# Ch32 String Matching

- Introduction
- Naïve Algorithm
- Rabin-Karp Algorithm
- String Matching using Finite Automata
- Knuth-Morris-Pratt (KMP) Algorithm
- Indexing Method: BWT (Suppl.)

To various sources, including Profs. Ananth Grama, Mehmet Koyuturk, Michael Raymer, Wiki sources (pictures), and other noted attributions

# Introduction

- What is *string matching*?
  - Finding all occurrences of a *pattern* in a given *text* (or *body of text*).
- Many applications:
  - While using editor/word processor/browser.
  - Login name & password checking.
  - Virus detection.
  - Header analysis in data communications.
  - DNA sequence analysis.

# History of String Search

- The brute force algorithm
  - invented in the dawn of computer history.
  - re-invented many times, *still* common.
- Knuth & Pratt invented a better one in 1970
  - published 1976 as “Knuth-Morris-Pratt”.
- Boyer & Moore found a better one before 1976
  - Published 1977.
- Karp & Rabin found a “better” one in 1980
  - Published 1987.

| 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 | $\Theta(m)$ | $\Theta(n)$ |
| Boyer-Moore | $\Theta(m)$ | $\Theta(n)$ |

# String-Matching Problem

- The *text* is in an array $T[1..n]$ of length $n$.
- The *pattern* is in an array $P[1..m]$ of length $m$.
- Elements of $T$ and $P$ are characters from a *finite alphabet* $\Sigma$.
  - E.g., $\Sigma = \{0,1\}$ or $\Sigma = \{a, b, \ldots, z\}$.
- Usually $T$ and $P$ are called *strings* of characters.

# String-Matching Problem …contd

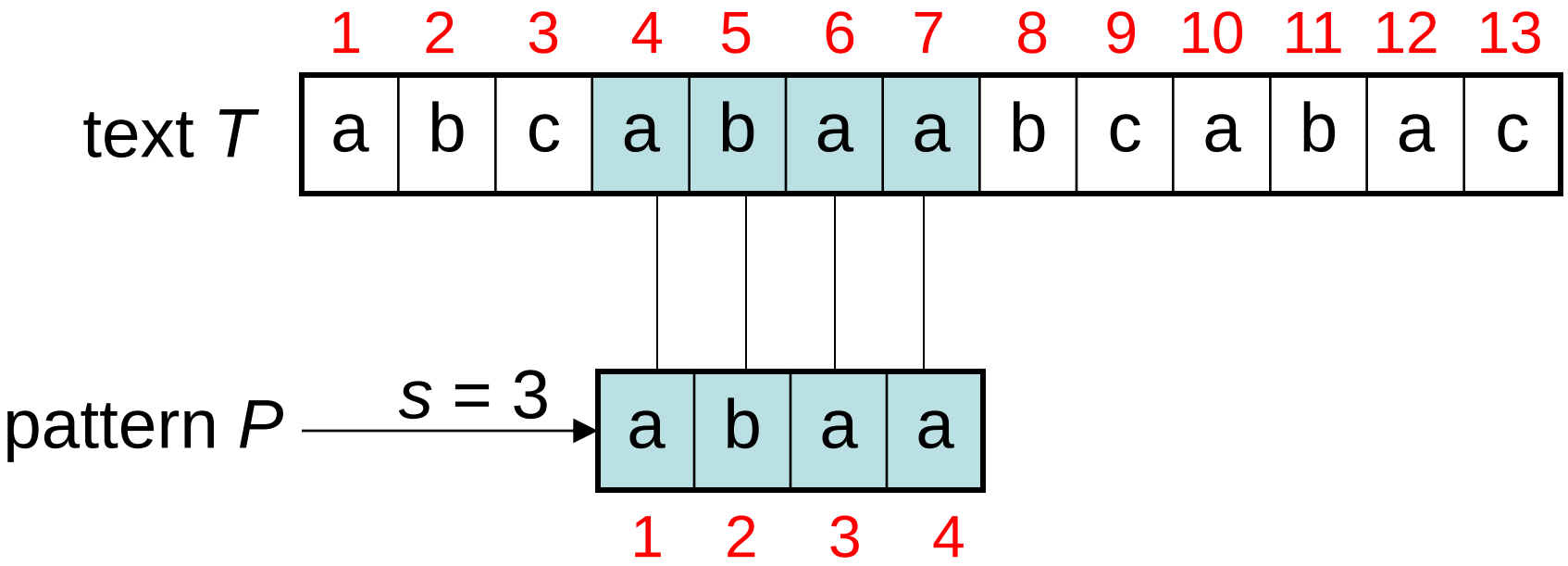
- We say that pattern $P$ *occurs with shift* $s$ in text $T$ if:

  a) $0 \leq s \leq n-m$ and

  *b)* $T[(s+1)..(s+m)] = P[1..m]$.

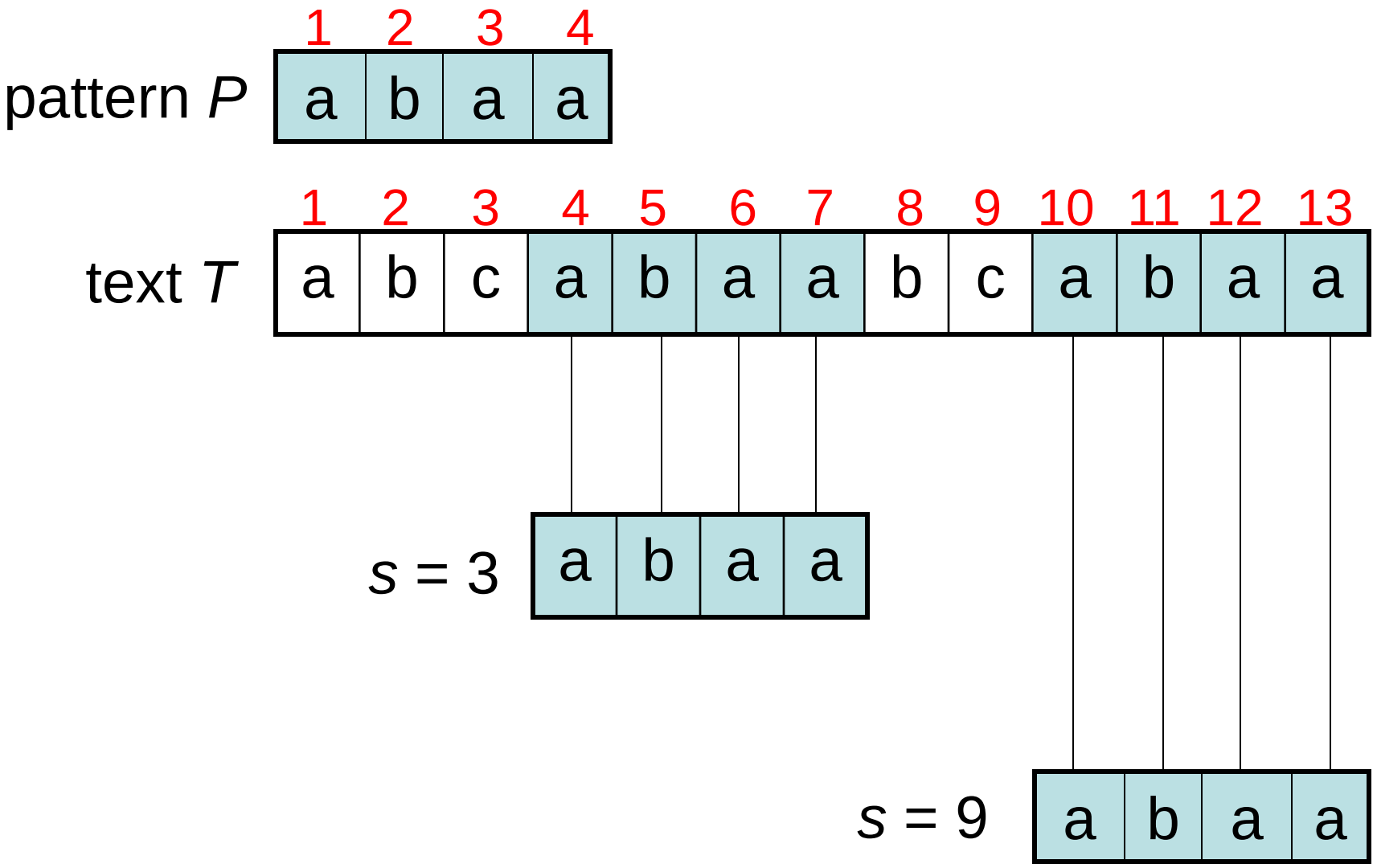
- If $P$ occurs with shift $s$ in $T$, then $s$ is a *valid shift*, otherwise $s$ is an *invalid shift.*
- String-matching problem: finding all valid shifts for a given $T$ and $P$.

# Example 1




shift $s = 3$ is a valid shift
($n$=13, $m$=4 and $0 \le s \le n\text{-}m$ holds)

# Example 2


1 2 3 4
pattern P
a b a a
1 2 3 4 5 6 7 8 9 10 11 12 13
text T
a b c a b a a b c a b a a
s = 3
a b a a
s = 9
a b a a

# Ch32 String Matching

- Introduction
- Naïve Algorithm
- Rabin-Karp Algorithm
- String Matching using Finite Automata
- Knuth-Morris-Pratt (KMP) Algorithm
- Indexing Method: BWT (Suppl.)

# Naïve String-Matching Algorithm

**Input**: Text strings $T$ [1..$n$] and $P$[1..$m$].

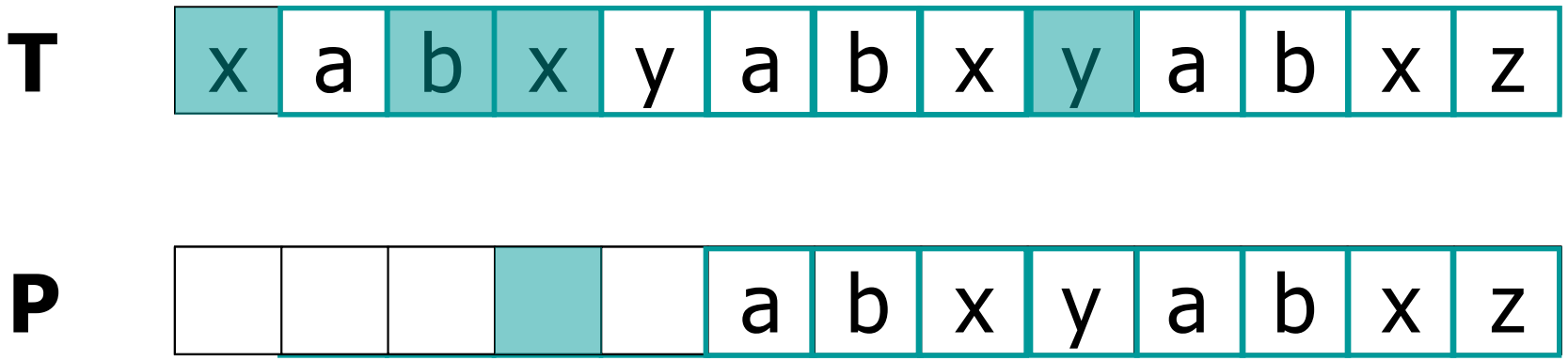
**Result**: All valid shifts displayed.

