Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

http://algs4.cs.princeton.edu

5.3 SUBSTRING SEARCH

- *introduction*
- *brute force*
- *Knuth–Morris–Pratt*
- *Boyer–Moore*
- *Rabin–Karp*

Last updated on 4/25/16 1:22 PM

Substring search quiz 0

Do any of the algorithms we've studied so far have a running time that's a *decreasing* function of the input size?

A. Yes

B. No

C. Haha no way

D. *I don't know.*

http://algs4.cs.princeton.edu

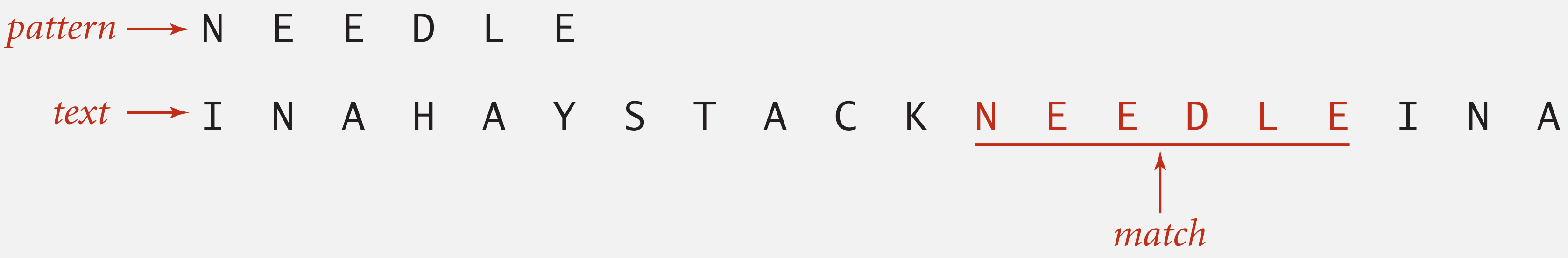
5.3 Substring Search

- *introduction*
- *brute force*
- *Knuth–Morris–Pratt*
- *Boyer–Moore*
- *Rabin–Karp*

Substring search

Goal. Find pattern of length M in a text of length N.

typically N >> M



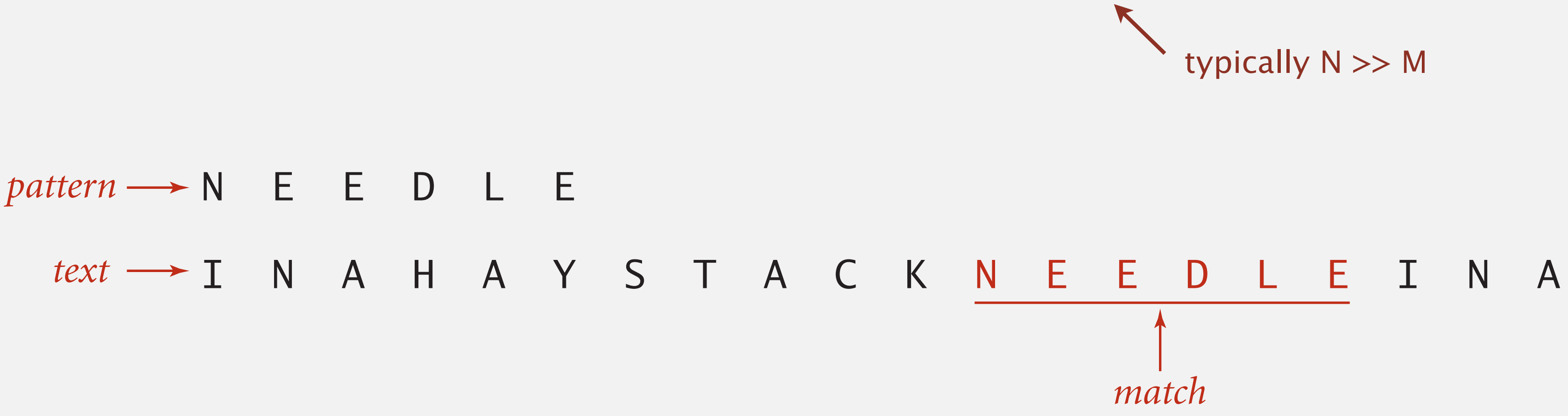
pattern → N E E D L E

text → I N A H A Y S T A C K N E E D L E I N A

match

Substring search applications

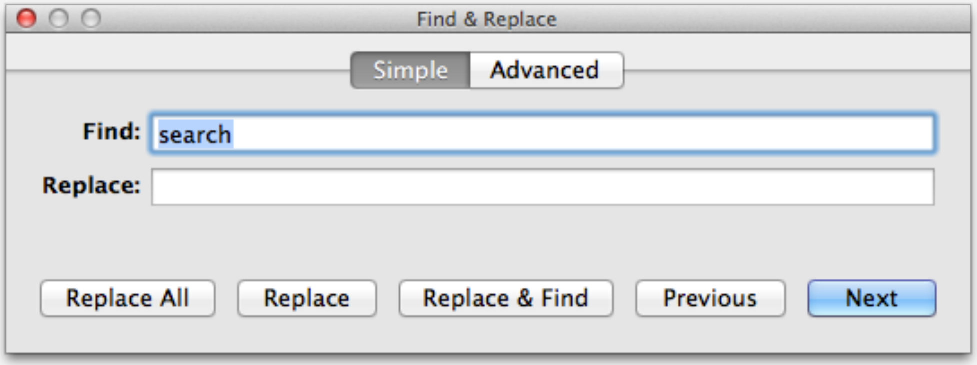
Goal. Find pattern of length M in a text of length N.


typically N >> M

pattern → N E E D L E

text → I N A H A Y S T A C K N E E D L E I N A

match



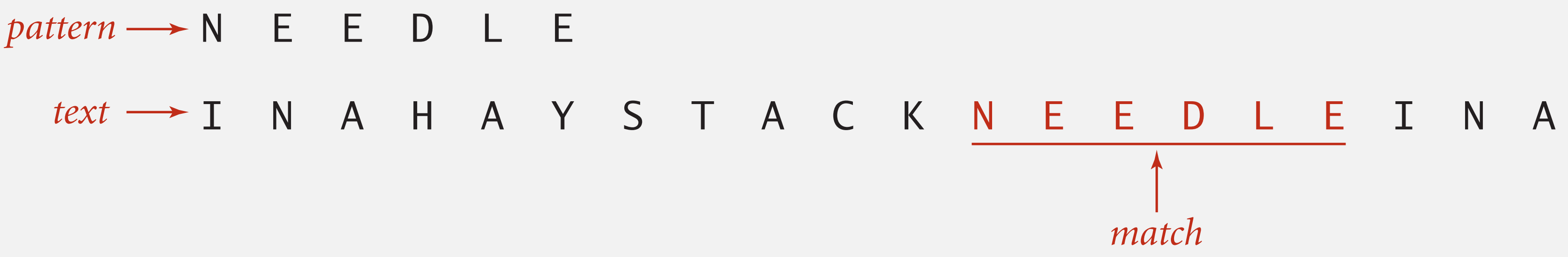
Find: search

Replace:

Substring search applications

Goal. Find pattern of length M in a text of length N.

typically N >> M



pattern → N E E D L E

text → I N A H A Y S T A C K N E E D L E I N A

match

Computer forensics. Search memory or disk for signatures, e.g., all URLs or RSA keys that the user has entered.

http://citp.princeton.edu/memory

Substring search applications

Electronic surveillance.

Need to monitor all internet traffic. (security)

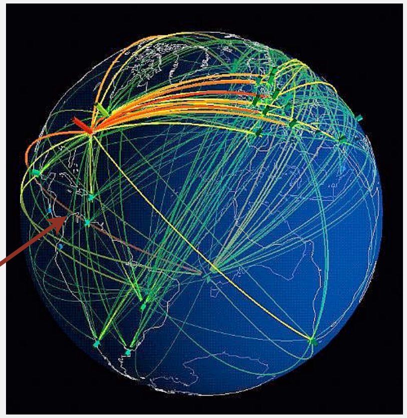
No way! (privacy)

Well, we're mainly interested in "ATTACK AT DAWN"

OK. Build a machine that just looks for that.

"ATTACK AT DAWN" substring search machine

found



Latest censored keywords in China

female infant + vaccine + die

Hebei + female infant + vaccine

Panama

Banama (Panama)

banama (Panama)

[Panama] Canal Papers

[Pa]nama Papers

launder money + brother-in-law

Xi + brother-in-law

top Chinese official + offshore

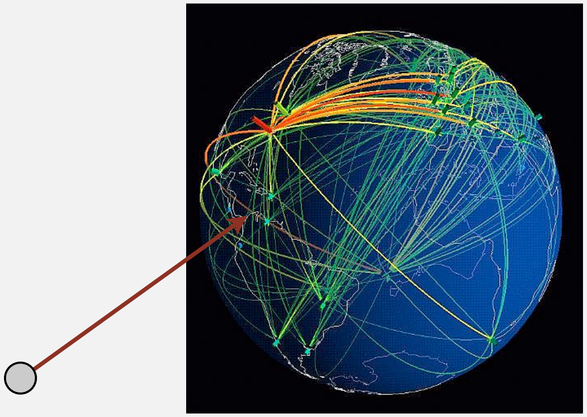
Wen [Jiabao] clan

Xi + explode

Wanda + bigwig

Leshi + [Jia] Yueting

50 cents + internet commentary



Drop packet if any keyword found

From http://chinadigitaltimes.net/2013/06/grass-mud-horse-list/

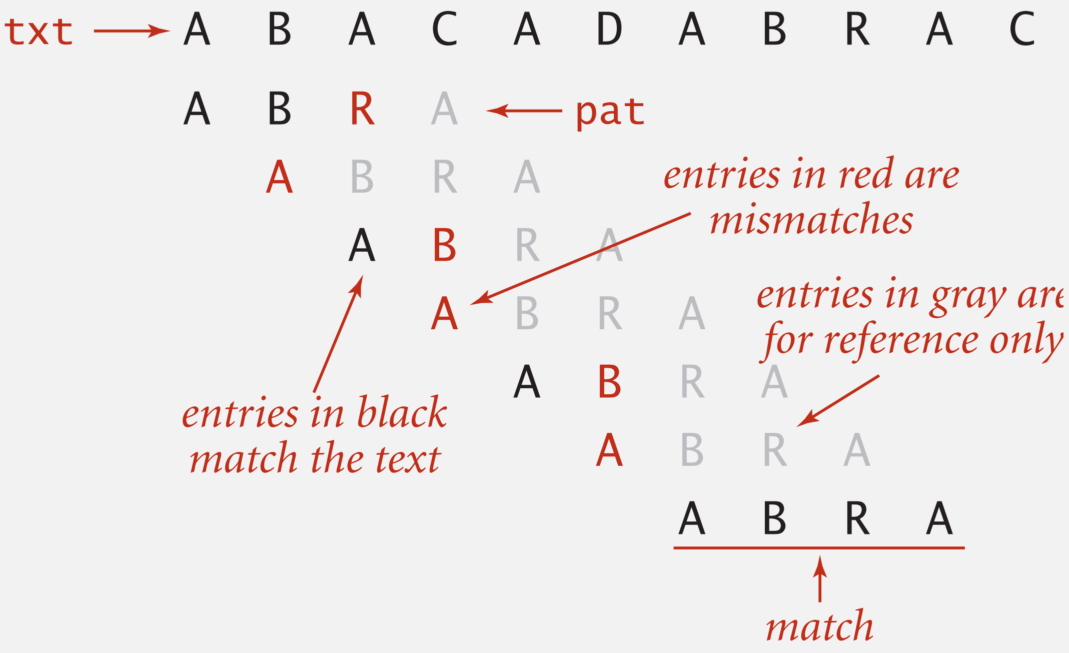
http://algs4.cs.princeton.edu

5.3 Substring Search

- *introduction*
- *brute force*
- *Knuth–Morris–Pratt*
- *Boyer–Moore*
- *Rabin–Karp*

Brute-force substring search

Check for pattern starting at each text position.



```
txt → A B A C A D A B R A C
      A B R A ← pat
        A B R A
          A B R A
            A B R A
              A B R A
                A B R A
                  A B R A
```

entries in red are mismatches

entries in gray are for reference only

entries in black match the text

match

Substring search quiz 1

Suppose you want to count the number of all occurrences of some pattern string of length M in a text of length N. What is the order of growth of the best-case and worst-case running time of the brute-force algorithm? Assume $M \leq N$.

A. N and $M N$

B. N and $M N^2$

C. MN and $M N$

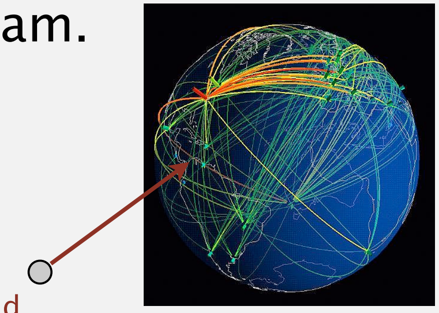
D. MN and $M N^2$

E. *I don't know.*

Backup

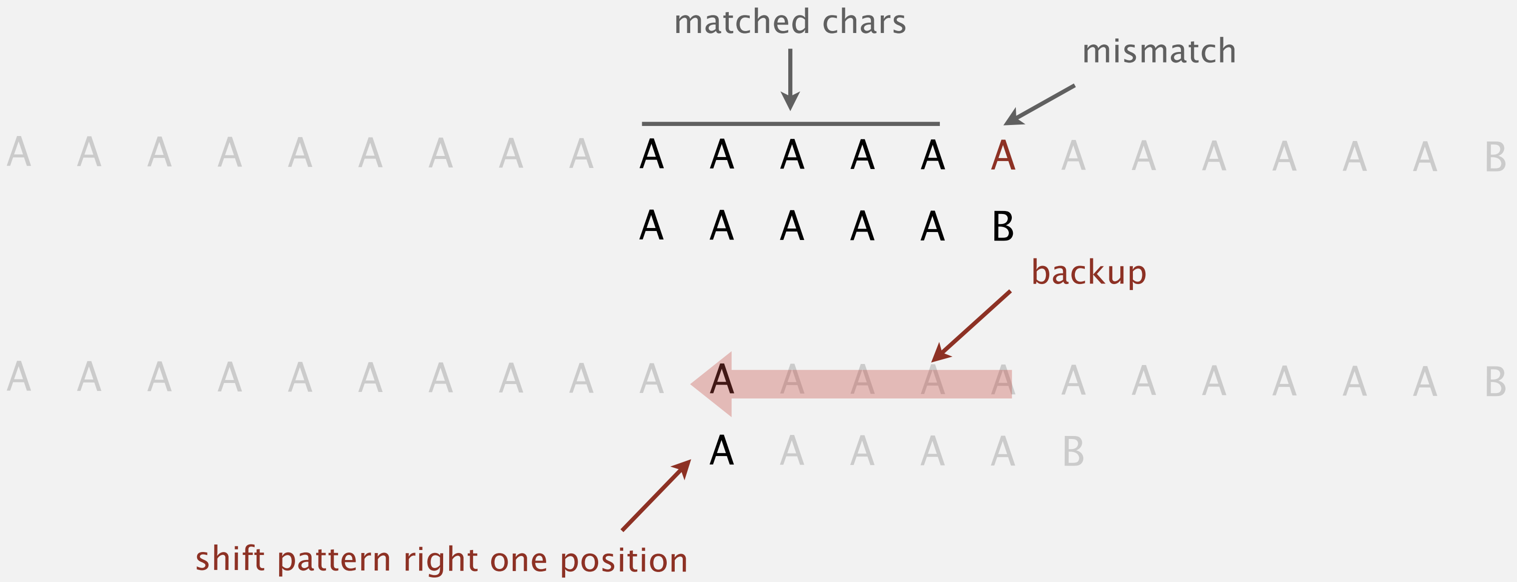
In many applications, we want to avoid backup in text stream.

