BBM 202 - ALGORITHMS

HACETTEPE UNIVERSITY

DEPT. OF COMPUTER ENGINEERING

SUBSTRING SEARCH

Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.

String processing

String. Sequence of characters.

Important fundamental abstraction.

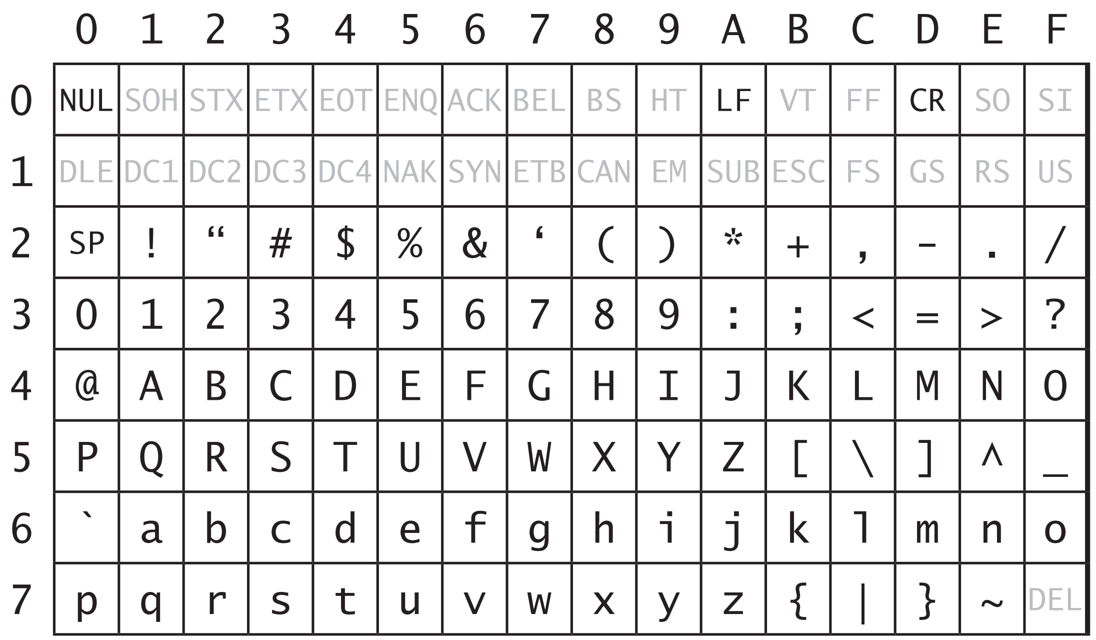
- Information processing.
- Genomic sequences.
- Communication systems (e.g., email).
- Programming systems (e.g., Java programs).
- …

> “ *The digital information that underlies biochemistry, cell biology, and development can be represented by a simple string of G's, A's, T's and C's. This string is the root data structure of an organism's biology.* ” — *M. V. Olson*

The char data type

C char data type. Typically an 8-bit integer.

- Supports 7-bit ASCII.
- Need more bits to represent certain characters.



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	“	#	$	%	&	‘	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Hexadecimal to ASCII conversion table

Java char data type. A 16-bit unsigned integer.

- Supports original 16-bit Unicode.
- Supports 21-bit Unicode 3.0 (awkwardly).