**NA ÏVE-STRING-MATCHER** ($T$, $P$)

```
n ← length[T]
m ← length[P]
for s ← 0 to n-m
  if P[1..m] = T [(s+1)..(s+m)]
        print "pattern occurs with shift" s
```

# Example

P="abxyabxz" and T="xabxyabxyabxz"



# Worst-case Analysis

- There are $m$ comparisons for each shift in the worst case.
- There are $n$-$m$+1 shifts.
- So, the worst-case running time is $\Theta((n-m+1)m)$.
- Naïve method is inefficient because information from a shift is not used again.

# Ch32 String Matching

- Introduction
- Naïve Algorithm
- Rabin-Karp Algorithm
- String Matching using Finite Automata
- Knuth-Morris-Pratt (KMP) Algorithm
- Indexing Method: BWT (Suppl.)

# Rabin-Karp Algorithm

- Has a worst-case running time of $O((n-m+1)m)$ but average-case is $O(n+m)$.
  - Also works well in practice.
- Based on number-theoretic notion of *modular equivalence.*
- We assume that $\Sigma = \{0,1, 2, \ldots, 9\}$, i.e., each character is a decimal digit.
  - In general, use radix-$d$ where $d = |\Sigma|$.

# Modular Equivalence

- If *(a mod n)* = *(b mod n)*, then we say "*a* is equivalent to *b*, modulo *n*" .
- Denoted by $a \equiv b \pmod{n}$.

- That is, $a \equiv b \pmod{n}$ if *a* and *b* have the same remainder when divided by *n.*
  - E.g., $23 \equiv 37 \equiv -19 \pmod{7}$.

# Rabin-Karp Approach

- We can view a string of $k$ characters (digits) as a length-$k$ decimal number.
  - E.g., the string "31425" corresponds to the decimal number 31,425.
- Given a pattern $P$ [1..$m$], let $p$ denote the corresponding decimal value.
- Given a text $T$ [1..$n$], let $t_s$ denote the decimal value of the length-$m$ substring $T$ [($s$+1)..($s$+$m$)] for $s$=0,1,…,($n$-$m$).

# Rabin-Karp Approach …contd

- $t_s = p$ iff $T[(s+1)..(s+m)] = P[1..m]$.
- $s$ is a valid shift iff $t_s = p$.
- $p$ can be computed in $O(m)$ time.
  - $p = P[m] + 10\,(P[m-1] + 10\,(P[m-2]+\ldots))$.
- $t_0$ can similarly be computed in $O(m)$ time.
- Other $t_1, t_2, \ldots, t_{n-m}$ can be computed in O(n-m) time since $t_{s+1}$ can be computed from $t_s$ in constant time.

# Rabin-Karp Approach …contd

- $t_{s+1} = 10(t_s - 10^{m-1} \cdot T[s+1]) + T[s+m+1]$
  - E.g., if $T$={…,3,1,4,1,5,2,…}, $m$=5 and $t_s$= 31,415, then $t_{s+1} = 10(31415 – 10000 \cdot 3) + 2$
- We can compute $p$, $t_0$, $t_1$, $t_2$,…, $t_{n-m}$ in $O(n+m)$ time.

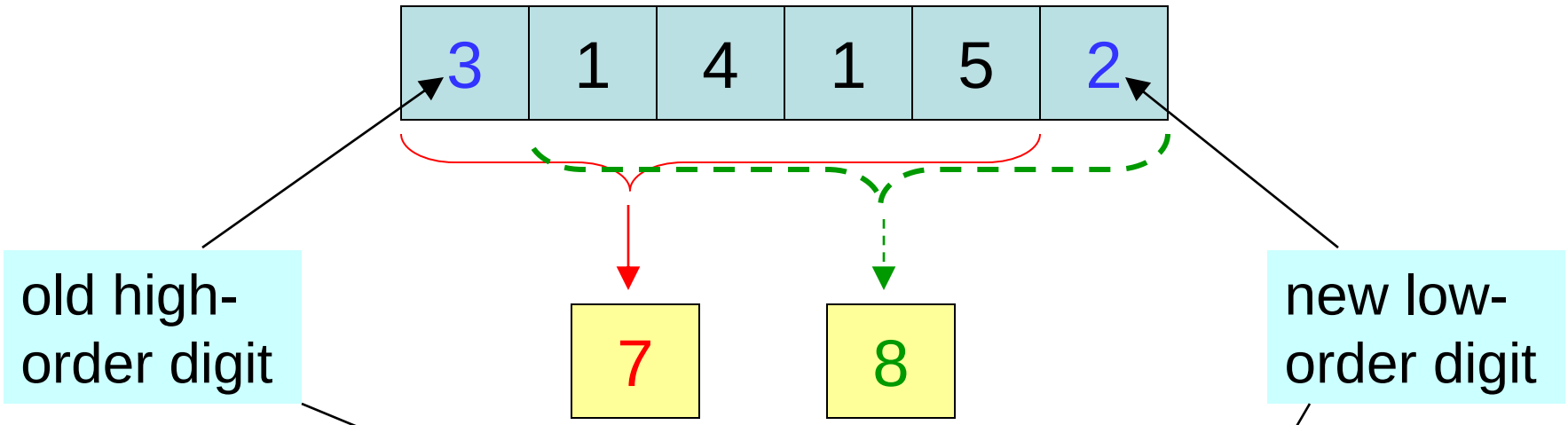
- But…a problem: this is assuming $p$ and $t_s$ are small numbers.
  - They may be too large to work with easily.

# Rabin-Karp Approach …contd

- Solution: we can use modular arithmetic with a suitable modulus, *q.*
  - E.g., $t_{s+1} \equiv 10(t_s - \ldots) + T[s+m+1] \pmod{q}$.

- *q* is chosen as a small *prime number*, e.g., 13 for radix 10.
  - Generally, if the radix is *d*, then *dq* should fit within one computer word.

# How values modulo 13 are computed




$$14152 \equiv (31415 - 3 \cdot 10000) \cdot 10 + 2 \pmod{13}$$

$$\equiv (7 - 3 \cdot 3) \cdot 10 + 2 \pmod{13}$$

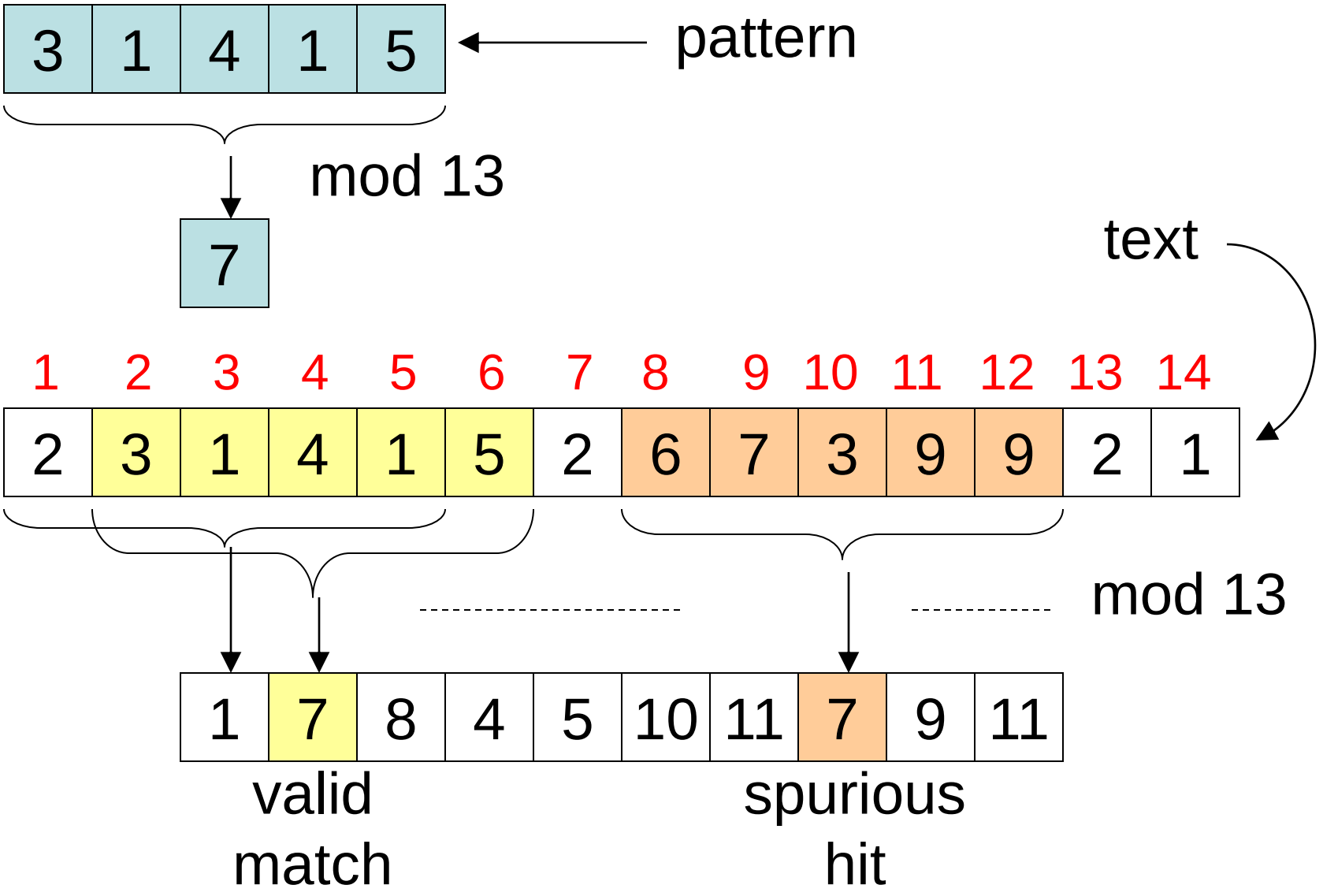
$$\equiv 8 \pmod{13}$$

# Problem of Spurious Hits

- $t_s \equiv p \pmod{q}$ does not imply that $t_s = p$.
  - Modular equivalence does not necessarily mean that two integers are equal.
- A case in which $t_s \equiv p \pmod{q}$ when $t_s \neq p$ is called a *spurious hit.*

- On the other hand, if two integers are not modular equivalent, then they cannot be equal.