- Treat input as stream of data.
- Abstract model: standard input.



"ATTACK AT DAWN"
substring search machine

Brute-force algorithm needs backup for every mismatch.


matched chars
mismatch
A A A A A A A A A A A A A A A A A A A A A B
A A A A A B
backup
A A A A A A A A A A A A A A A A A A A A A B
A A A A A B
shift pattern right one position

Approach 1. Maintain buffer of last M characters.

Approach 2. Streaming algorithm.

http://algs4.cs.princeton.edu

5.3 Substring Search

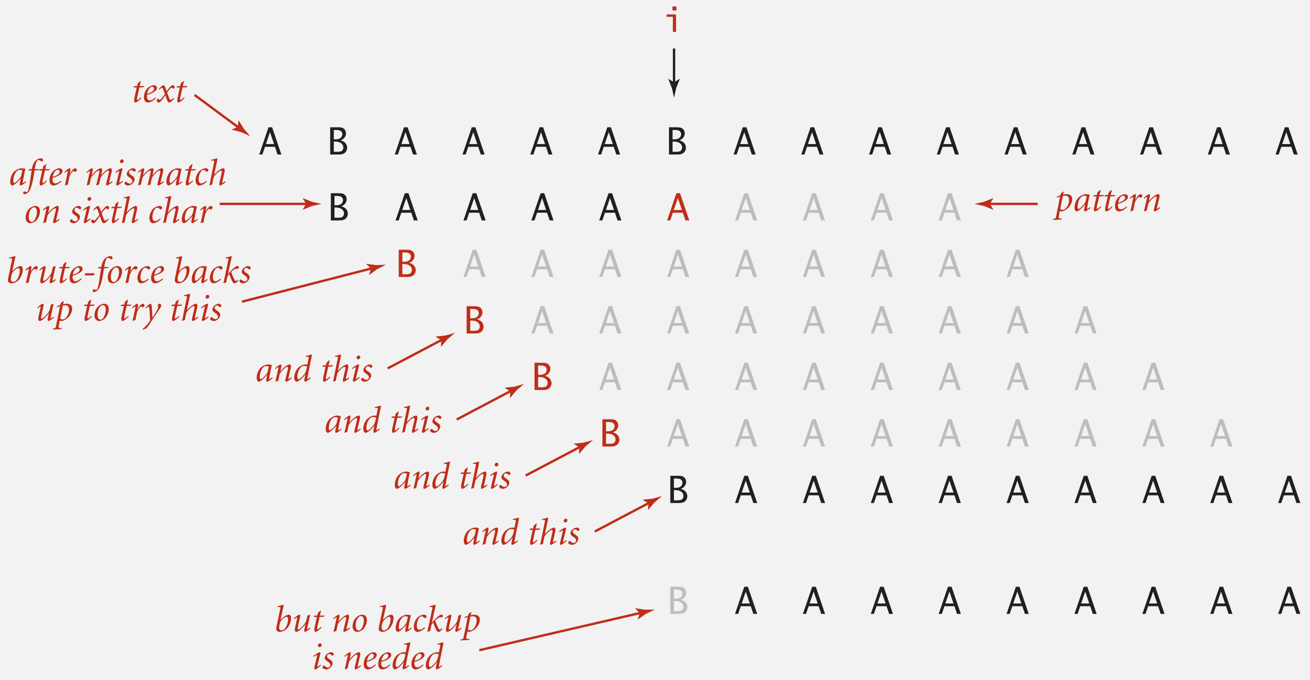
- introduction
- brute force
- *Knuth–Morris–Pratt*
- Boyer–Moore
- Rabin–Karp

Knuth–Morris–Pratt substring search

Intuition. Suppose we are searching in text for pattern BAAAAAAAAA.

- Suppose we match 5 chars in pattern, with mismatch on 6th char.
- We know previous 6 chars in text are BAAAAB.
- Don't need to back up text pointer!

assuming { A, B } alphabet



```
                                   i
                                   ↓
text                 A  B  A  A  A  A  B  A  A  A  A  A  A  A  A  A
after mismatch
on sixth char           B  A  A  A  A  A  A  A  A  A   pattern
brute-force backs
up to try this             B  A  A  A  A  A  A  A  A  A
and this                      B  A  A  A  A  A  A  A  A  A
and this                         B  A  A  A  A  A  A  A  A  A
and this                            B  A  A  A  A  A  A  A  A  A
and this                               B  A  A  A  A  A  A  A  A  A

but no backup
is needed                              B  A  A  A  A  A  A  A  A  A
```

Knuth–Morris–Pratt algorithm. Clever method to always avoid backup!

Searching for BAAAAAAAAA

Let j = #characters matched so far.

When $j = 0$:

- If we see 'A': j remains 0
- If we see 'B': j becomes 1

When 1 <= j < 10:

- If we see 'A': $j = j + 1$
- If we see 'B': $j = 1$

$j = 10$: match!

Properties of state transition matrix:

- Depends only on pattern, not text
- #rows = alphabet size
- #columns = length of pattern
- In each col, exactly one row lets us increment the state j

	B	A	A	A	A	A	A	A	A	A
j	0	1	2	3	4	5	6	7	8	9
A	0	2	3	4	5	6	7	8	9	10
B	1	1	1	1	1	1	1	1	1	1

Exercise

Construct the state transition matrix for the pattern ABABAC.

e.g., $j = 3$: we've matched the string 'ABA' in the text (XXXXXX**ABA**)

- If we see 'A': we've matched 'A' (XXXXXXABA**A**) ⇒ j becomes 1
- If we see 'B': we've matched 'ABAB' (XXXXXX**ABAB**) ⇒ j becomes 4
- If we see 'C': we've matched nothing (XXXXXXABAC) ⇒ j becomes 0

	A	B	A	B	A	C
j	0	1	2	3	4	5
A				1		
B				4		
C				0		

old:

	B	A	A	A	A	A	A	A	A	A
j	0	1	2	3	4	5	6	7	8	9
A	0	2	3	4	5	6	7	8	9	10
B	1	1	1	1	1	1	1	1	1	1

Exercise

Construct the state transition matrix for the pattern ABABAC.

e.g., $j = 3$: we've matched the string 'ABA' in the text (XXXXXX**ABA**)

- If we see 'A': we've matched 'A' (XXXXXXABA**A**) ⇒ j becomes 1
- If we see 'B': we've matched 'ABAB' (XXXXXX**ABAB**) ⇒ j becomes 4
- If we see 'C': we've matched nothing (XXXXXXABAC) ⇒ j becomes 0

	A	B	A	B	A	C
j	0	1	2	3	4	5
A	1	1	3	1	5	1
B	0	2	0	4	0	4
C	0	0	0	0	0	6

old:

	B	A	A	A	A	A	A	A	A	A
j	0	1	2	3	4	5	6	7	8	9
A	0	2	3	4	5	6	7	8	9	10
B	1	1	1	1	1	1	1	1	1	1

Deterministic finite state automaton (DFA)

DFA is abstract string-searching machine.

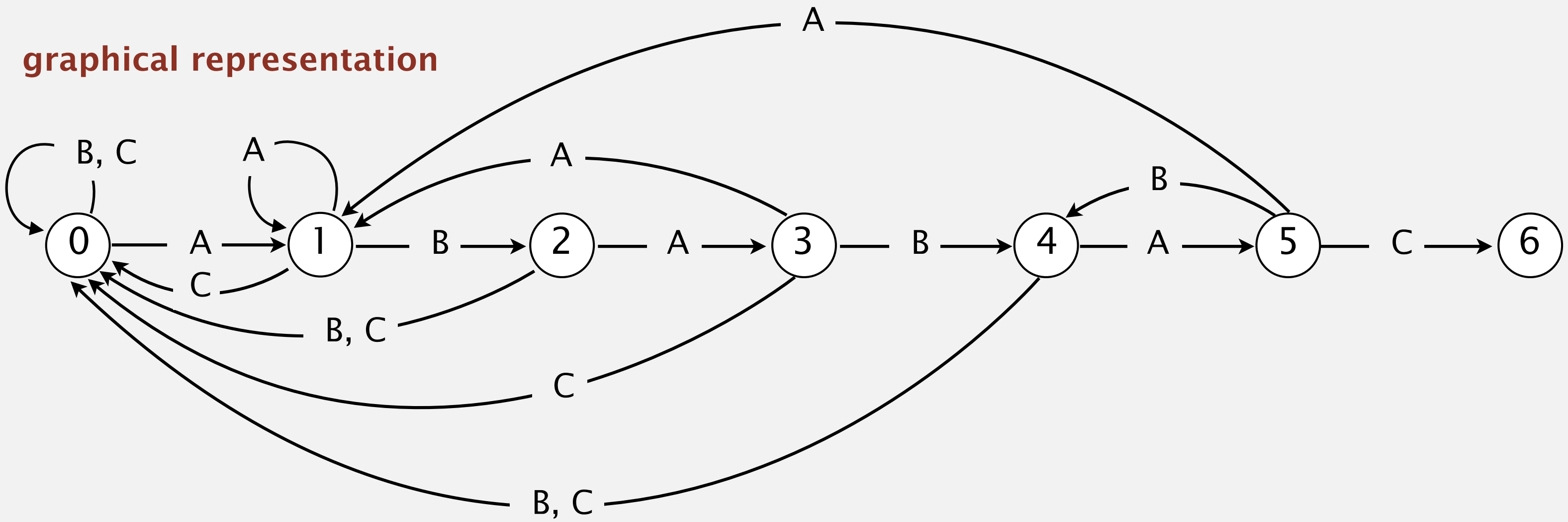
- Finite number of states (including start and halt).
- Exactly one state transition for each char in alphabet.
- Accept if sequence of state transitions leads to halt state.

internal representation

j		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6

If in state j reading char c:
 if j is 6 halt and accept
 else move to state dfa[c][j]

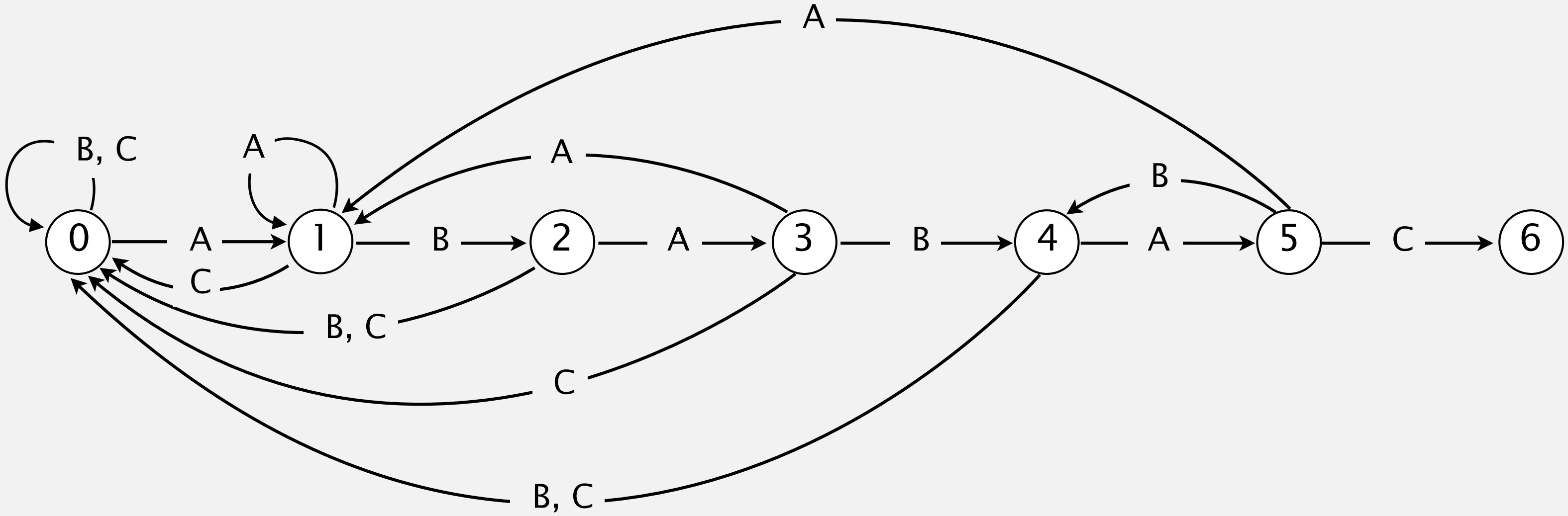
graphical representation



Knuth–Morris–Pratt demo: DFA simulation

A A B A C A A B A B A C A A

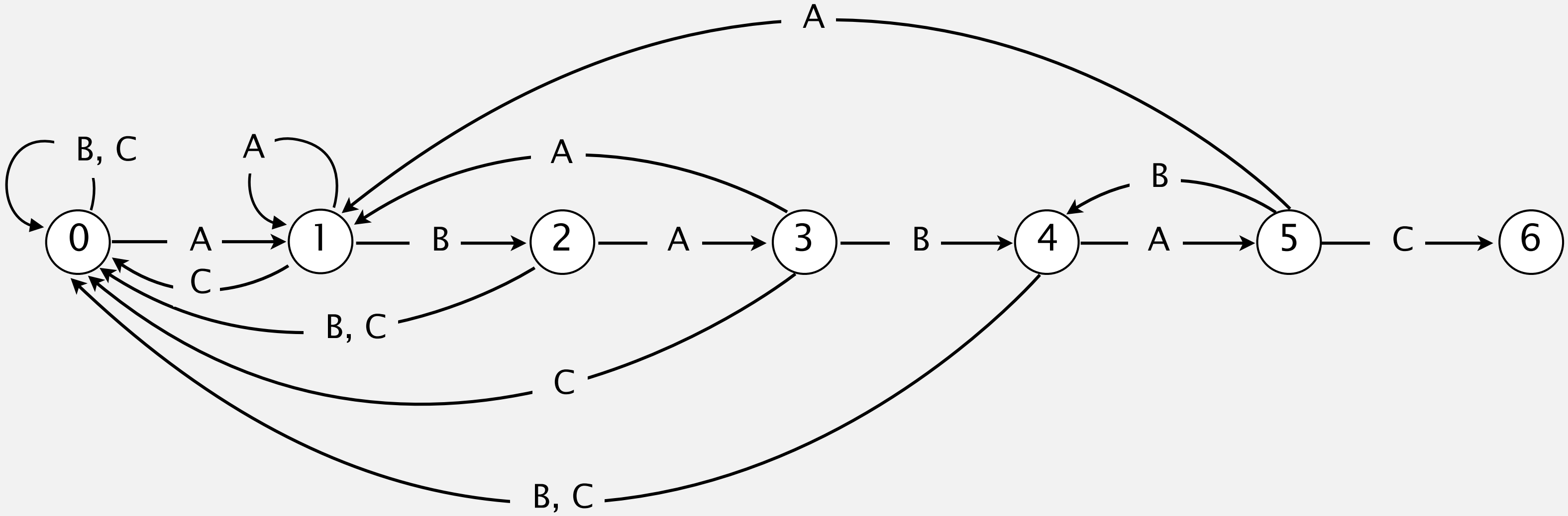
		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6



