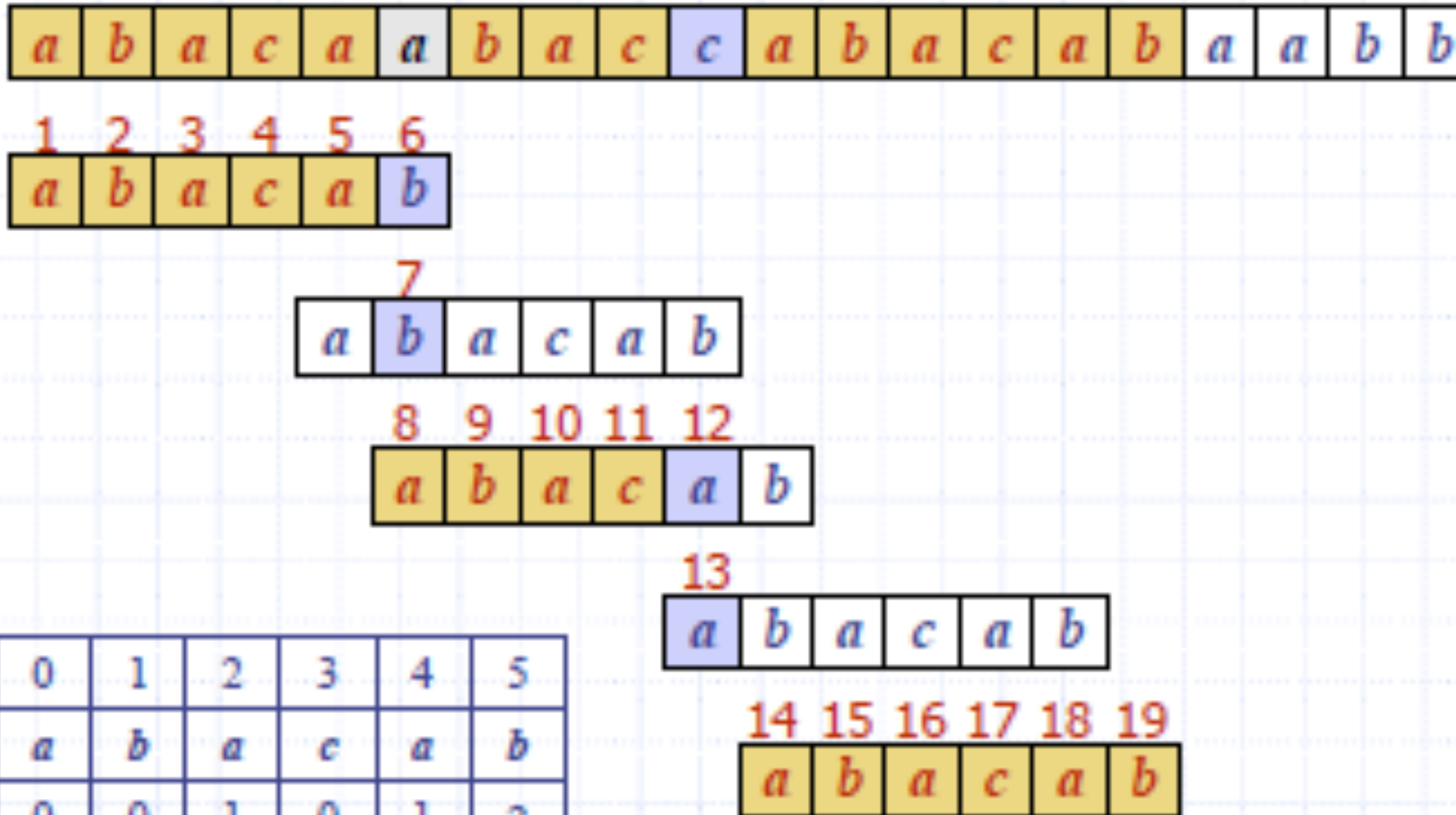
KMP – The Idea

- When searching for AABAAA in AABAABAAAA
  - Mismatch detected at position 5
  - Better to restart at position 3
- Key idea
  - It is possible to decide ahead of time exactly how to restart search
  - This is dependent only only on the pattern
- To decide how far to backup pointer, use failure matrix

KMP Failure Function

- KMP computes the failure function F(j)
  - defined as the size of the largest prefix of P[0..j] that is also a suffix of P[1..j]
- Uses the failure function to skip intelligently
  - If a mismatch occurs at $P[j] \neq T[i]$ set $j \leftarrow F(j-1)$
    - j   = 0 1 2 3 4 5
    - P[j] = a b a a b a
    - F(j) = 0 0 1 1 2 3

Example



j	0	1	2	3	4	5
$P[j]$	a	b	a	c	a	b
$F(j)$	0	0	1	0	1	2

Src: Algorithm Design : Goodrich and Tamasssia

KMP Algorithm

- Algorithm KMPMatch(T, P)

```
F ← failureFunction(P)
i ← 0; j ← 0
while i < n
    if T[i] = P[j]
        if j = m − 1
            return i − j { match }
        else
            i ← i + 1; j ← j + 1
    else
        if j > 0
            j ← F[j − 1]
        else
            i ← i + 1
```

KMP Failure Function

```
Algorithm failureFunction(P)
    F[0] ← 0; i ← 1; j ← 0
    while i < m
            If P[i] = P[j] // {we have matched j + 1 chars}
                F[i] ← j + 1
                i ← i + 1
                j ← j + 1
            else if j > 0 then  //{use failure function to shift P}
                j ← F[j − 1]
            else
                F[i] ← 0 { no match }
                i ← i + 1
```

Exercise

- Give the failure function for the pattern A B R A C A D A B R A
- Show the trace of KMP for the following
  - pattern: AAABAAAB
  - text: AAAAAAAABAAAAAAAAAAAAABAAAB