# Example


3 1 4 1 5
pattern
mod 13
7
text
1 2 3 4 5 6 7 8 9 10 11 12 13 14
2 3 1 4 1 5 2 6 7 3 9 9 2 1
mod 13
1 7 8 4 5 10 11 7 9 11
valid match
spurious hit

# Rabin-Karp Algorithm

- Basic structure like the naïve algorithm, but uses modular arithmetic as described.
- For each *hit*, i.e., for each $s$ where $t_s \equiv p \pmod{q}$, verify character by character. whether $s$ is a valid shift or a spurious hit
- In the worst case, every shift is verified.
  - Running time can be shown as $O((n-m+1)m)$.
- Average-case running time is $O(n+m)$.

# Ch32 String Matching

- Introduction
- Naïve Algorithm
- Rabin-Karp Algorithm
- String Matching using Finite Automata
- Knuth-Morris-Pratt (KMP) Algorithm
- Indexing Method: BWT (Suppl.)

# 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 set of *accepting states*.
  - $\Sigma$ is a finite *input alphabet*.
  - $\delta$ is the *transition function* that gives the next state for a given current state and input.

# How a Finite Automaton Works

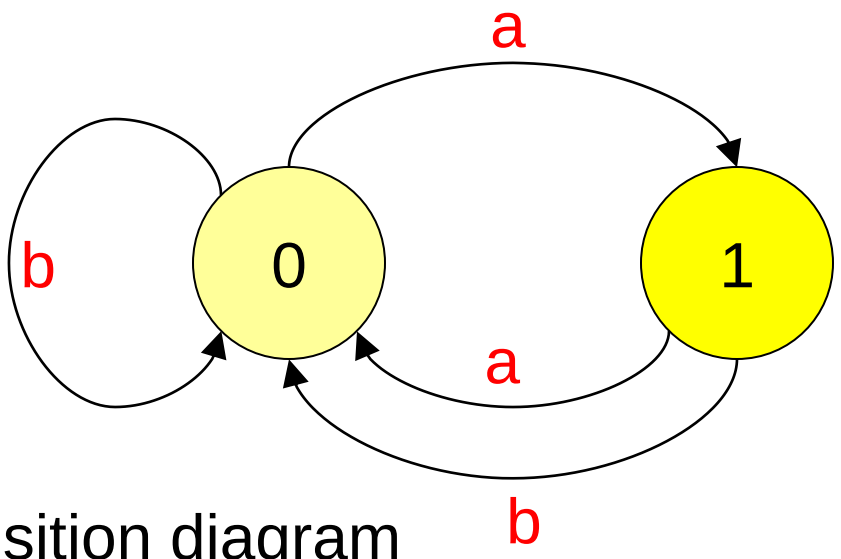
- The finite automaton $M$ begins in state $q_0$.
- Reads characters from $\sum$ one at a time.
- If $M$ is in state $q$ and reads input character $a$, $M$ moves to state $\delta(q,a)$.
- If its current state $q$ is in $A$, $M$ is said to have *accepted* the string read so far.
- An input string that is not accepted is said to be *rejected.*

# Example

- $Q = \{0,1\}$, $q_0 = 0$, A={1}, $\Sigma = \{a, b\}$.
- $\delta(q,a)$ shown in the transition table/diagram.
- This accepts strings that end in an odd number of a's; e.g., abbaaa is accepted, aa is rejected.

| state | input a | input b |
|---|---|---|
| 0 | 1 | 0 |
| 1 | 0 | 0 |

transition table




transition diagram

# String-Matching Automata

- Given the pattern $P$ [1..$m$], build a finite automaton $M$.
  - The state set is $Q$={0, 1, 2, …, $m$}.
  - The start state is 0.
  - The only accepting state is $m$.

- Time to build $M$ can be large if $\sum$ is large.

# String-Matching Automata …contd

- Scan the text string $T$ [1..$n$] to find all occurrences of the pattern $P$ [1..$m$].

- String matching is efficient: $\Theta(n)$.
  - Each character is examined exactly once.
  - Constant time for each character.
- But …time to compute $\delta$ is $O(m\ |\Sigma|)$.
  - $\delta$ Has $O(m\ |\Sigma|\ )$ entries.

# Algorithm

**Input**: Text string $T$ [1..$n$], $\delta$ and $m$

**Result**: All valid shifts displayed

**FINITE-AUTOMATON-MATCHER** ($T$, $m$, $\delta$)

$n \leftarrow length[T]$

$q \leftarrow 0$

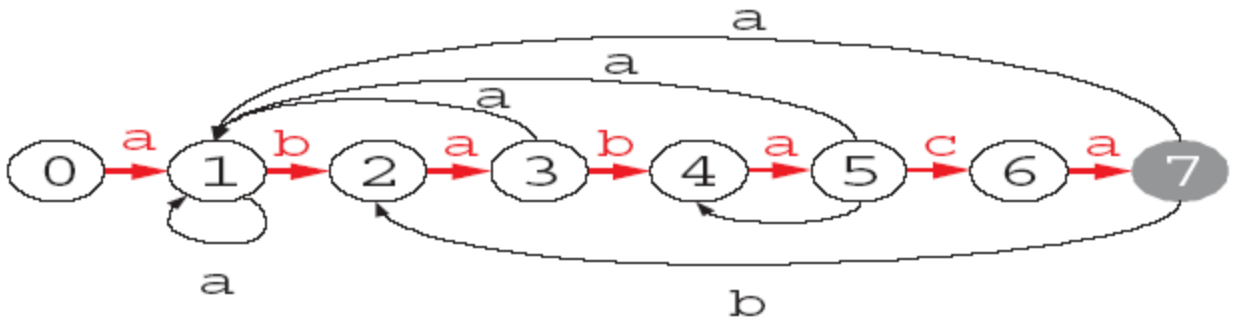
**for** $i \leftarrow 1$ **to** $n$

$q \leftarrow \delta\ (q, T\ [i])$

**if** $q = m$

print "pattern occurs with shift" $i$-$m$

# Example




| state | $a$ | $b$ | $c$ | $P$ |
|---|---|---|---|---|
| 0 | 1 | 0 | 0 | $a$ |
| 1 | 1 | 2 | 0 | $b$ |
| 2 | 3 | 0 | 0 | $a$ |
| 3 | 1 | 4 | 0 | $b$ |
| 4 | 5 | 0 | 0 | $a$ |
| 5 | 1 | 4 | 6 | $c$ |
| 6 | 7 | 0 | 0 | $a$ |
| 7 | 1 | 2 | 0 | |

| $i$ | — | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $T[i]$ | — | $a$ | $b$ | $a$ | $b$ | $a$ | $b$ | $a$ | $c$ | $a$ | $b$ | $a$ |
| $\phi(T_i)$ | 0 | 1 | 2 | 3 | 4 | 5 | 4 | 5 | 6 | 7 | 2 | 3 |

# Ch32 String Matching

- Introduction
- Naïve Algorithm
- Rabin-Karp Algorithm
- String Matching using Finite Automata
- Knuth-Morris-Pratt (KMP) Algorithm
- Indexing Method: BWT (Suppl.)

# Knuth-Morris-Pratt (KMP) Method

- Avoids computing $\delta$ (transition function).
- Instead computes a *prefix function* $\pi$ in $O(m)$ time.
  - $\pi$ has only $m$ entries.

- Prefix function stores info about how the pattern matches against shifts of itself.
  - Can avoid testing useless shifts.

# Terminology/Notations

- String *w* is a *prefix* of string *x*, if *x*=*wy* for some string *y* (e.g., “srilan” of “srilanka”).
- String *w* is a *suffix* of string *x*, if *x*=*yw* for some string *y* (e.g., “anka” of “srilanka”).
- The *k*-character prefix of the pattern *P* [1..*m*] denoted by $P_k$.
  - E.g., $P_0 = \varepsilon$, $P_m = P = P$ [1..*m*].

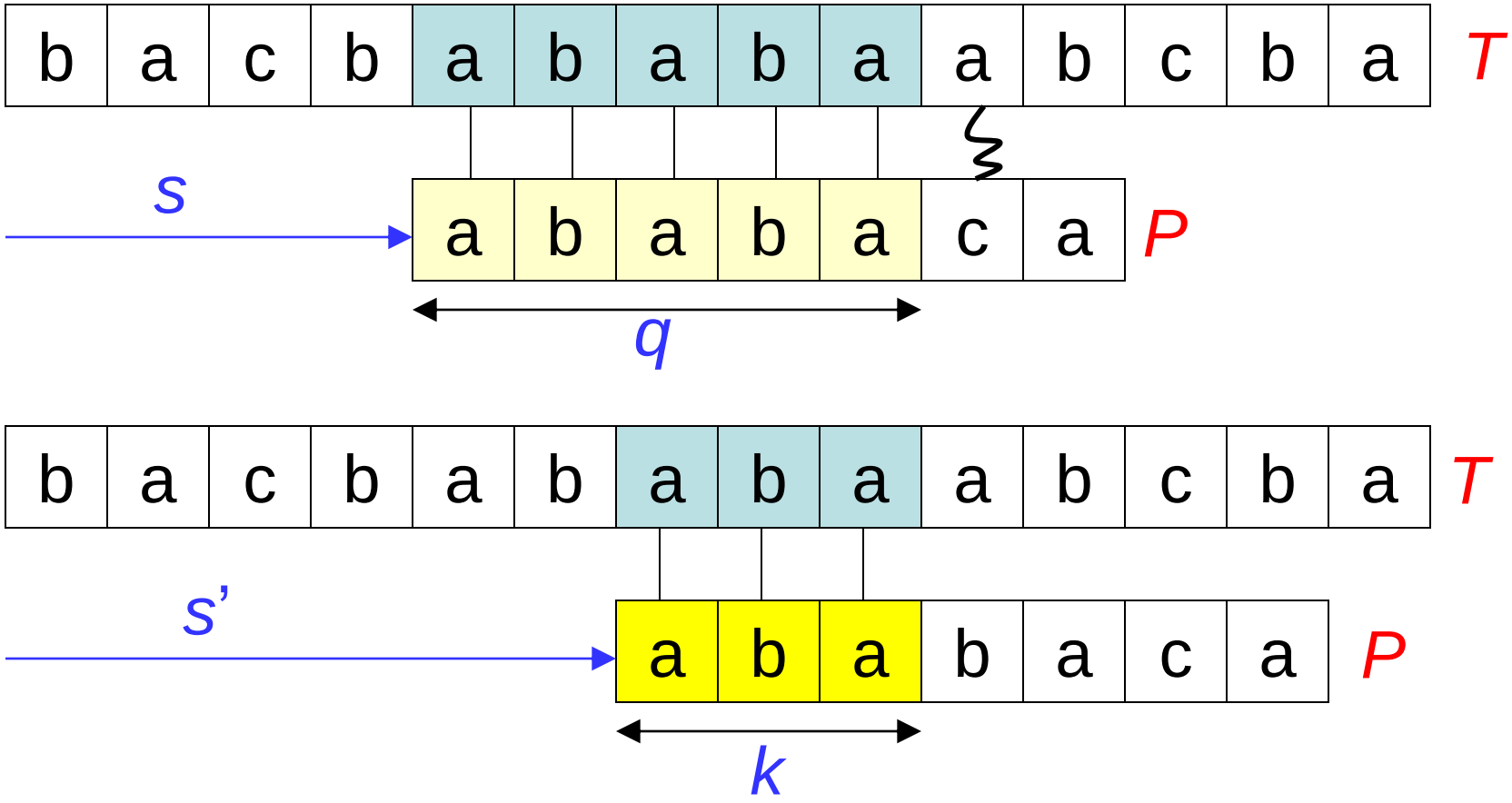
# Prefix Function for a Pattern

- Given that pattern prefix $P[1..q]$ matches text characters $T[(s+1)..(s+q)]$, what is the least shift $s' > s$ such that

  $P[1..k] = T[(s'+1)..(s'+k)]$ where $s'+k=s+q$?