Knuth–Morris–Pratt demo: DFA simulation

A A B A C A A B A B A C A A

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6

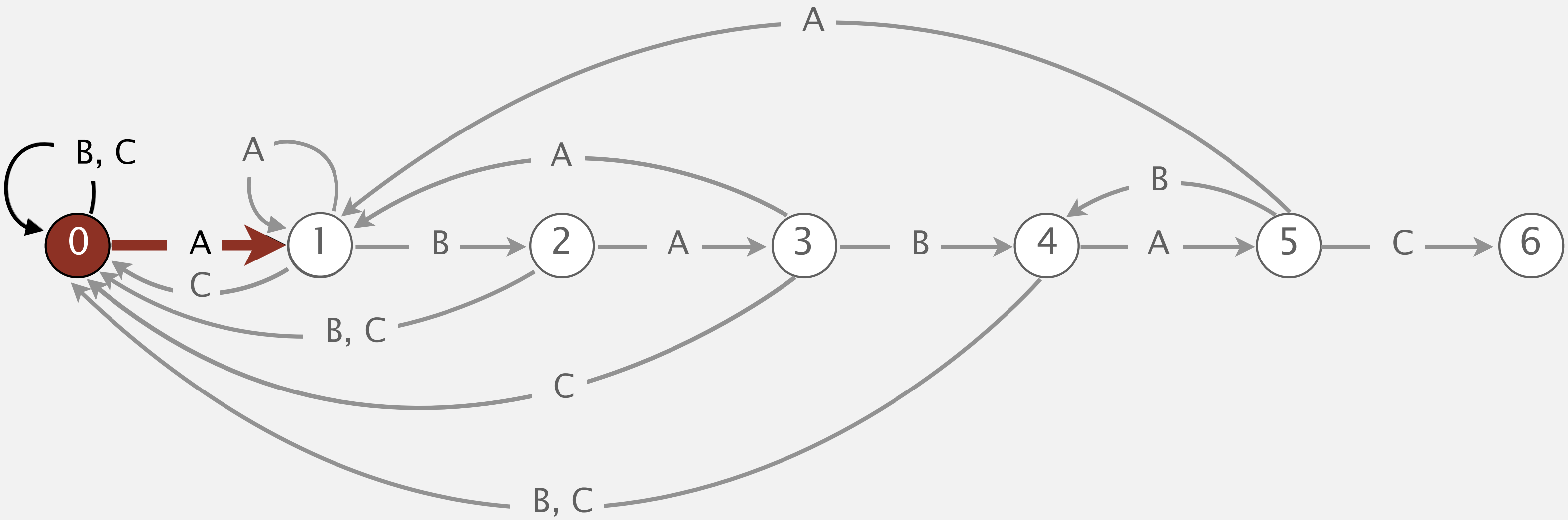


Knuth-Morris-Pratt demo: DFA simulation

A A B A C A A B A B A C A A

↑

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
	A	1	1	3	1	5	1
dfa[][j]	B	0	2	0	4	0	4
	C	0	0	0	0	0	6

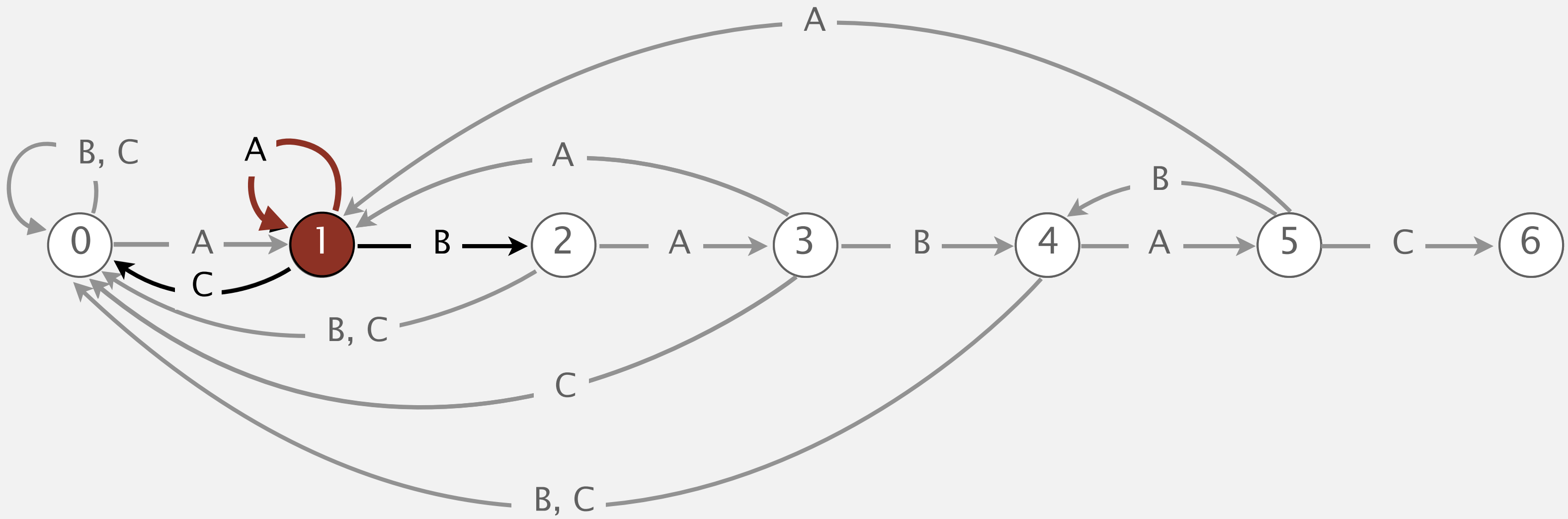


Knuth–Morris–Pratt demo: DFA simulation

A A B A C A A B A B A C A A

↑

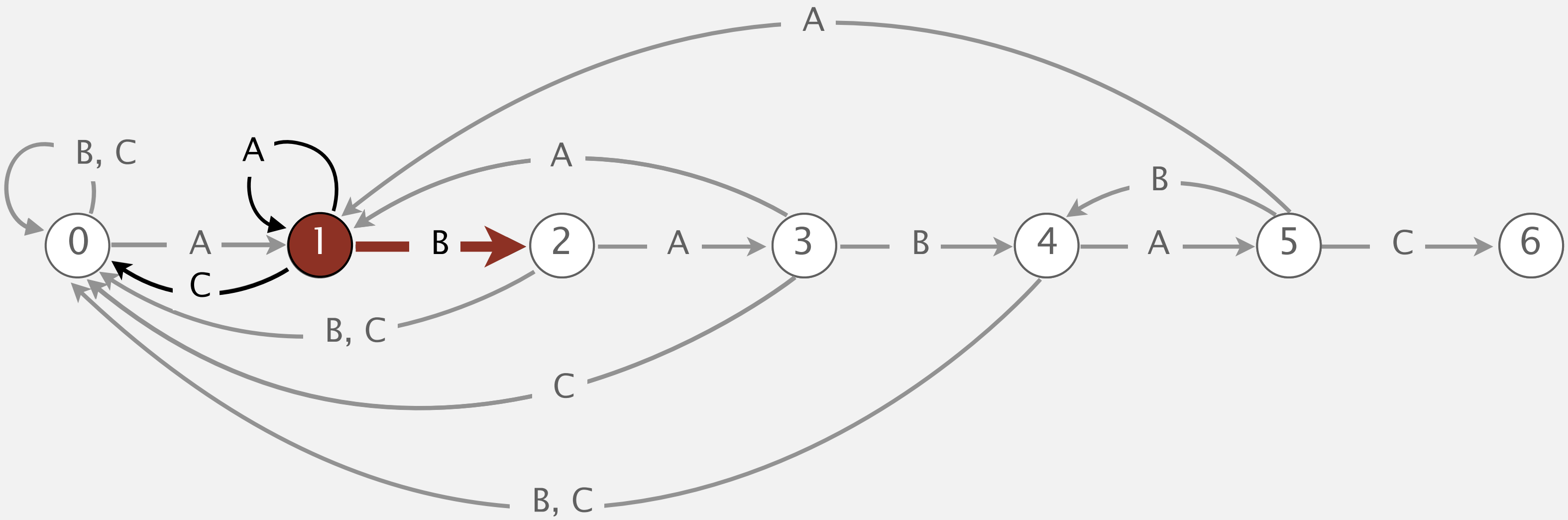
		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
	A	1	1	3	1	5	1
dfa[][j]	B	0	2	0	4	0	4
	C	0	0	0	0	0	6



Knuth-Morris-Pratt demo: DFA simulation

A A B A C A A B A B A C A A

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6

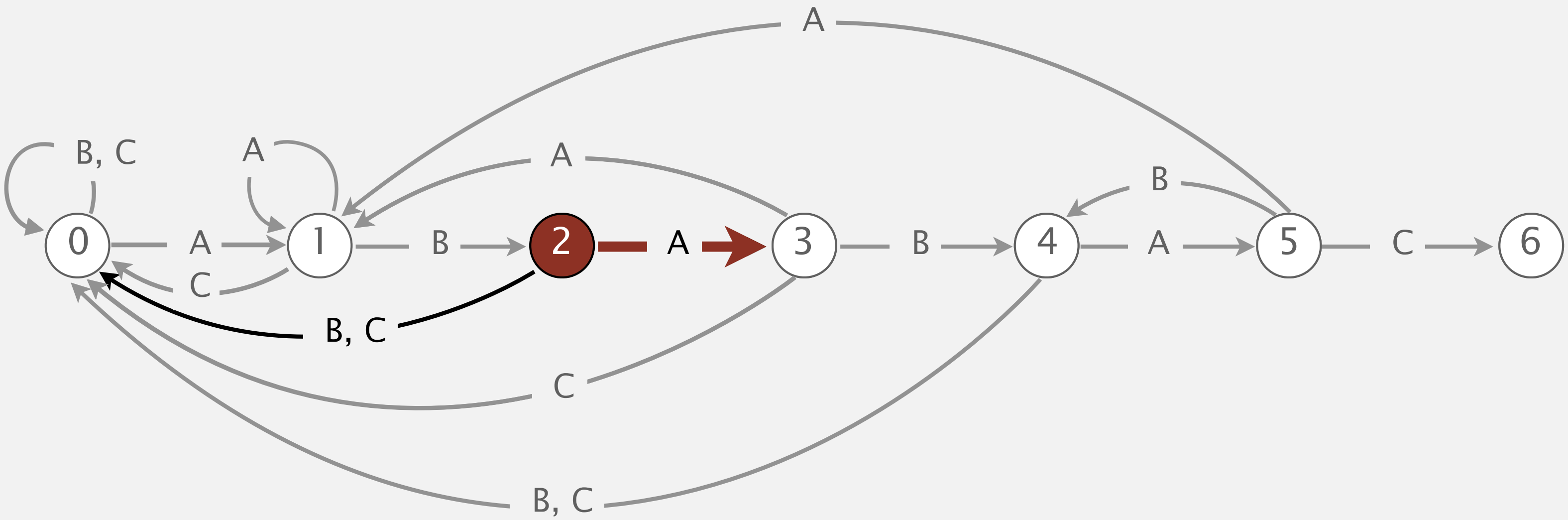


Knuth–Morris–Pratt demo: DFA simulation

A A B A C A A B A B A C A A

↑

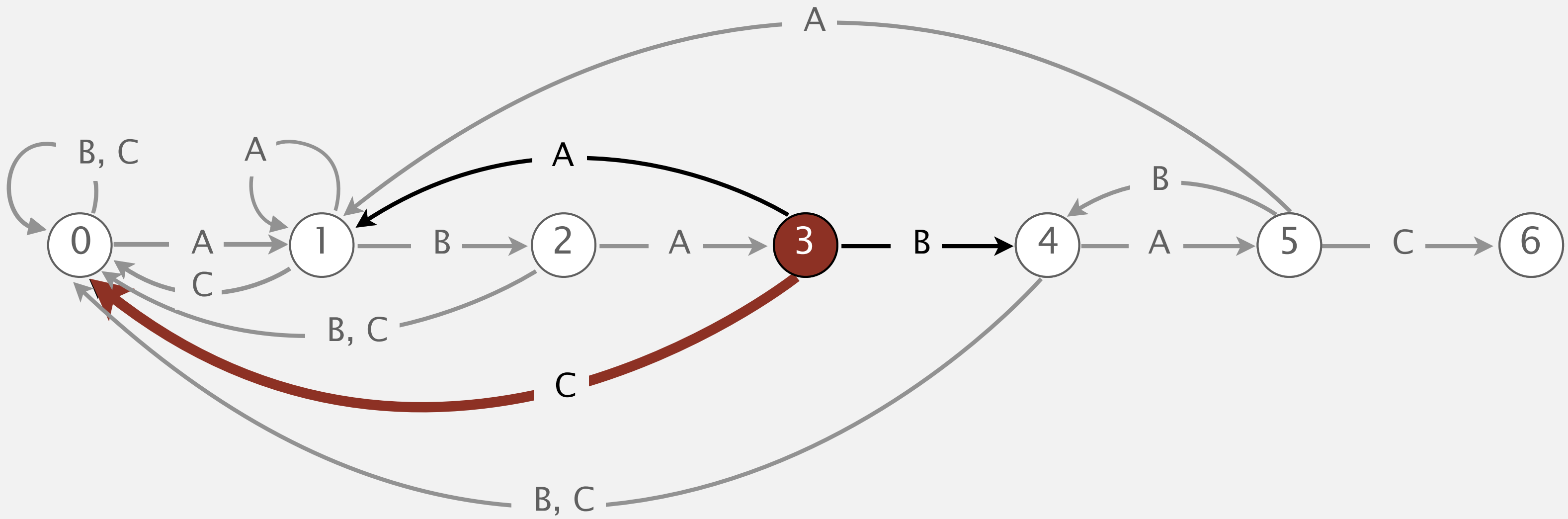
		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6