- At the new shift $s'$, no need to compare the first $k$ characters of $P$ with corresponding characters of $T$.
  - Since we know that they match.

# Prefix Function: Example 1



# Prefix Function: Example 2

| $i$ | 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 |

$\pi[q] = \max \{ k \mid k<q \text{ and } P_k \text{ is a suffix of } P_q \}$

Illustration: given a String ‘S’ and pattern ‘p’ as follows:

S

| b | a | c | b | a | b | a | b | a | b | a | c | a | c | a |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

p

| a | b | a | b | a | c | a |
|---|---|---|---|---|---|---|

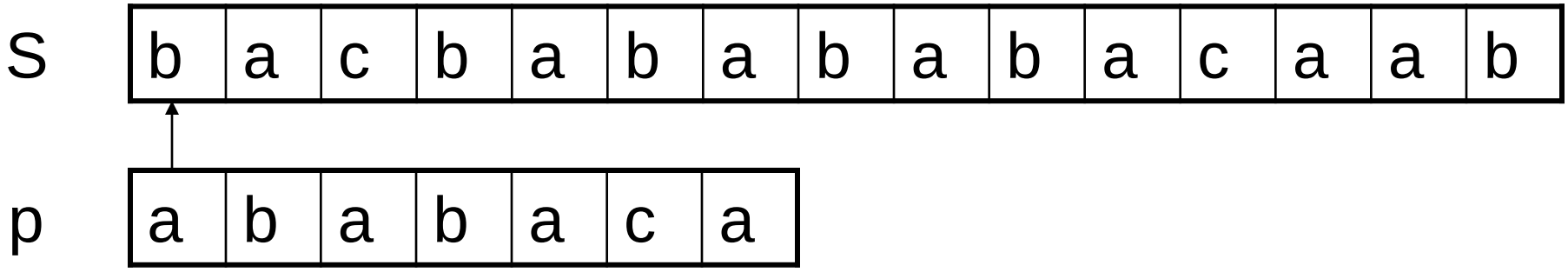
Let us execute the KMP algorithm to find whether ‘p’ occurs in ‘S’.

*For ‘p’ the prefix function, Π was computed previously and is as follows:*

| q | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| p | a | b | a | b | a | c | a |
| Π | 0 | 0 | 1 | 2 | 3 | 0 | 1 |

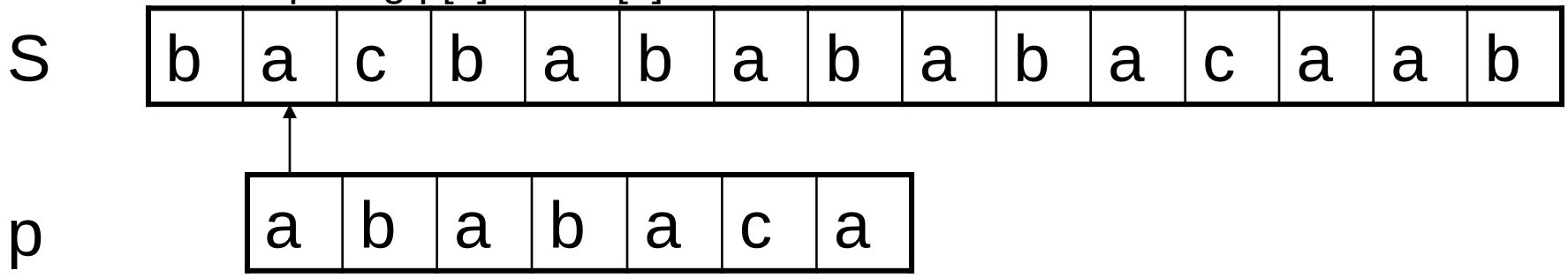
Initially: n = size of S = 15;
m = size of p = 7

Step 1: i = 1, q = 0
comparing p[1] with S[1]




P[1] does not match with S[1]. ‘p’ will be shifted one position to the right.

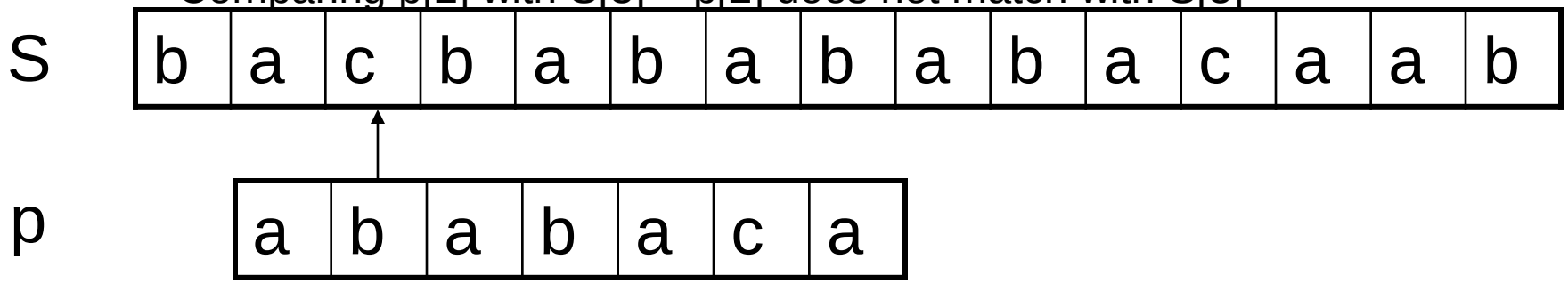
Step 2: i = 2, q = 0
comparing p[1] with S[2]




P[1] matches S[2]. Since there is a match, p is not shifted.

Step 3: i = 3, q = 1

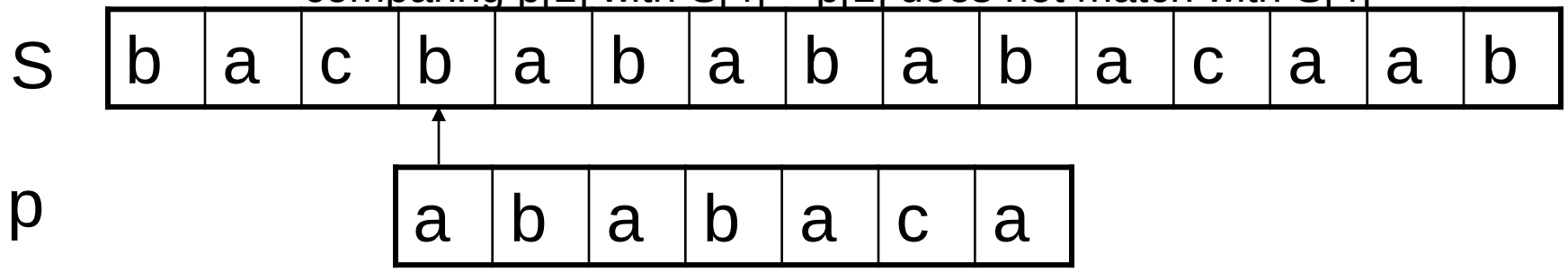
Comparing p[2] with S[3]  p[2] does not match with S[3]




Backtracking on p, comparing p[1] and S[3]

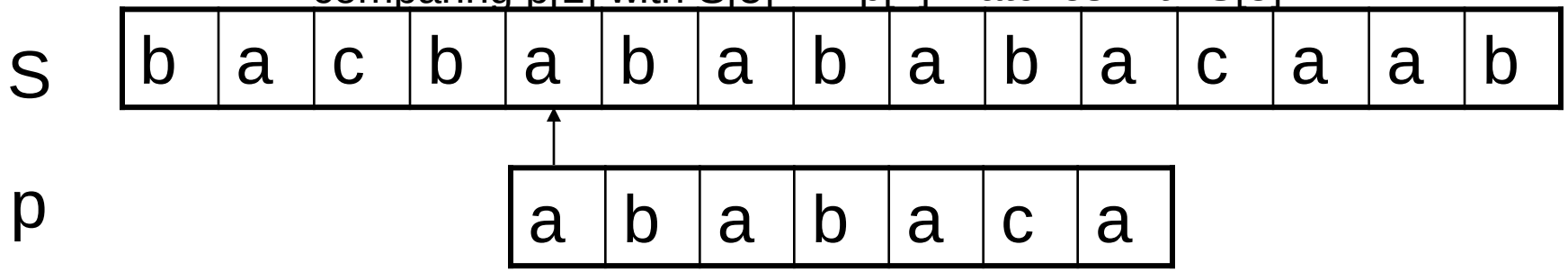
Step 4: i = 4, q = 0

comparing p[1] with S[4]  p[1] does not match with S[4]



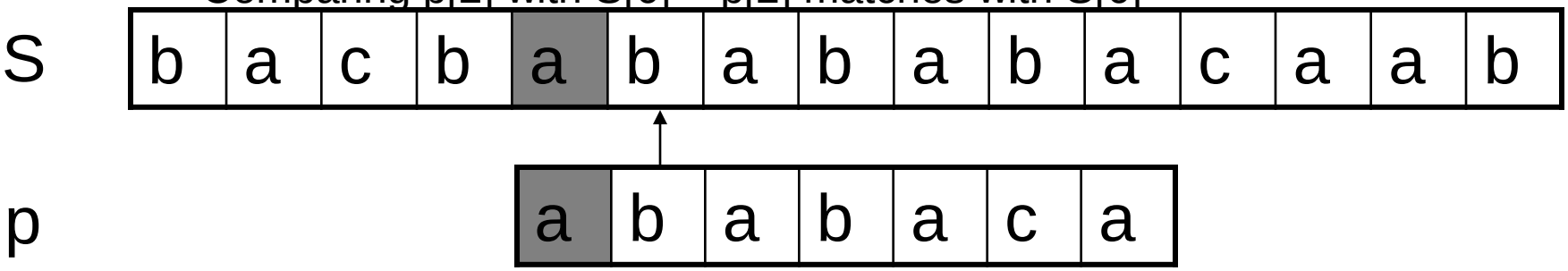

Step 5: i = 5, q = 0

comparing p[1] with S[5]  p[1] matches with S[5]