Knuth-Morris-Pratt demo: DFA simulation

A A B A C A A B A B A C A A

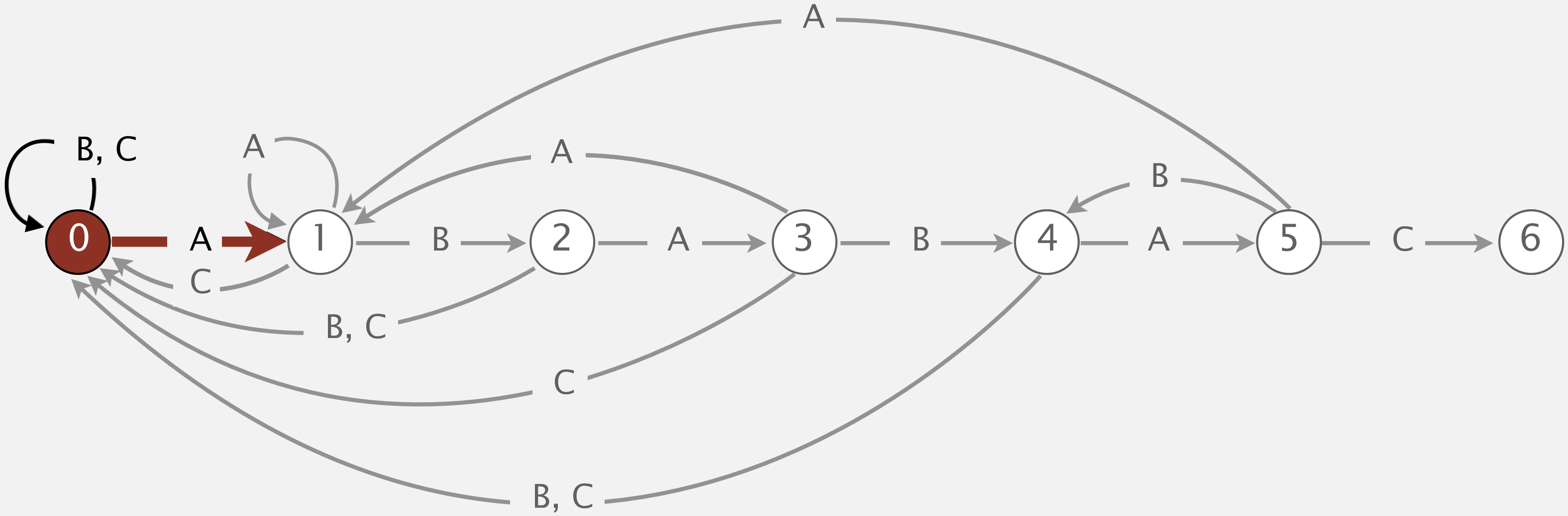
		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
	A	1	1	3	1	5	1
dfa[][j]	B	0	2	0	4	0	4
	C	0	0	0	0	0	6



Knuth–Morris–Pratt demo: DFA simulation

A A B A C A A B A B A C A A

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6

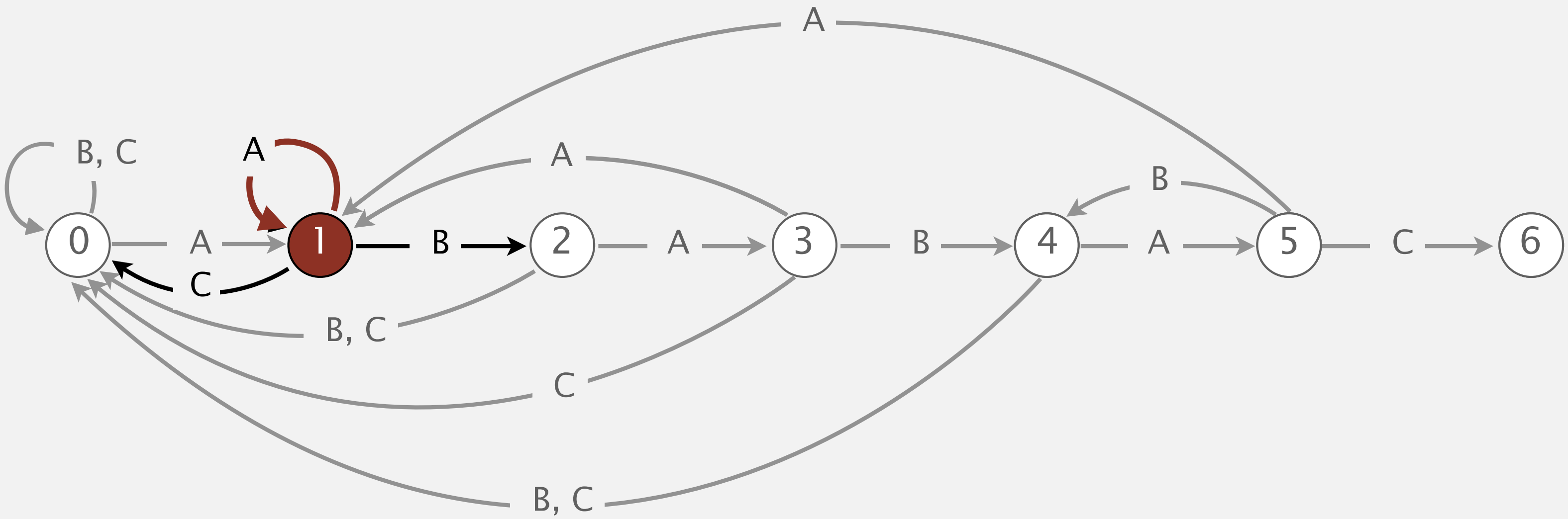


Knuth–Morris–Pratt demo: DFA simulation

A A B A C A A B A B A C A A

↑

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6

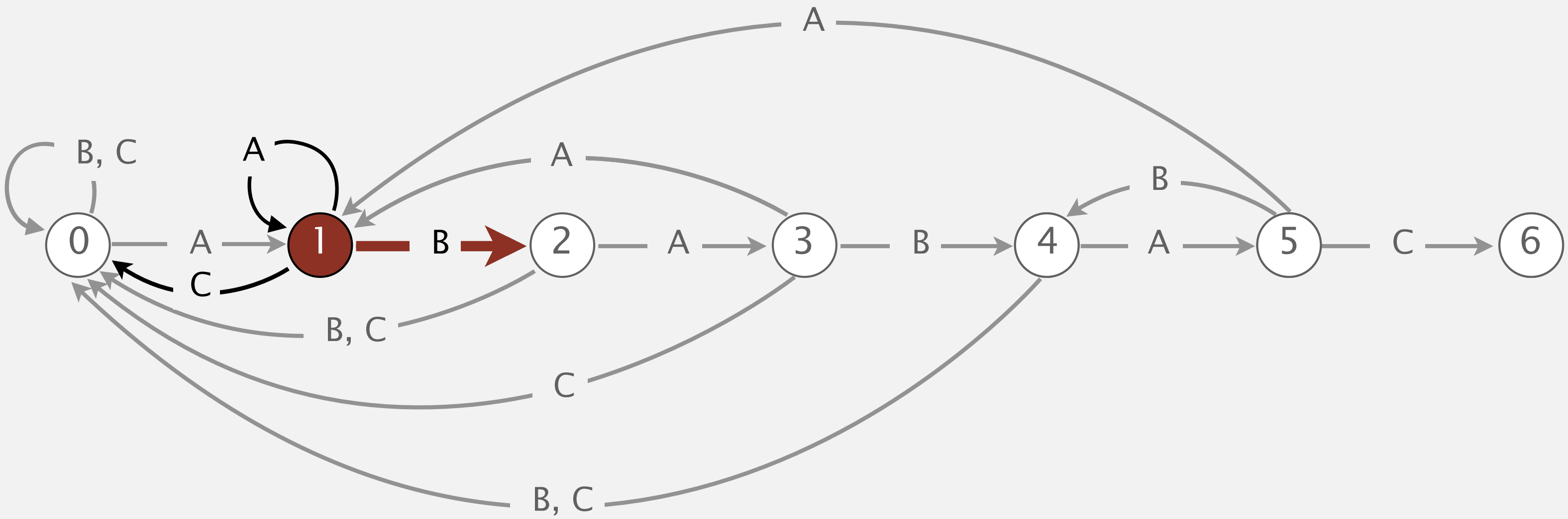


Knuth–Morris–Pratt demo: DFA simulation

A A B A C A A B A B A C A A

↑

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
	A	1	1	3	1	5	1
dfa[][j]	B	0	2	0	4	0	4
	C	0	0	0	0	0	6

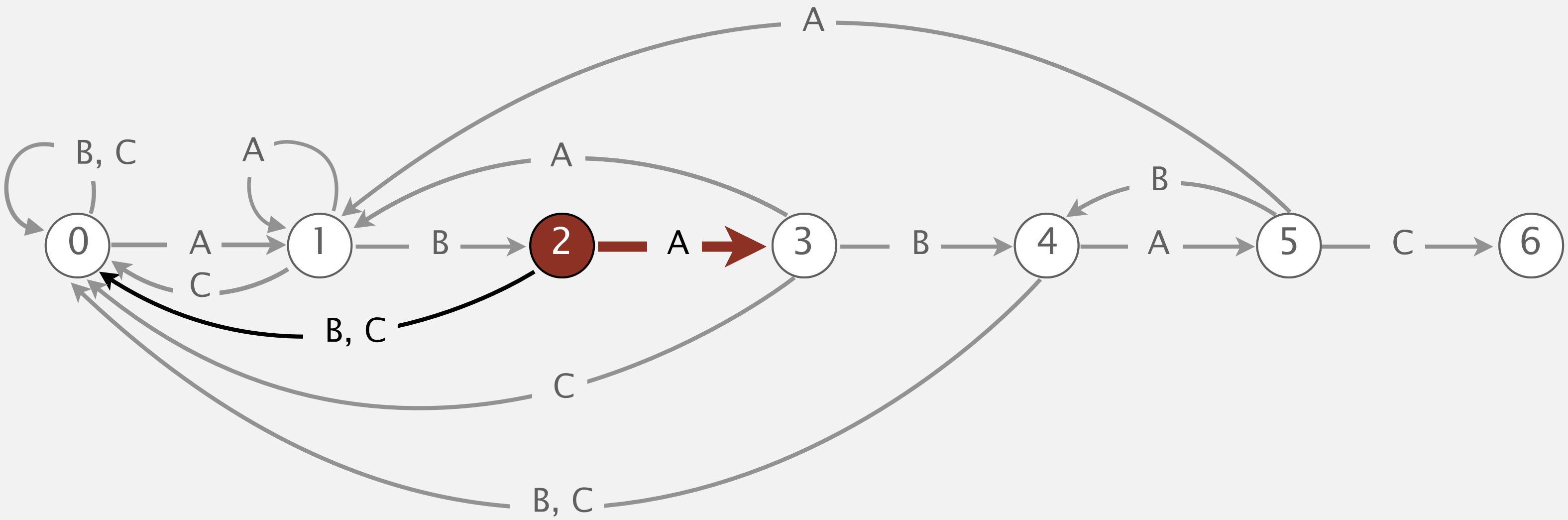


Knuth-Morris-Pratt demo: DFA simulation

A A B A C A A B A B A C A A

↑

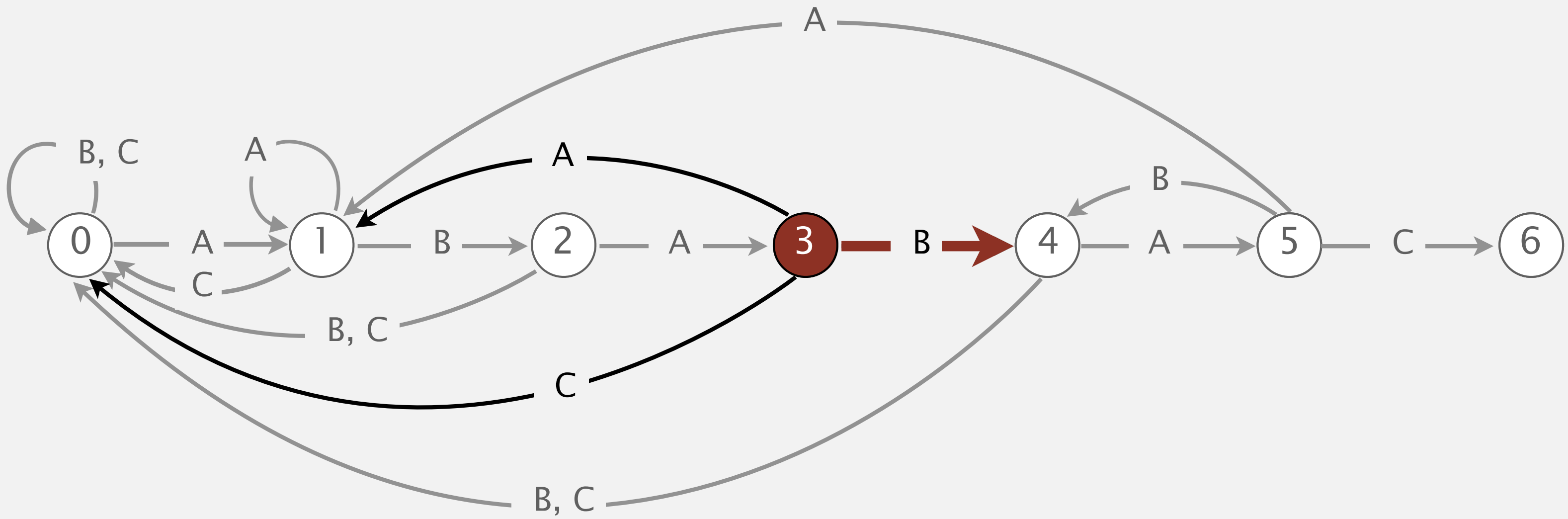
		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
	A	1	1	3	1	5	1
dfa[][j]	B	0	2	0	4	0	4
	C	0	0	0	0	0	6