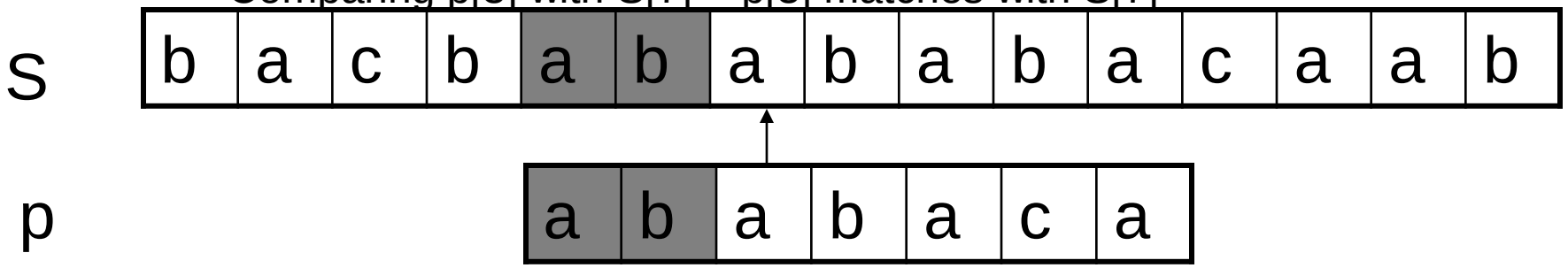
Step 6: i = 6, q = 1

Comparing p[2] with S[6]  p[2] matches with S[6]



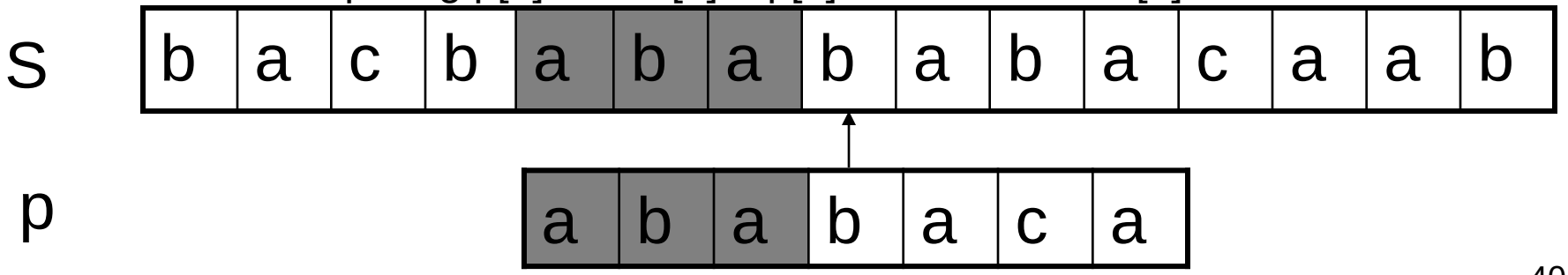

Step 7: i = 7, q = 2

Comparing p[3] with S[7]  p[3] matches with S[7]



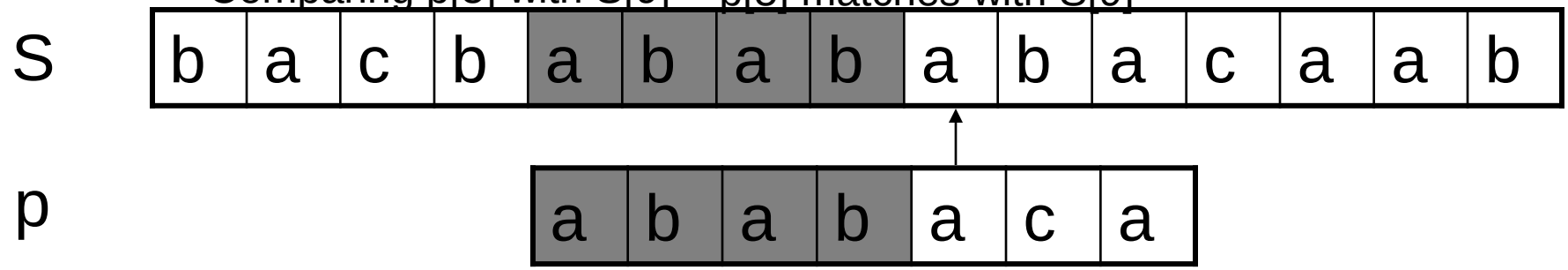

Step 8: i = 8, q = 3

Comparing p[4] with S[8]  p[4] matches with S[8]



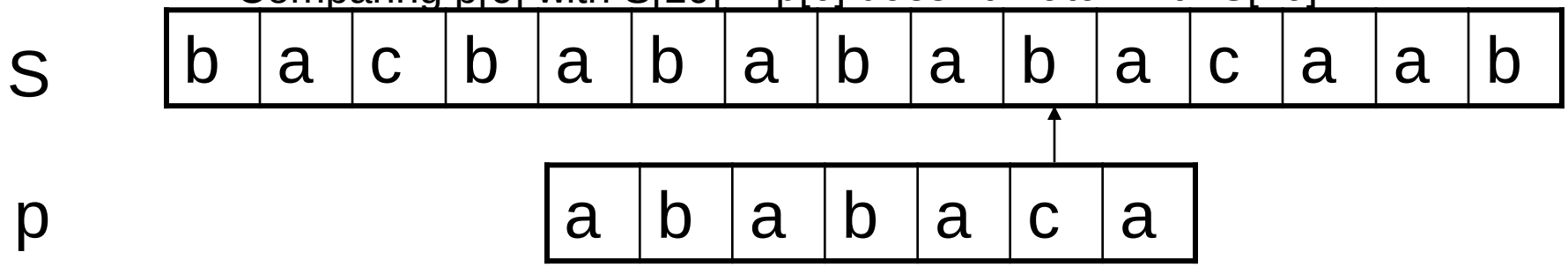
Step 9: i = 9, q = 4

Comparing p[5] with S[9]  p[5] matches with S[9]




Step 10: i = 10, q = 5

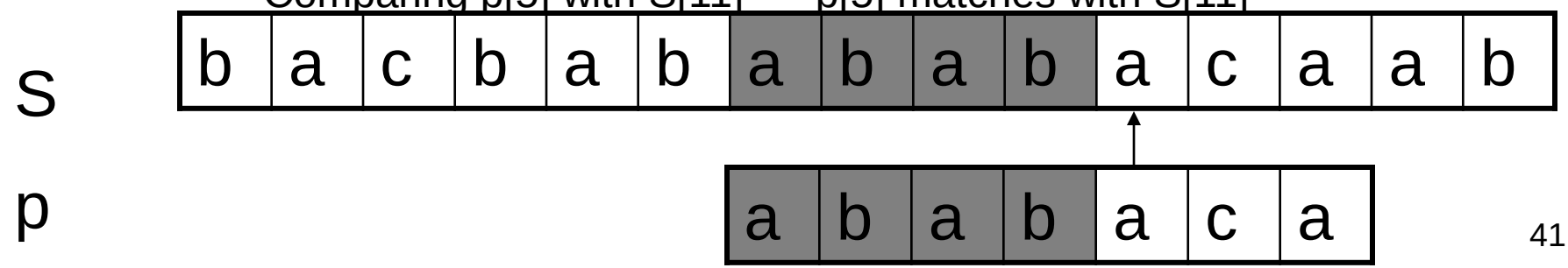
Comparing p[6] with S[10]  p[6] doesn't match with S[10]



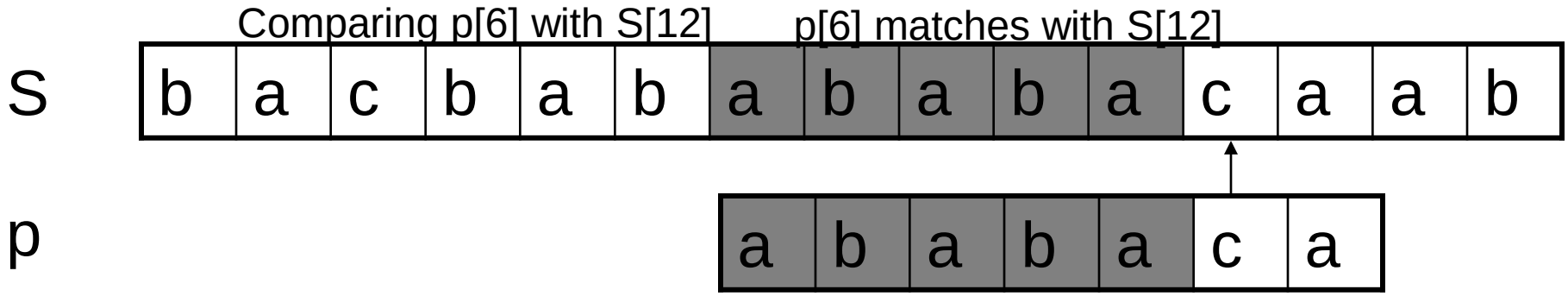

Backtracking on p, comparing p[4] with S[10] because after mismatch q = Π[5] = 3

Step 11: i = 11, q = 4

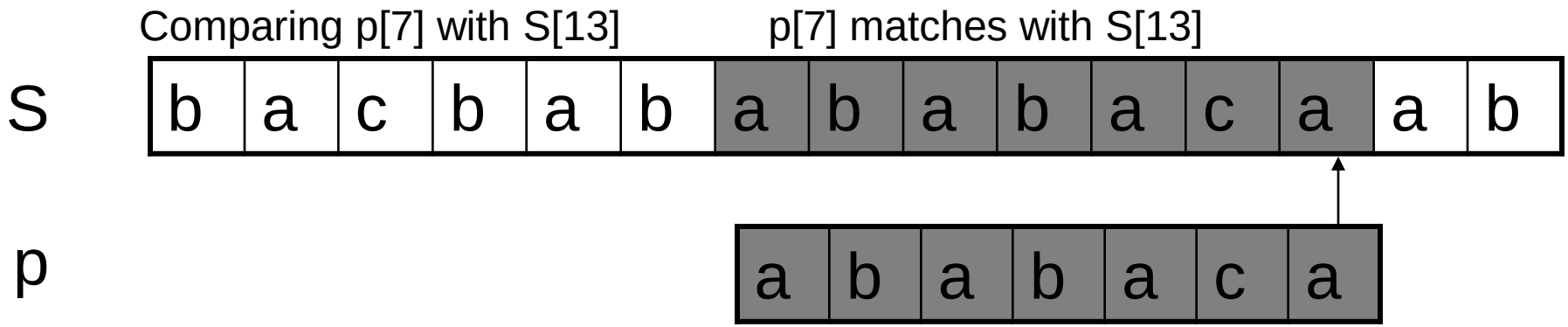
Comparing p[5] with S[11]  p[5] matches with S[11]



Step 12: i = 12, q = 5




Step 13: i = 13, q = 6




Pattern ‘p’ has been found to completely occur in string ‘S’. The total number of shifts that took place for the match to be found are: i – m = 13 – 7 = 6 shifts.

# Knuth-Morris-Pratt (KMP) Algorithm

- Information stored in prefix function
  - Can speed up both the naïve algorithm and the finite-automaton matcher.
- KMP Algorithm
  - 2 parts: KMP-MATCHER, PREFIX.
- Running time
  - PREFIX takes $O(m)$.
  - KMP-MATCHER takes $O(m+n)$.

# Boyer-Moore Algorithm

- Published in 1977.
- The longer the pattern is, the faster it works.
- Starts from the end of pattern, while KMP starts from the beginning.
- Works best for character string, while KMP works best for binary string.
- KMP and Boyer-Moore
  - Preprocessing existing patterns.
  - Searching patterns in input strings.

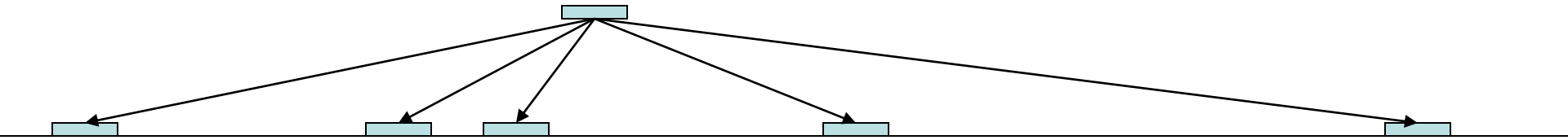
# Ch32 String Matching

- Introduction
- Naïve Algorithm
- Rabin-Karp Algorithm
- String Matching using Finite Automata
- Knuth-Morris-Pratt (KMP) Algorithm
- Indexing Method: BWT (Suppl.)

# Short read mapping

- Input:
  - A reference genome.
  - A collection of many 25-100bp reads.
  - User-specified parameters (best or all mapping…).
- Output:
  - One or more genomic coordinates for each read.

Multiple mapping



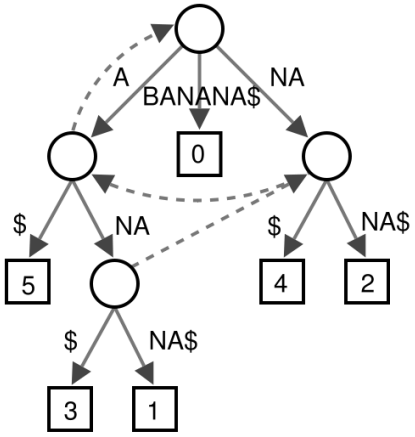
# Mapping Reads Review

- Hash Table (Lookup table): exact match
  - Fast, but only for with fixed length. [O(α(n)N), lookup time α(n)]
- Suffix Trees or Suffix Array: exact match
  - Can handle patterns with variable length. [O(mN)]
  - Constructing suffix array require at least $n\lceil \log_2 n \rceil$ bits of working space.
- Dynamic Programming (Smith Waterman): approximate match
  - Mathematically optimal solution for Indels (插入/删除) .
  - Slow, needs filter out impossible positions in practice. [O(*mnN*)]
- FM-index with Burrows-Wheeler Transform: exact match
  - Fast for small alphabet. [O($mN\log_2\alpha$), α is the size of alphabet]
  - Memory efficient, total is less than 1.5GB for human genome.

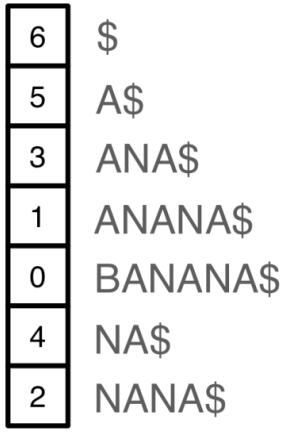
Where *m* is the length of reads, *n* is the length of genome, *N* is the number of reads.

# Indexing (1)

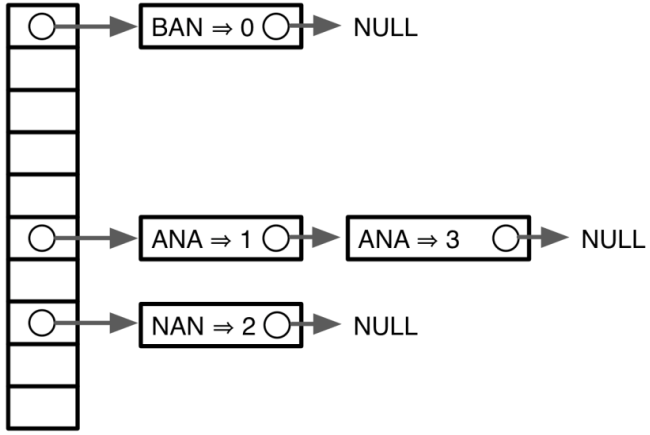
- *Indexing* is required




Suffix tree




Suffix array



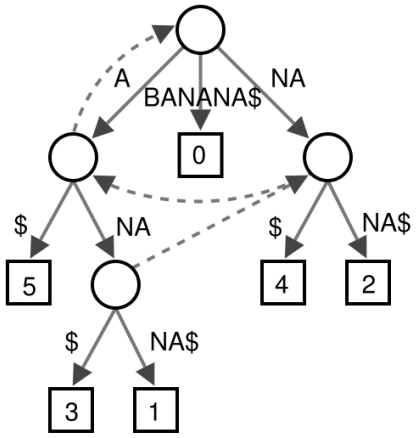

Seed hash tables

Many variants, incl. spaced seeds

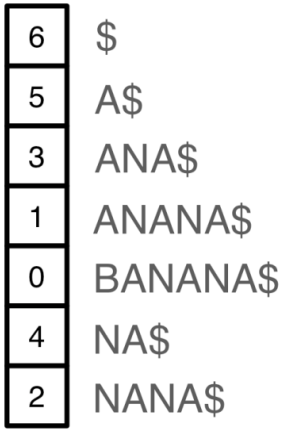
- Choice of index is key to performance.

# Indexing (2)

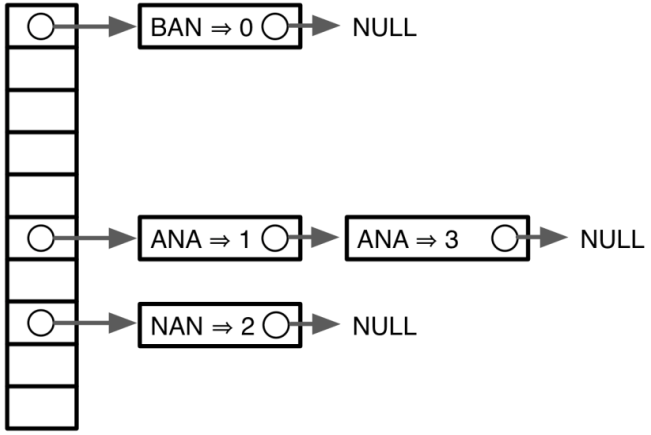
- Genome indices can be big. For human:




> 35 GBs




> 12 GBs

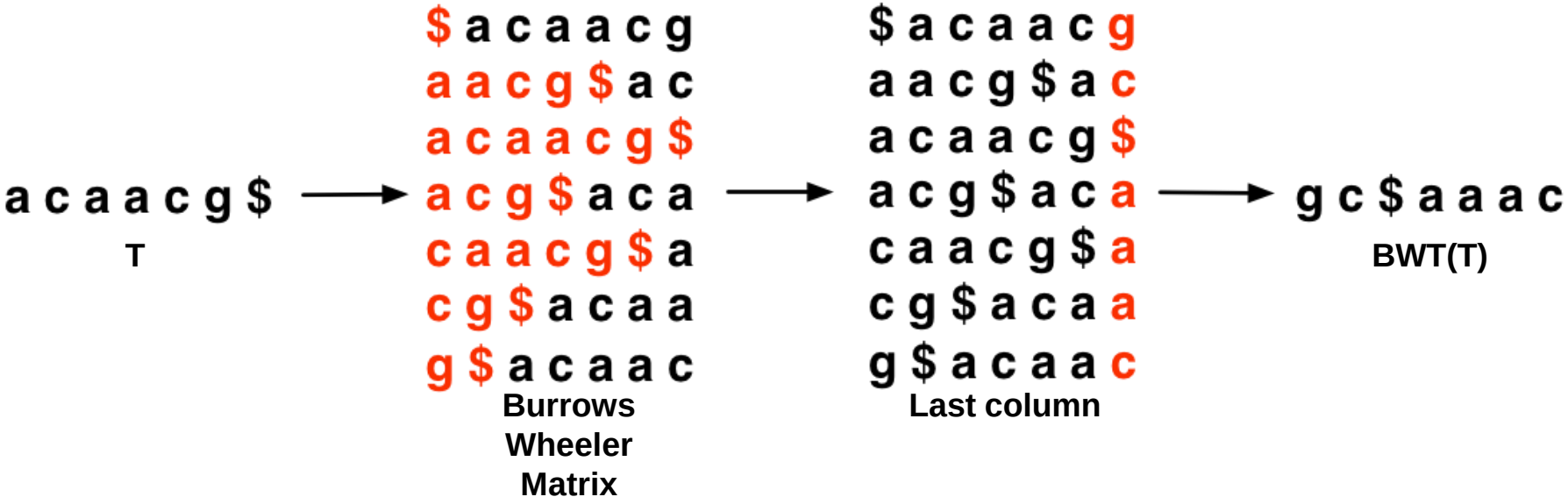



> 12 GBs

- Large indices necessitate painful compromises:
  1. Require big-memory machine
  2. Use secondary storage
  3. Build new index each run
  4. Subindex and do multiple passes

# Building: From T to BWT(T)

- Reversible permutation used originally in compression.