Knuth–Morris–Pratt demo: DFA simulation

A A B A C A A B A B A C A A

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
	A	1	1	3	1	5	1
dfa[][j]	B	0	2	0	4	0	4
	C	0	0	0	0	0	6

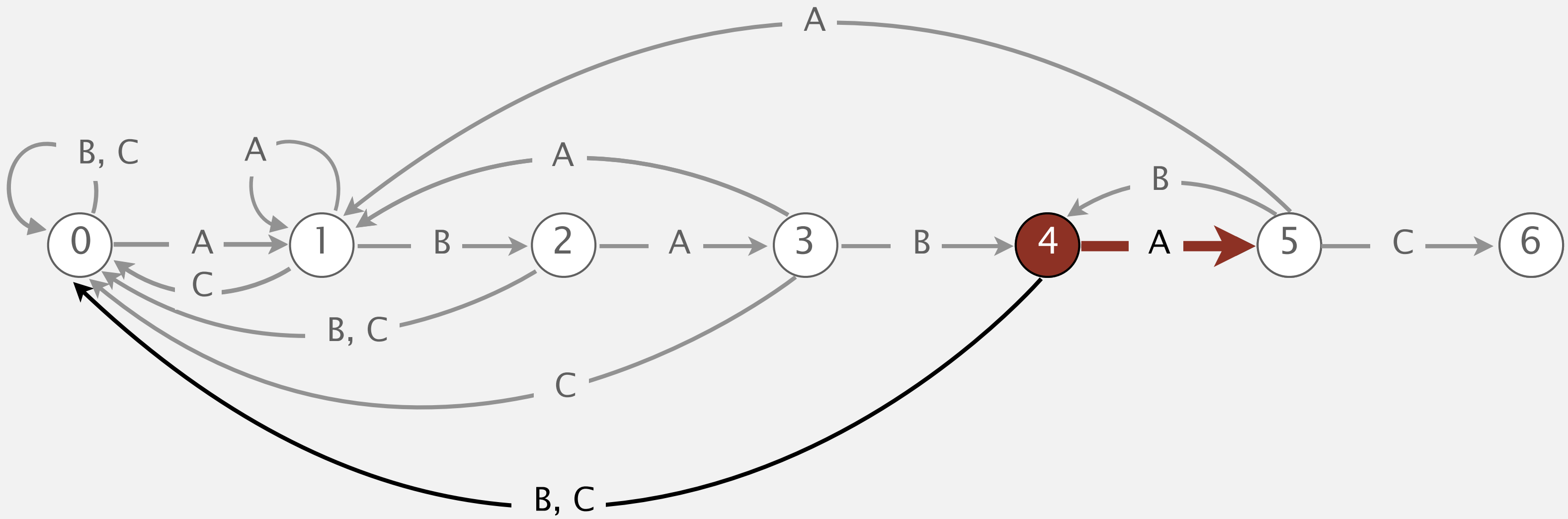


Knuth–Morris–Pratt demo: DFA simulation

A A B A C A A B A B A C A A

↑

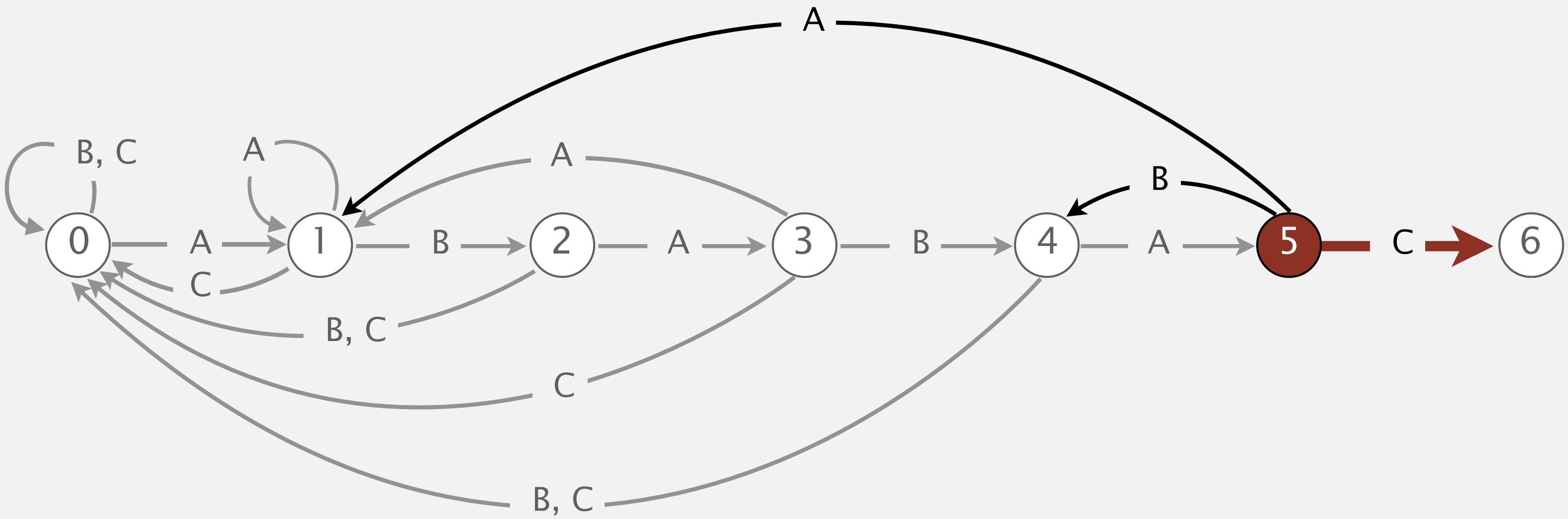
		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6



Knuth–Morris–Pratt demo: DFA simulation

AABACAABABACAA

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6

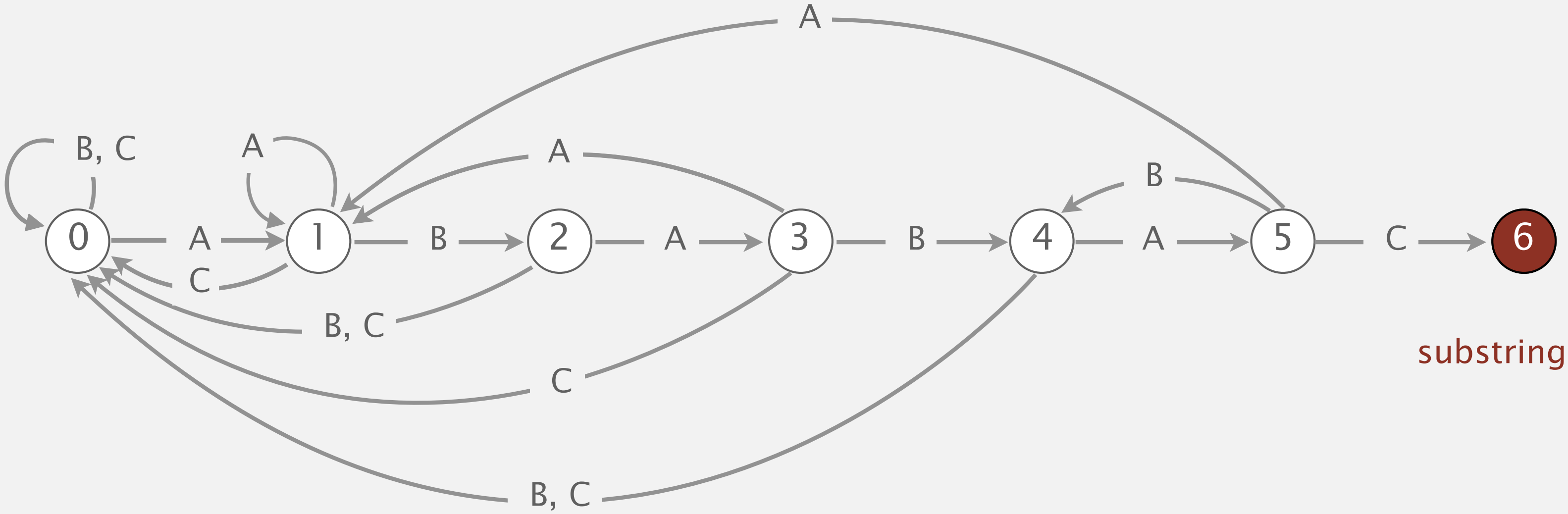


Knuth–Morris–Pratt demo: DFA simulation

A A B A C A A B A B A C A A

↑

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6



substring found

Interpretation of Knuth-Morris-Pratt DFA

Q. What is interpretation of DFA state after reading in `txt[i]`?

A. State = number of characters in pattern that have been matched.

length of longest prefix of `pat[]` that is a suffix of `txt[0..i]`

Ex. DFA is in state 3 after reading in `txt[0..6]`.

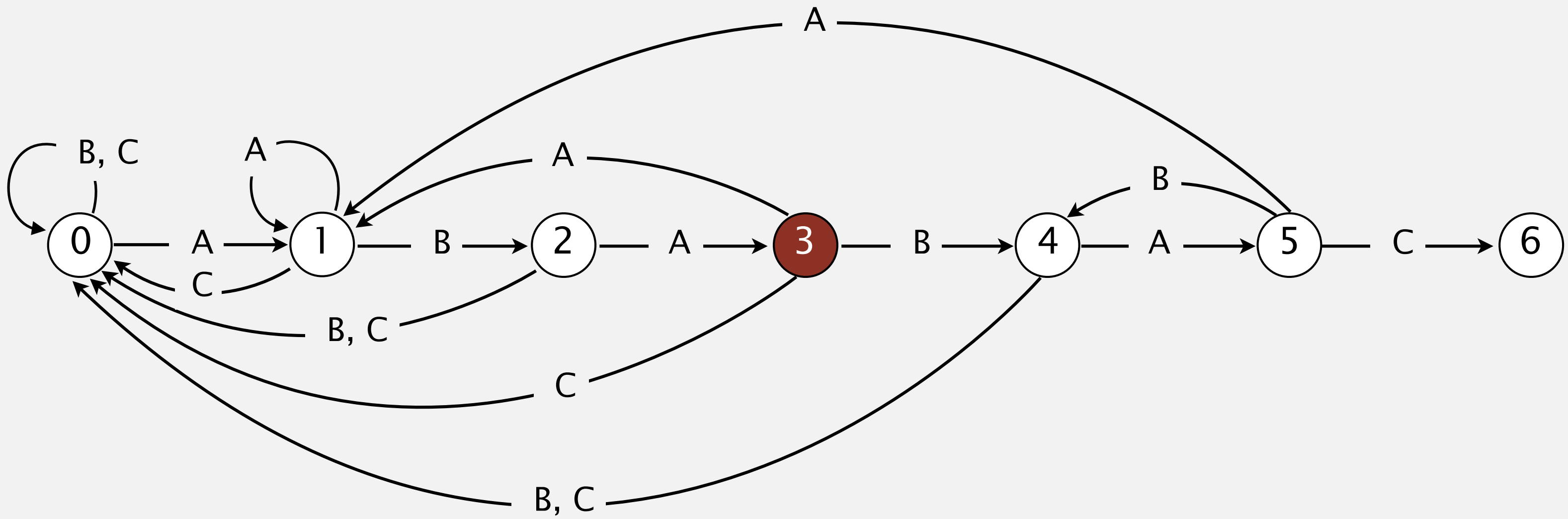
i

	0	1	2	3	4	5	6	7	8
txt →	B	C	B	A	A	B	A	C	A

suffix of `txt[0..6]`

	0	1	2	3	4	5
pat →	A	B	A	B	A	C

prefix of `pat[]`



Substring search quiz 2

Which state is the DFA in after processing the following input?

A A B B A B A B C A B A A B A A C A A A B A B A B A A C A A B A A B A B A B

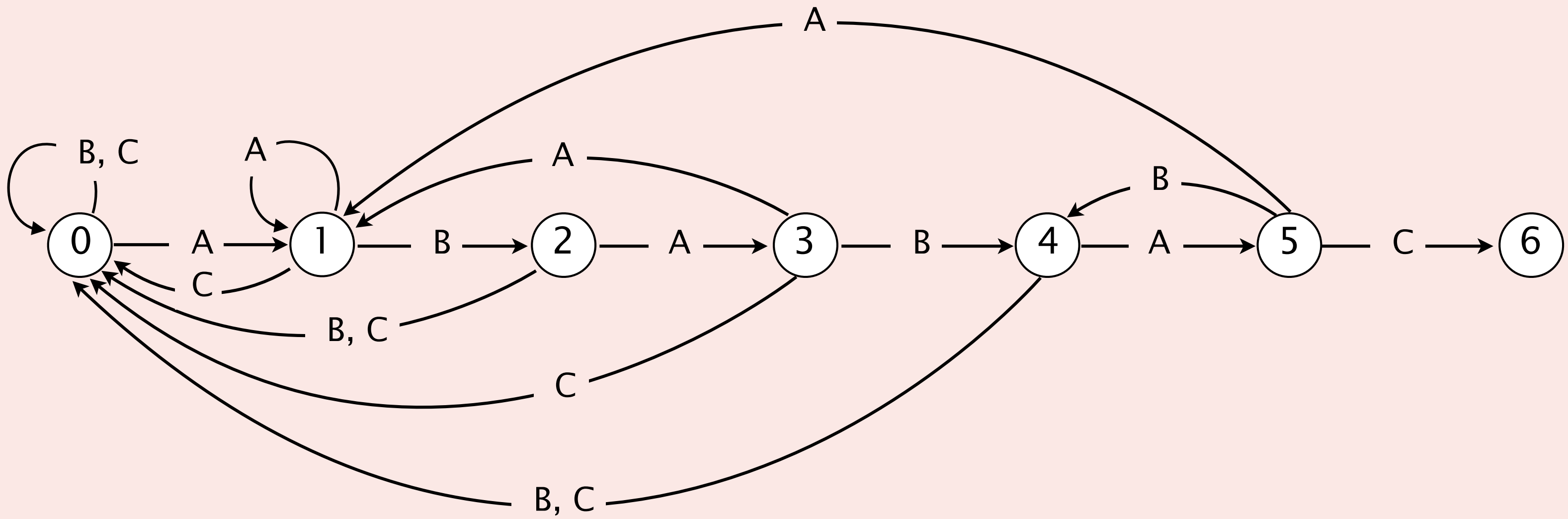
A. 0

B. 1

C. 3

D. 4

E. *I don't know.*



Knuth–Morris–Pratt substring search: Java implementation

Key differences from brute-force implementation.