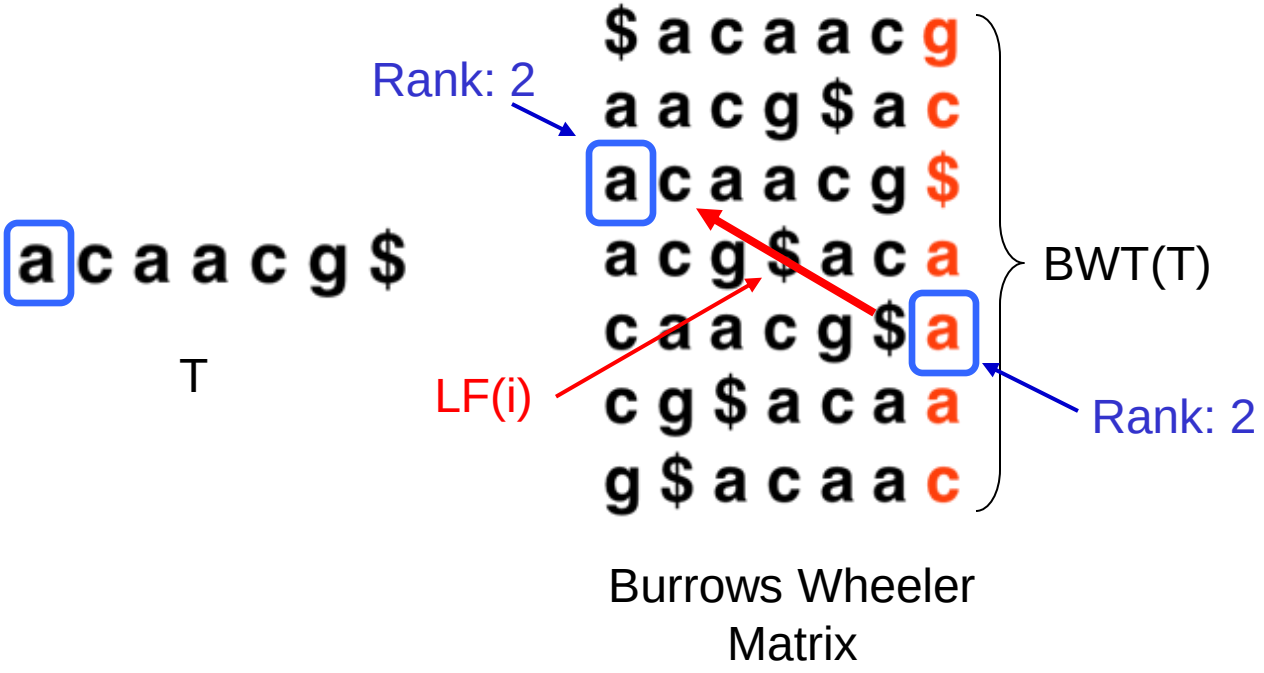



- Once BWT(T) is built, *all else shown here is discarded.*
  - Matrix will be shown for illustration only.

Burrows M, Wheeler DJ: A block sorting lossless data compression algorithm. *Digital Equipment Corporation, Palo Alto, CA* 1994, Technical Report 124; 1994

# LF Mapping of BWT

- Property that makes BWT(T) reversible is "LF Mapping".
  - Property: $i^{th}$ occurrence of a character in **L**ast column is same *text* occurrence as the $i^{th}$ occurrence in **F**irst column.
  - E.g. LF(5)=3 is the map from last row to first row, where 5 is line# of last and 3 is line# of first for the same a of rank 2.



# Recover: From BWT(T) to T

- To recreate T from BWT(T), repeatedly apply rule:
  - **T** = **BWT**[ **LF**(i) ] + **T**; j = **LF**(i).
  - Where **LF**(i) maps row i to row j whose first column character corresponds to j's last column per LF Mapping.


- Could be called "unpermute (解置换)" or "walk-left" algorithm.

# FM Index

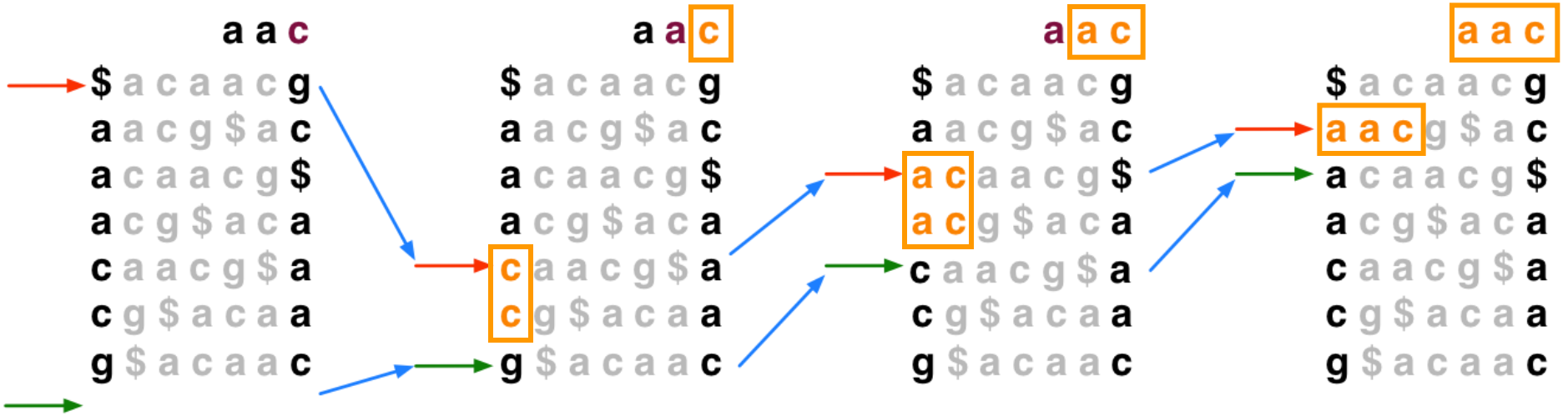
- Ferragina & Manzini propose “FM Index” based on BWT.

- Observed:
  - LF Mapping also allows *exact matching* within T.
  - **LF**(i) can be made fast with *checkpointing.*
  - …and more (see FOCS paper).


- Ferragina P, Manzini G: Opportunistic data structures with applications. *FOCS. IEEE Computer Society; 2000.*
- Ferragina P, Manzini G: An experimental study of an opportunistic index. *SIAM symposium on Discrete algorithms*. Washington, D.C.; 2001.

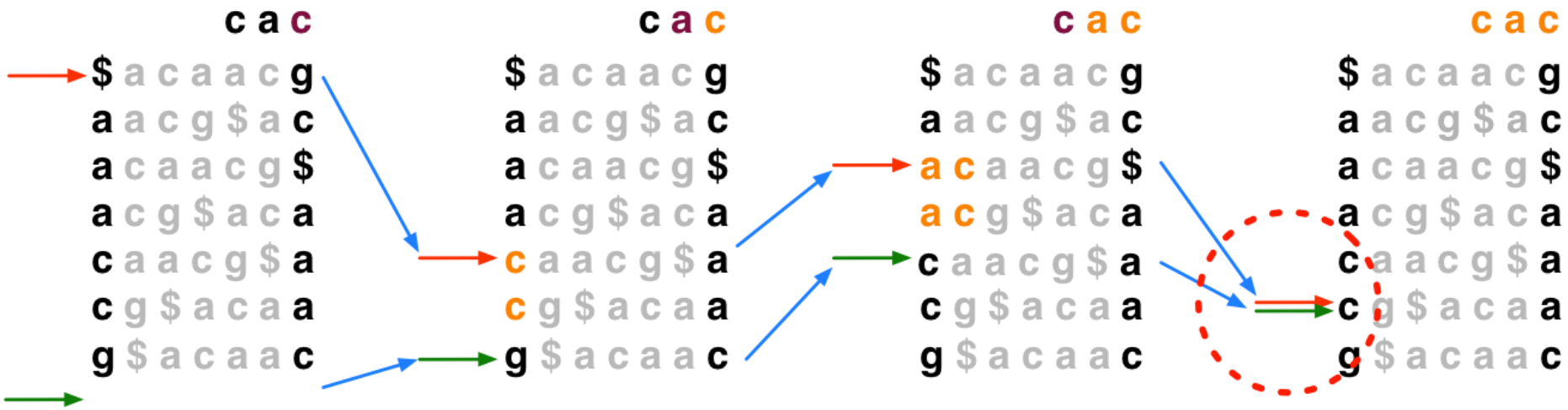
# Searching with FM Index: Existed

- To match Q in T using BWT(T), repeatedly apply rule:
  - **top** = **LF**(**top**, **qc**)  //also by sp;  **bot** = **LF**(**bot**, **qc**)  //also by ep
  - Where **qc** is the next character in Q (right-to-left) and **LF**(i, **qc**) maps row i to the row whose first column character corresponds to i's last column character *as if it were **qc**.*




- In progressive rounds, **top** & **bot** delimit the range of rows beginning with progressively longer suffixes of Q.

# Searching with FM Index: Inexisted




- If range becomes empty (**top** = **bot**) the query suffix (and therefore the query) does not occur in the text.

# Auxiliary data structures

Key for efficient pattern matching: how to find the corresponding chars in the first column efficiently, in terms of both time and space.

| | a | c | g | T |
|---|---|---|---|---|
| rank | 1 | 5 | 8 | 11 |

| | SA | | BWT |
|---|---|---|---|
| | | **$**agcagcagac**t** | t |
| 1 | 9 | **a**ct**$**agcagca**g** | g |
| 2 | 7 | **a**gact**$**agcag**c** | c |
| 3 | 4 | **a**gcagact**$**ag**c** | c |
| 4 | 1 | **a**gcagcagact**$** | $ |
| 5 | 6 | **c**agact**$**agca**g** | g |
| 6 | 3 | **c**agcagact**$**a**g** | g |
| 7 | 10 | **c**t**$**agcagcag**a** | a |
| 8 | 8 | **g**act**$**agcagc**a** | a |
| 9 | 5 | **g**cagact**$**agc**a** | a |
| 10 | 2 | **g**cagcagact**$**a | a |
| 11 | 11 | **t**$agcagcaga**c** | c |

| a | c | g | t |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 |
| 0 | 2 | 1 | 1 |
| 0 | 2 | 1 | 1 |
| 0 | 2 | 2 | 1 |
| 0 | 2 | 3 | 1 |
| 1 | 2 | 3 | 1 |
| 2 | 2 | 3 | 1 |
| 3 | 2 | 3 | 1 |
| 4 | 2 | 3 | 1 |
| 4 | 3 | 3 | 1 |

FM indices

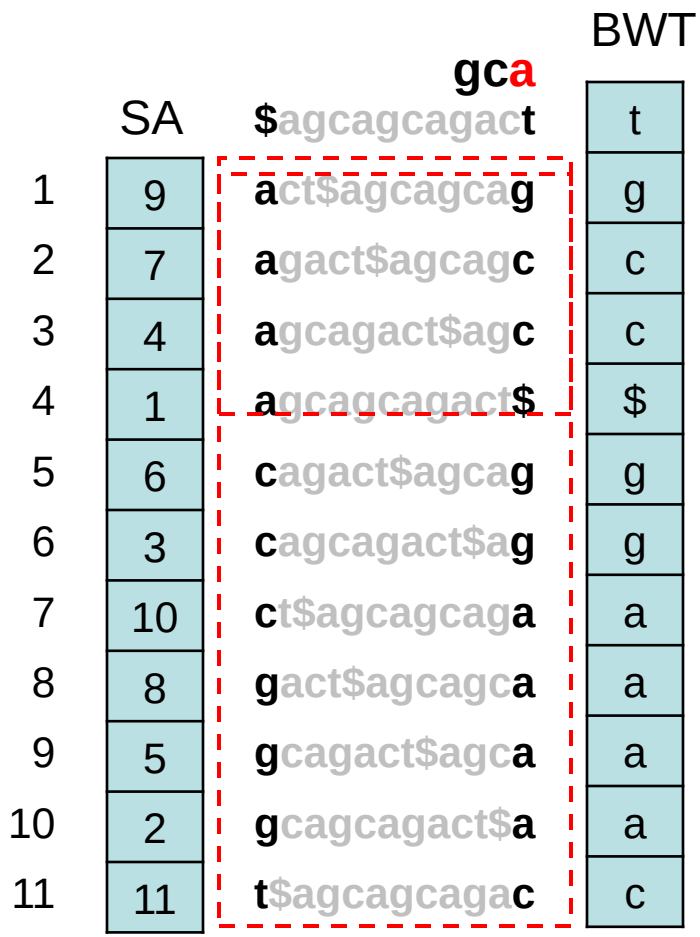
Checkpoints

# Auxiliary data structures

Key for efficient pattern matching: how to find the corresponding chars in the first column efficiently, in terms of both time and space.

TOTAL

| | a | c | g | t |
|---|---|---|---|---|
| rank | 1 | 5 | 8 | 11 |




OCC

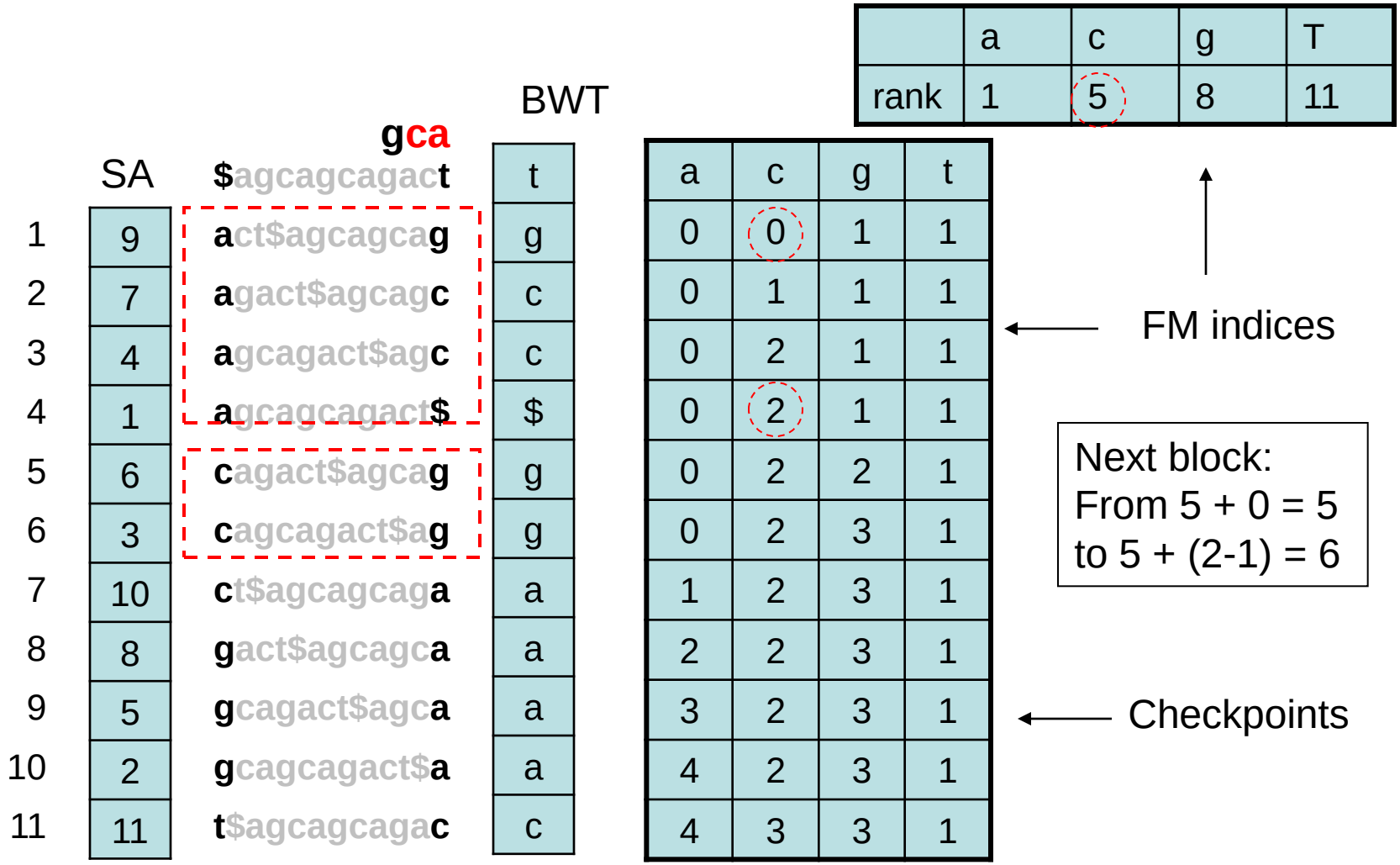
| a | c | g | t |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 |
| 0 | 2 | 1 | 1 |
| 0 | 2 | 1 | 1 |
| 0 | 2 | 2 | 1 |
| 0 | 2 | 3 | 1 |
| 1 | 2 | 3 | 1 |
| 2 | 2 | 3 | 1 |
| 3 | 2 | 3 | 1 |
| 4 | 2 | 3 | 1 |
| 4 | 3 | 3 | 1 |

FM indices

Next block:
From 1 + 0 = 1
to 1 + (4-1) = 4

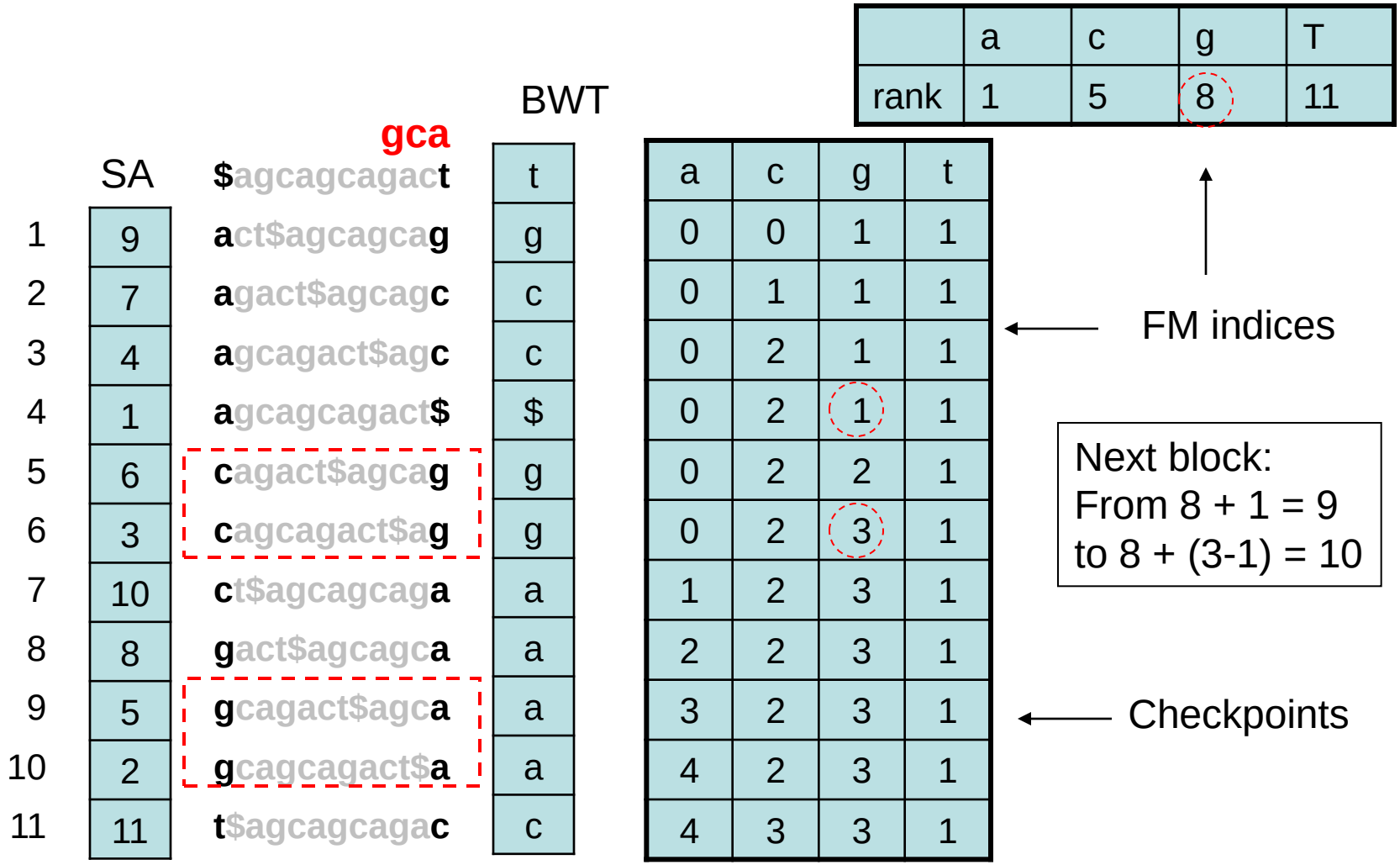
# Auxiliary data structures

Key for efficient pattern matching: how to find the corresponding chars in the first column efficiently, in terms of both time and space.



# Auxiliary data structures

Key for efficient pattern matching: how to find the corresponding chars in the first column efficiently, in terms of both time and space.



# FM Index: small memory footprint

Components of the FM Index:

| | |
|---|---|
| First column (*F*): | ~ $\lvert\Sigma\rvert$ integers |
| Last column (*L*): | *m* characters |
| SA sample: | $m \cdot a$ integers, where *a* is fraction of rows kept |
| Checkpoints: | $m \times \lvert\Sigma\rvert \cdot b$ integers, where *b* is fraction of rows checkpointed |

Example: DNA alphabet (2 bits per nucleotide), *T* = human genome, *a* = 1/32, *b* = 1/128

| | |
|---|---|
| First column (*F*): | 16 bytes |
| Last column (*L*): | 2 bits * 3 billion chars = 750 MB |
| SA sample: | 3 billion chars * 4 bytes/char / 32 = ~ 400 MB |
| Checkpoints: | *3 billion* * 4 bytes/char / *128* = ~ 100 MB |
| | Total < 1.5 GB |

End of Ch32