- Need to precompute `dfa[][]` from pattern.
- Text pointer `i` never decrements.

```
public int search(String txt)
{
   int i, j, N = txt.length();
   for (i = 0, j = 0; i < N && j < M; i++)
      j = dfa[txt.charAt(i)][j];
   if (j == M) return i - M;
   else        return N;
}
```

no backup

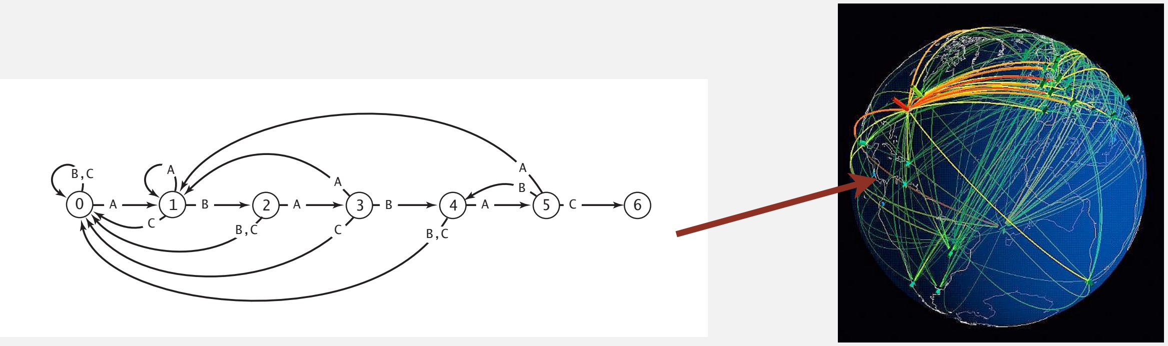
Knuth-Morris-Pratt substring search: Java implementation

Key differences from brute-force implementation.

- Need to precompute `dfa[][]` from pattern.
- Text pointer `i` never decrements.
- Could use input stream.

```
public int search(In in)
{
   int i, j;
   for (i = 0, j = 0; !in.isEmpty() && j < M; i++)
      j = dfa[in.readChar()][j];
   if (j == M) return i - M;
   else        return NOT_FOUND;
}
```

no backup

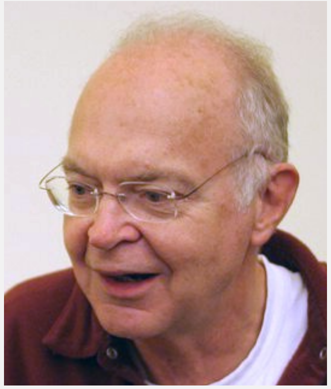


Knuth-Morris-Pratt Running time

Running time.

- Simulate DFA on text: at most N character accesses.
- Build DFA: how to do efficiently? See textbook/video.
  - In the vast majority of applications, the running time of building the DFA is irrelevant. [Arvind's opinion.]

> " *Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered.*
>
> *We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.*



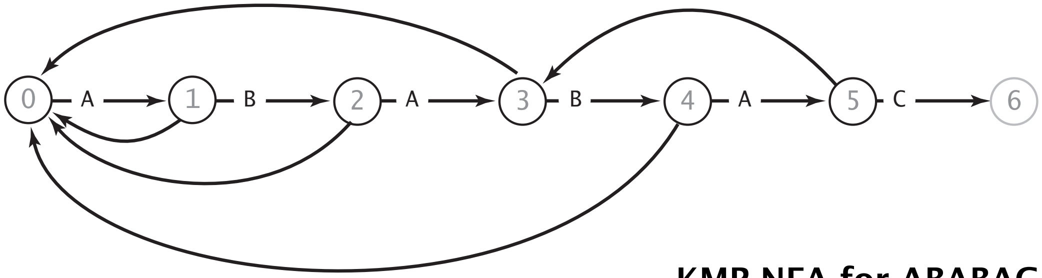
KMP substring search analysis

Proposition. KMP substring search accesses no more than $M + N$ chars to search for a pattern of length M in a text of length N.

Pf. Each pattern character accessed once when constructing the DFA; each text character accessed once (in the worst case) when simulating the DFA.

Proposition. KMP constructs `dfa[][]` in time and space proportional to $R\,M$.

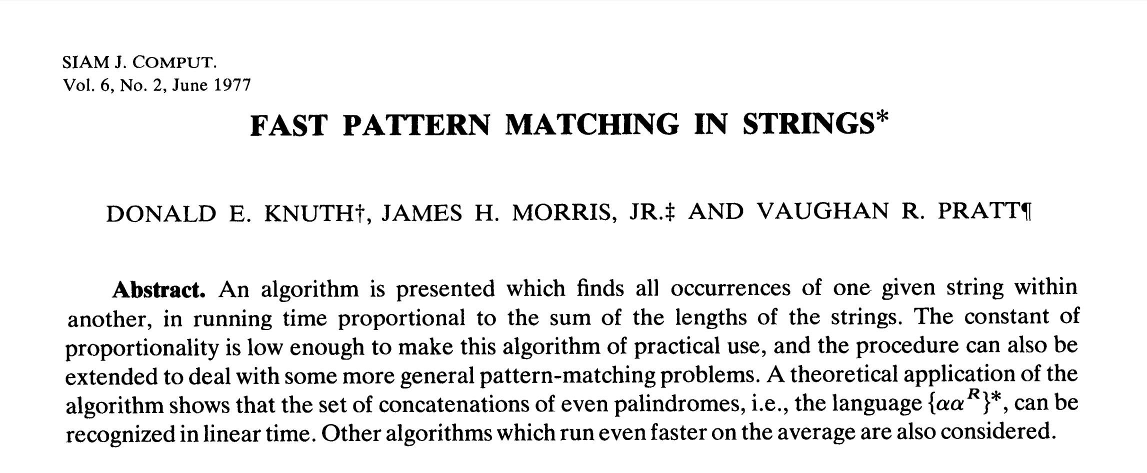
Larger alphabets. Improved version of KMP constructs `nfa[]` in time and space proportional to M.



KMP NFA for ABABAC

Knuth–Morris–Pratt: brief history

- Independently discovered by two theoreticians and a hacker.
  - Knuth: inspired by esoteric theorem, discovered linear algorithm
  - Pratt: made running time independent of alphabet size
  - Morris: built a text editor for the CDC 6400 computer
- Theory meets practice.



SIAM J. COMPUT.
Vol. 6, No. 2, June 1977

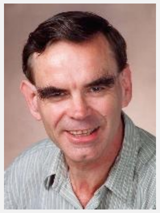
FAST PATTERN MATCHING IN STRINGS*

DONALD E. KNUTH†, JAMES H. MORRIS, JR.‡ AND VAUGHAN R. PRATT¶

Abstract. An algorithm is presented which finds all occurrences of one given string within another, in running time proportional to the sum of the lengths of the strings. The constant of proportionality is low enough to make this algorithm of practical use, and the procedure can also be extended to deal with some more general pattern-matching problems. A theoretical application of the algorithm shows that the set of concatenations of even palindromes, i.e., the language $\{\alpha\alpha^R\}^*$, can be recognized in linear time. Other algorithms which run even faster on the average are also considered.

Don Knuth

Jim Morris



Vaughan Pratt

Cyclic Rotation

A string s is a cyclic rotation of t if s and t have the same length and s is a suffix of t followed by a prefix of t.

yes	yes	no
ROTATEDSTRING	ABABABBABBABA	ROTATEDSTRING
STRINGROTATED	BABBABBABAABA	GNIRTSDETATOR

Problem. Given two binary strings s and t, design a linear-time algorithm to determine if s is a cyclic rotation of t.

Algorithms

Robert Sedgewick | Kevin Wayne

http://algs4.cs.princeton.edu

5.3 Substring Search

- *introduction*
- *brute force*
- *Knuth–Morris–Pratt*
- *Boyer–Moore*
- *Rabin–Karp*

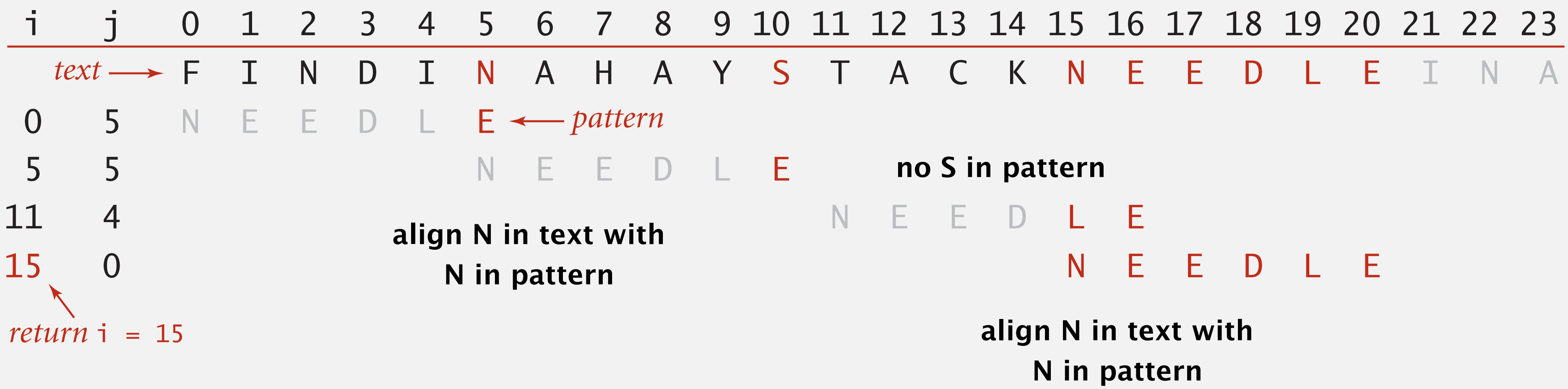
Robert Boyer

J. Strother Moore

Boyer-Moore: mismatched character heuristic

Intuition.

- Scan characters in pattern from right to left.
- Can skip as many as M text chars when finding one not in the pattern.



i	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
	text →	F	I	N	D	I	N	A	H	A	Y	S	T	A	C	K	N	E	E	D	L	E	I	N	A
0	5	N	E	E	D	L	E ← *pattern*																		
5	5						N	E	E	D	L	E													
11	4												N	E	E	D	L	E							
15	0																N	E	E	D	L	E			

return i = 15

no S in pattern

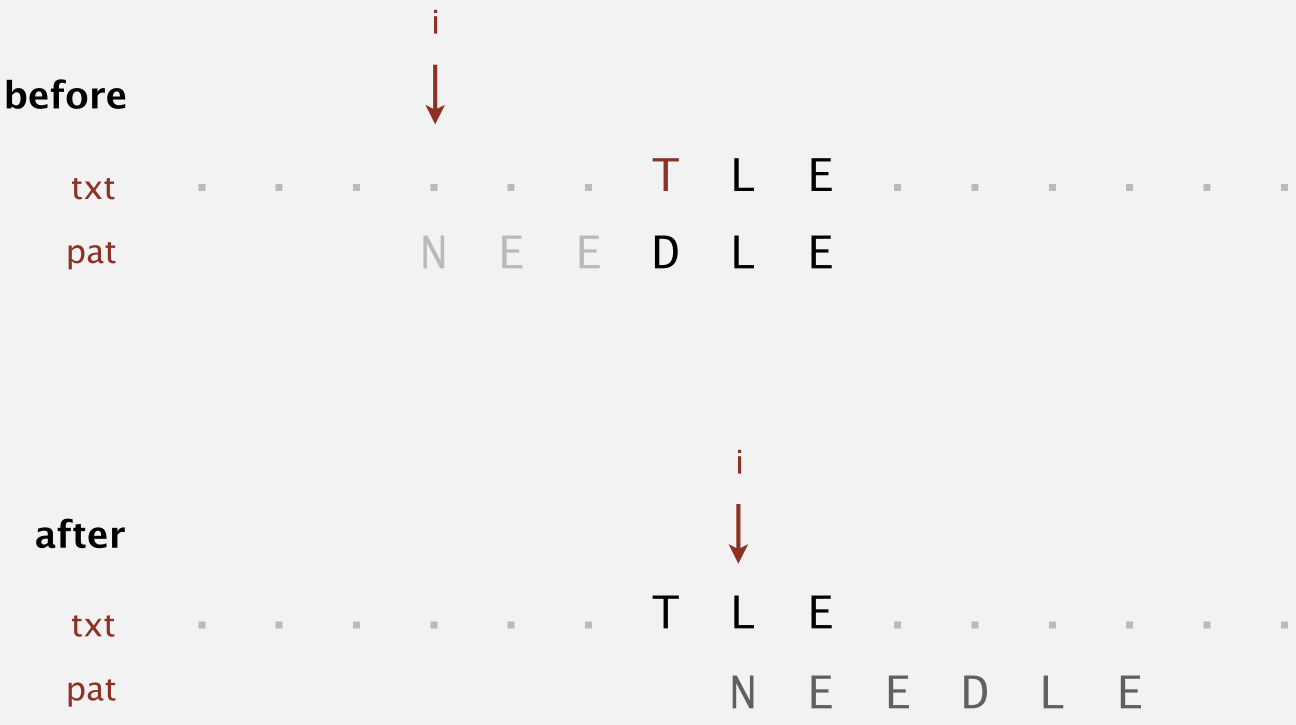
align N in text with N in pattern

align N in text with N in pattern

Boyer-Moore: mismatched character heuristic

Q. How much to skip?

Case 1. Mismatch character not in pattern.



before

i

txt . . . . . . T L E . . . . . .

pat N E E D L E

after

i

txt . . . . . . T L E . . . . . .

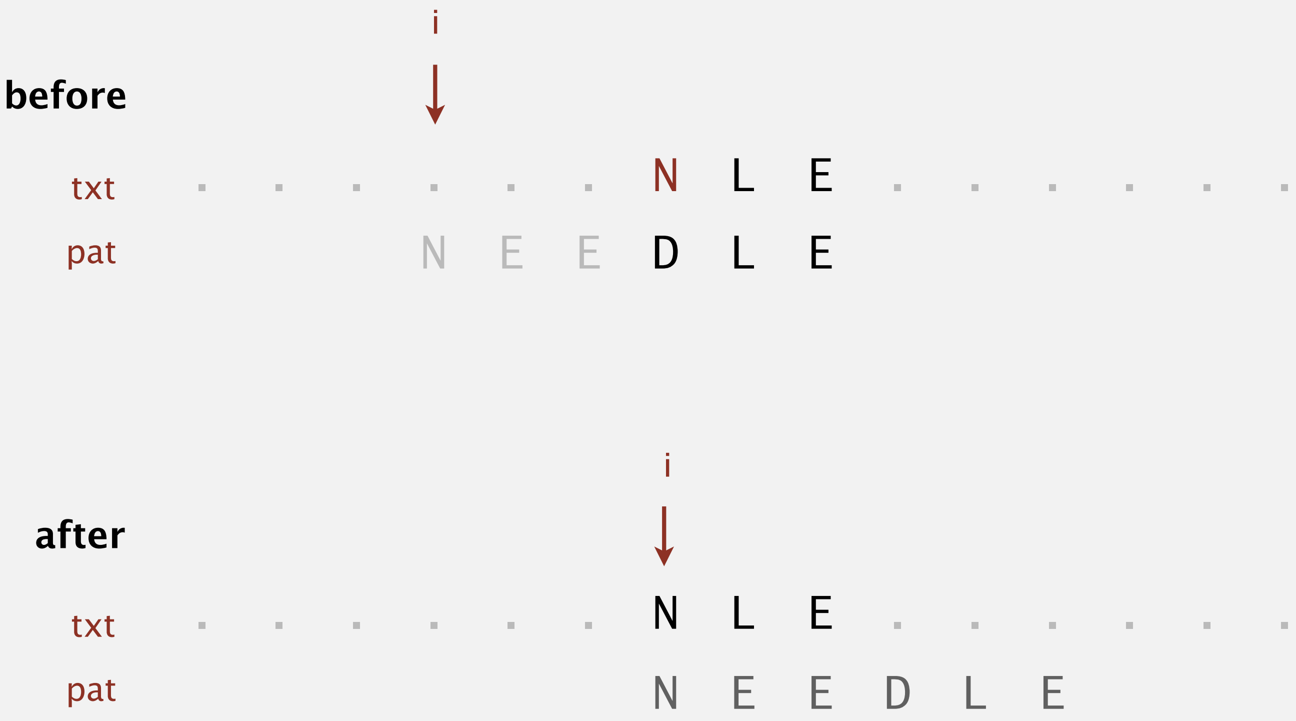
pat N E E D L E

mismatch character 'T' not in pattern: increment i one character beyond 'T'

Boyer-Moore: mismatched character heuristic

Q. How much to skip?

Case 2a. Mismatch character in pattern.



before

i

txt . . . . . . N L E . . . . . .

pat N E E D L E

after

i

txt . . . . . . N L E . . . . . .

pat N E E D L E

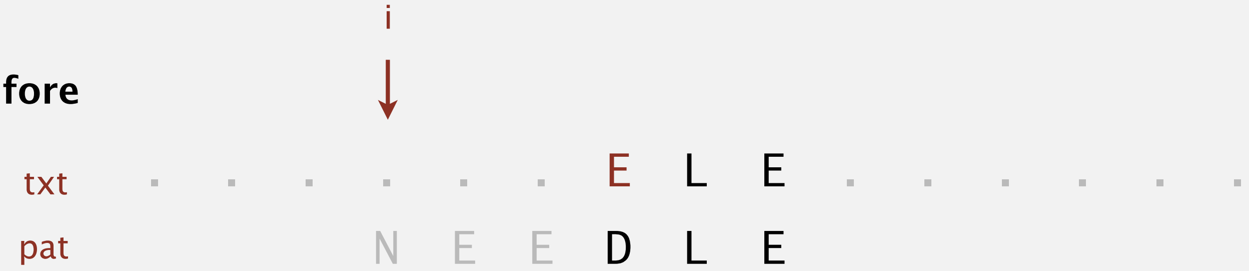
mismatch character 'N' in pattern: align text 'N' with rightmost (why?) pattern 'N'

Boyer-Moore: mismatched character heuristic

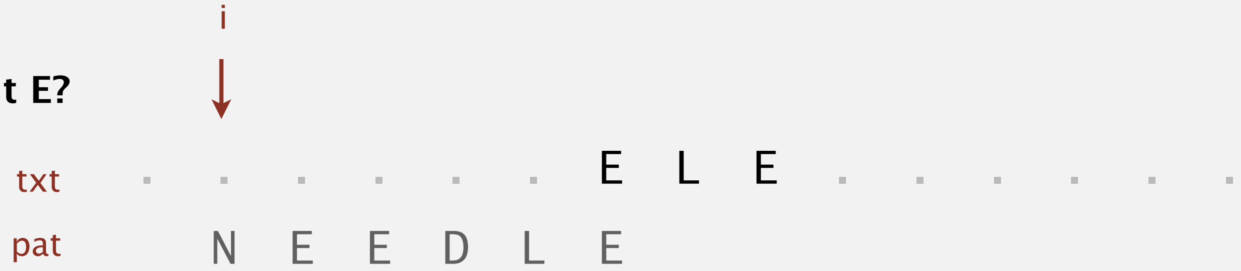
Q. How much to skip?

Case 2b. Mismatch character in pattern (but heuristic no help).

before



aligned with rightmost E?

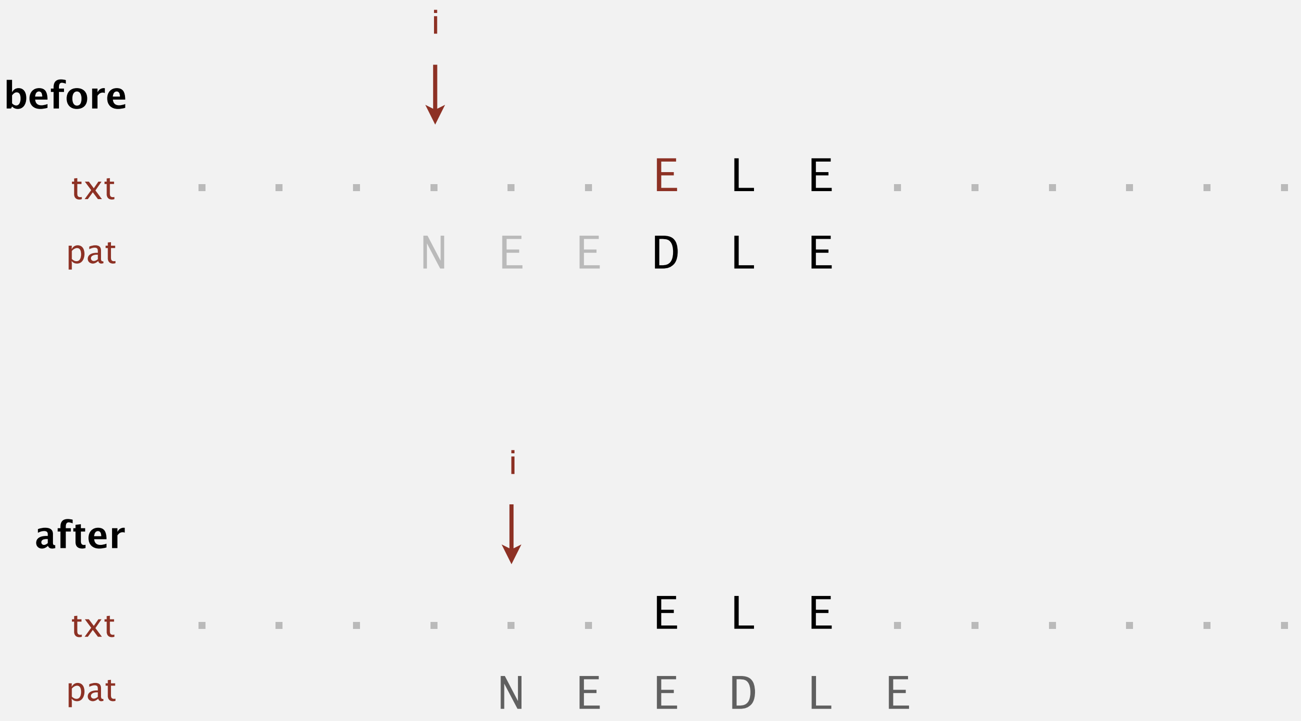


mismatch character 'E' in pattern: align text 'E' with rightmost pattern 'E' ?

Boyer-Moore: mismatched character heuristic

Q. How much to skip?

Case 2b. Mismatch character in pattern (but heuristic no help).


i
before
txt . . . . . . E L E . . . . . .
pat N E E D L E
i
after
txt . . . . . . E L E . . . . . .
pat N E E D L E

mismatch character 'E' in pattern: increment i by 1

Boyer-Moore: mismatched character heuristic

Q. How much to skip?

A. Precompute index of rightmost occurrence of character `c` in pattern. (-1 if character not in pattern)

	N	E	E	D	L	E	
c	0	1	2	3	4	5	right[c]
A							-1
B							-1
C							-1
D							3
E							5
...							-1
L							4
M							-1
N							0
...							-1

Boyer-Moore skip table computation

Boyer–Moore: analysis

Property. Substring search with the Boyer–Moore mismatched character heuristic takes about $\sim N / M$ character compares to search for a pattern of length M in a text of length N.

the longer the pattern, the faster to search!

Worst-case. Can be as bad as $\sim M N$.

Q. What's the worst-case input?

i	skip	0	1	2	3	4	5	6	7	8	9
	txt →	B	B	B	B	B	B	B	B	B	B
0	0	A	B	B	B	B	← pat				
1	1		A	B	B	B	B				
2	1			A	B	B	B	B			
3	1				A	B	B	B	B		
4	1					A	B	B	B	B	
5	1						A	B	B	B	B

Boyer–Moore variant. Can improve worst case to $\sim 3N$ character compares by adding a KMP-like rule to guard against repetitive patterns.

Algorithms

Robert Sedgewick | Kevin Wayne

http://algs4.cs.princeton.edu

5.3 Substring Search

- *introduction*
- *brute force*
- *Knuth–Morris–Pratt*
- *Boyer–Moore*
- *Rabin–Karp*

Michael Rabin
Dick Karp

Simplified example

Assume 10-character alphabet: abcdefghij

Text: `beachheadacidifiedjadedheadbeheadeadbeef`

Pattern: beheaded

“Hash” of string: number obtained replacing each char by corresponding digit

b	e	a	c	h	h	e	a	d	a	c	i	d	i	f	i	e	d	j	a
1	4	7	2	7	7	4	0	3	0	2	8	3	8	5	8	4	3	9	0
1	4	7	4	0	3	4	3												
b	e	h	e	a	d	e	d												

h_0 = 14727740

Precompute hash of pattern: h = 14740343

Compute h_0, h_0, h_2.... Match if h = h_0.

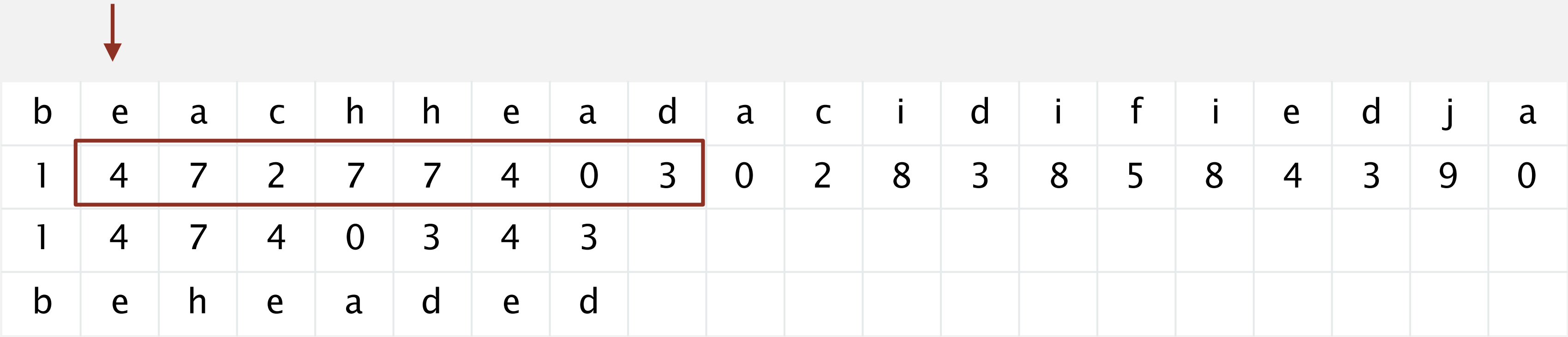
Simplified example

Assume 10-character alphabet: abcdefghij

Text: beachheadacidifiedjadedheadbeheadeadbeef

Pattern: beheaded

“Hash” of string: number obtained replacing each char by corresponding digit



b	e	a	c	h	h	e	a	d	a	c	i	d	i	f	i	e	d	j	a
1	4	7	2	7	7	4	0	3	0	2	8	3	8	5	8	4	3	9	0
1	4	7	4	0	3	4	3												
b	e	h	e	a	d	e	d												

h_0 = 14727740

h_1 = 47277403

Precompute hash of pattern: h = 14740343

Simplified example

Assume 10-character alphabet: abcdefghij

Text: beachheadacidifiedjadedheadbeheadeadbeef

Pattern: beheaded

"Hash" of string: number obtained replacing each char by corresponding digit

b	e	a	c	h	h	e	a	d	a	c	i	d	i	f	i	e	d	j	a
1	4	7	2	7	7	4	0	3	0	2	8	3	8	5	8	4	3	9	0
1	4	7	4	0	3	4	3												
b	e	h	e	a	d	e	d												

$h_1 = 47277403$

Precompute hash of pattern: $h = 14740343$

$h_2 = 72774030$

Q. Express h_{i+1} in terms of h_i, $t[0..N]$ (digits corresponding to text) and M

A. $h_{i+1} = (h_i - t_i \cdot 10^{M-1}) \cdot 10 + t_{i+M}$

Basic idea of Rabin-Karp

- Compute a hash of `pat[0..M)`.
- For each `i`, compute a hash of `txt[i..M+i)`.
- If pattern hash = text substring hash, declare match.

b	e	a	c	h	h	e	a	d	a	c	i	d	i	f	i	e	d	j	a
1	4	7	2	7	7	4	0	3	0	2	8	3	8	5	8	4	3	9	0
1	4	7	4	0	3	4	3												
b	e	h	e	a	d	e	d												

Problem 1: alphabet size R may not be 10

Problem 2: integer overflow if M is too long (M >= 10 for 32-bit `ints`)

Solution 1: use base R

Solution 2: do modulo Q arithmetic, where Q is a prime

Now it is an actual hash function — collisions exist.
Hash equality does not guarantee substring equality.

Rabin-Karp fingerprint search

Modular hashing.

- Compute a hash of `pat[0..M)`.
- For each `i`, compute a hash of `txt[i..M+i)`.
- If pattern hash = text substring hash, **check for a match**.

`pat.charAt(i)`

i	0	1	2	3	4	
	2	6	5	3	5	% 997 = 613

`txt.charAt(i)`

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3
0	3	1	4	1	5	% 997 = 508										
1		1	4	1	5	9	% 997 = 201									
2			4	1	5	9	2	% 997 = 715								
3				1	5	9	2	6	% 997 = 971							
4					5	9	2	6	5	% 997 = 442						
5						9	2	6	5	3	% 997 = 929					*match*
6 ← *return* i = 6							2	6	5	3	5	% 997 = 613				

modular hashing with R = 10 and hash(s) = s (mod 997)

Modular arithmetic

Math trick. To keep numbers small, take intermediate results modulo Q.

Ex.

$$\begin{aligned} &(10000 + 535) * 1000 && \pmod{997} \\ &= (30 + 535) * 3 && \pmod{997} \\ &= 1695 && \pmod{997} \\ &= 698 && \pmod{997} \end{aligned}$$

1000 mod 997 = 3

10000 mod 997 = 30

For more depth take COS 340

$$(a + b) \bmod Q \;=\; ((a \bmod Q) + (b \bmod Q)) \;\bmod Q$$

$$(a * b) \bmod Q \;=\; ((a \bmod Q) * (b \bmod Q)) \;\bmod Q$$

two useful modular arithmetic identities

Efficiently computing the hash function

Modular hash function. Using the notation t_i for `txt.charAt(i)`, we wish to compute

$$x_i \;=\; t_i\, R^{M-1} + t_{i+1}\, R^{\,M-2} + \ldots + t_{i+M-1}\, R^0 \pmod{Q}$$

Compare to hash tables lecture

Intuition. M-digit, base-R integer, modulo Q.

Horner's method. Linear-time method to evaluate degree-M polynomial.

```
   pat.charAt()
i  0 1 2 3 4
   2 6 5 3 5
0  2 % 997 = 2
                   R         Q
1  2 6 % 997 = (2*10 + 6) % 997 = 26
2  2 6 5 % 997 = (26*10 + 5) % 997 = 265
3  2 6 5 3 % 997 = (265*10 + 3) % 997 = 659
4  2 6 5 3 5 % 997 = (659*10 + 5) % 997 = 613
```

```
// Compute hash for M-digit key
private long hash(String key, int M)
{
   long h = 0;
   for (int j = 0; j < M; j++)
      h = (h * R + key.charAt(j)) % Q;
   return h;
}
```

26535 = 2*10000 + 6*1000 + 5*100 + 3*10 + 5

= ((((2) *10 + 6) * 10 + 5) * 10 + 3) * 10 + 5

Efficiently computing the hash function

Challenge. How to efficiently compute x_{i+1} given that we know x_i.

$$x_i \;=\; t_i\, R^{\,M-1} + t_{i+1}\, R^{\,M-2} + \ldots + t_{i+M-1}\, R^0$$

$$x_{i+1} = t_{i+1}\, R^{\,M-1} + t_{i+2}\, R^{\,M-2} + \ldots + t_{i+M}\, R^0$$

Key property. Can update "rolling" hash function in constant time!

$$x_{i+1} \;=\; (\, x_i \;-\; t_i\, R^{\,M-1}\,)\;\; R \;\;+\;\; t_{i\,+\,M}$$

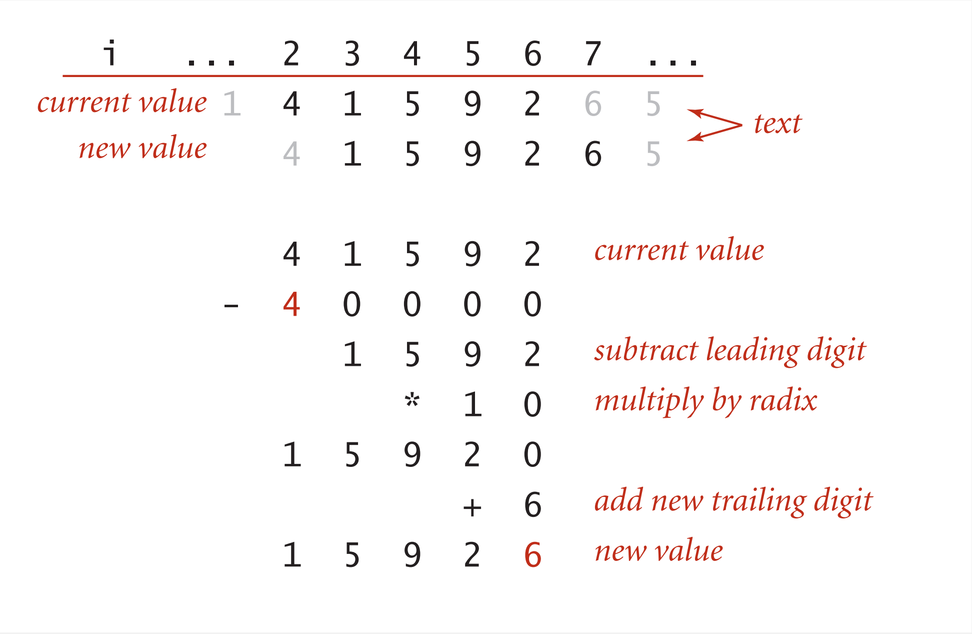
current value

subtract leading digit

multiply by radix

add new trailing digit

(can precompute R^{M-1})



i	...	2	3	4	5	6	7	...
current value	1	4	1	5	9	2	6	5
new value		4	1	5	9	2	6	5

text

```
    4  1  5  9  2     current value
 -  4  0  0  0  0
       1  5  9  2     subtract leading digit
          *  1  0     multiply by radix
    1  5  9  2  0
             +  6     add new trailing digit
    1  5  9  2  6     new value
```

Rabin-Karp: Java implementation

```
public class RabinKarp
{
   private long patHash;     // pattern hash value
   private int M;            // pattern length
   private long Q;           // modulus
   private int R;            // radix
   private long RM1;         // R^(M-1) % Q

   public RabinKarp(String pat) {
      M = pat.length();
      R = 256;
      Q = longRandomPrime();

      RM1 = 1;
      for (int i = 1; i <= M-1; i++)
         RM1 = (R * RM1) % Q;
      patHash = hash(pat, M);
   }

   private long hash(String key, int M)
   {  /* as before */  }

   public int search(String txt)
   {  /* see next slide */  }
}
```

a large prime
(but avoid overflow)

precompute R^{M-1} (mod Q)

Rabin-Karp: Java implementation (continued)

Monte Carlo version. Return match if hash match.

```
public int search(String txt)
{
    int N = txt.length();
    int txtHash = hash(txt, M);
    if (patHash == txtHash) return 0;
    for (int i = M; i < N; i++)
    {
        txtHash = (txtHash + Q - RM*txt.charAt(i-M) % Q) % Q;
        txtHash = (txtHash*R + txt.charAt(i)) % Q;
        if (patHash == txtHash) return i - M + 1;
    }
    return N;
}
```

check for hash collision
using rolling hash function

Las Vegas version. Modify code to check for substring match if hash match; continue search if false collision.

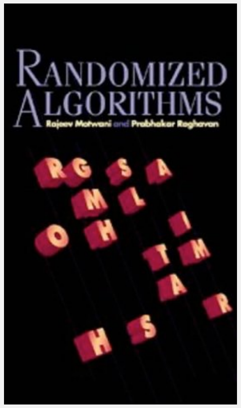
Rabin–Karp analysis

Theory. If Q is a sufficiently large random prime (about $M N^2$), then the probability of a false collision is about $1 / N$.

Practice. Choose Q to be a large prime (but not so large to cause overflow). Under reasonable assumptions, probability of a collision is about $1 / Q$.

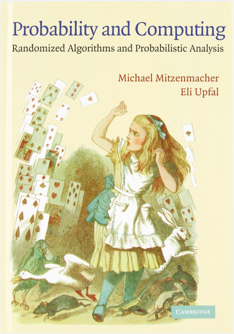
Monte Carlo version.

- Always runs in linear time.
- Extremely likely to return correct answer (but not always!).



Las Vegas version.

- Always returns correct answer.
- Extremely likely to run in linear time (but worst case is $M N$).


Probability and Computing
Randomized Algorithms and Probabilistic Analysis
Michael Mitzenmacher
Eli Upfal
Cambridge

Rabin–Karp fingerprint search

Advantages.

- Extends to two-dimensional patterns.
- Extends to finding multiple patterns.

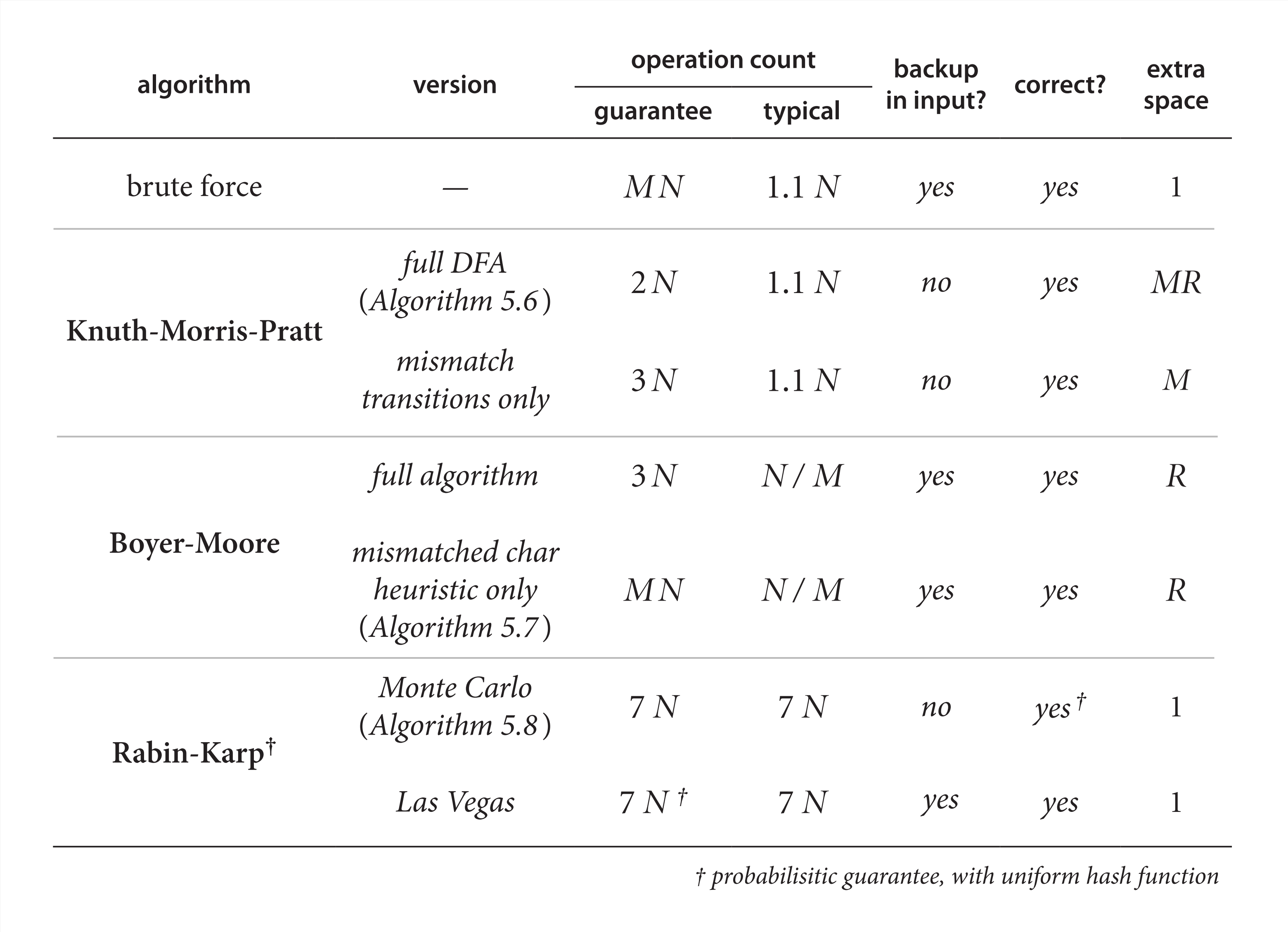
Disadvantages.

- Arithmetic ops slower than char compares.
- Las Vegas version requires backup.
- Poor worst-case guarantee.

Q. How would you extend Rabin–Karp to efficiently search for any one of P possible patterns in a text of length N?

Substring search cost summary

Cost of searching for an M-character pattern in an N-character text.



algorithm	version	operation count		backup in input?	correct?	extra space
		guarantee	typical			
brute force	—	MN	$1.1\ N$	*yes*	*yes*	1
Knuth-Morris-Pratt	*full DFA (Algorithm 5.6)*	$2N$	$1.1\ N$	*no*	*yes*	MR
	mismatch transitions only	$3N$	$1.1\ N$	*no*	*yes*	M
Boyer-Moore	*full algorithm*	$3N$	N / M	*yes*	*yes*	R
	mismatched char heuristic only (Algorithm 5.7)	MN	N / M	*yes*	*yes*	R
Rabin-Karp[†]	*Monte Carlo (Algorithm 5.8)*	$7\ N$	$7\ N$	*no*	*yes* [†]	1
	Las Vegas	$7\ N$ [†]	$7\ N$	*yes*	*yes*	1

† probabilisitic guarantee, with uniform hash function

Substring search quiz ∞

Which of today's algorithms do you like the best?

A. Knuth-Morris-Pratt (finite automaton).

B. Boyer–Moore (skip–ahead heuristic).

C. Rabin-Karp (rolling hash function).

D. *It's all a blur.*