A	á	∂	𝔊
U+0041	U+00E1	U+2202	U+1D50A

Unicode characters

I (heart) Unicode

The String data type

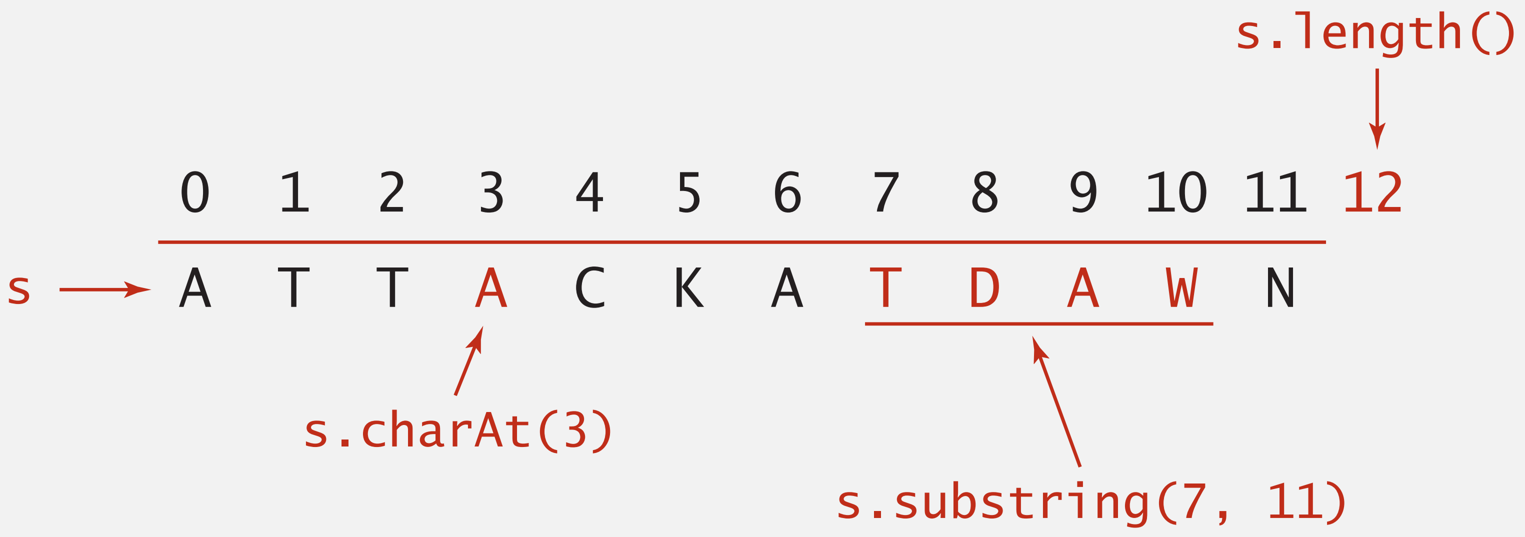
String data type. Sequence of characters (immutable).

Length. Number of characters.

Indexing. Get the i^{th} character.

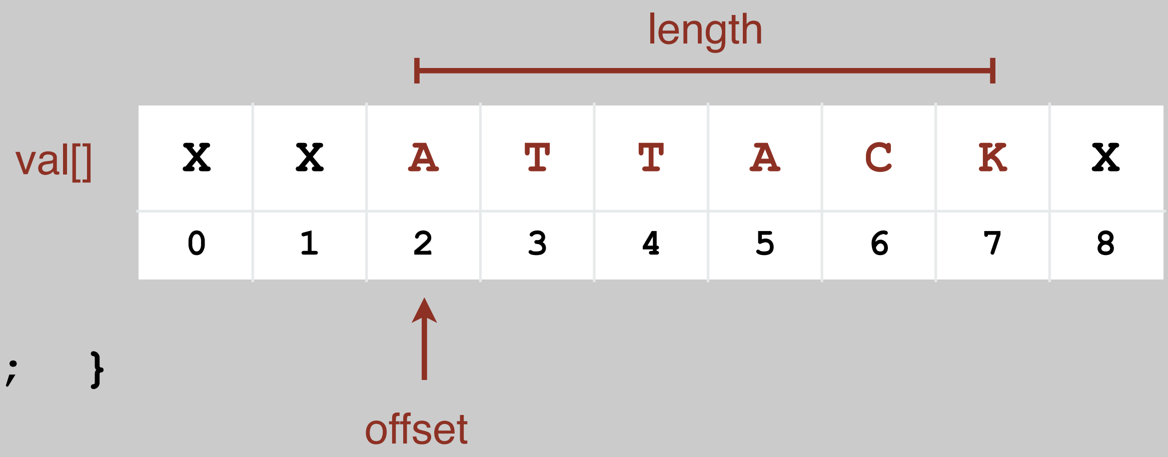
Substring extraction. Get a contiguous sequence of characters.

String concatenation. Append one character to end of another string.



The String data type: Java implementation

```
public final class String implements Comparable<String>
{
   private char[] val;    // characters
   private int offset;    // index of first char in array
   private int length;    // length of string
   private int hash;      // cache of hashCode()


   public int length()
   {  return length; }

   public char charAt(int i)
   {  return value[i + offset];   }


   private String(int offset, int length, char[] val)
   {
      this.offset = offset;
      this.length = length;
      this.val    = val;
   }

   public String substring(int from, int to)
   {  return new String(offset + from, to - from, val);   }
   …
```


length

val[]	X	X	A	T	T	A	C	K	X
	0	1	2	3	4	5	6	7	8

offset

copy of reference to original char array

The String data type: performance

String data type. Sequence of characters (immutable).

Design Choice. Immutable, cache or share the backing array

Underlying implementation. Immutable `char[]` array, offset, and length.

	`String`	
operation	guarantee	extra space
`length()`	1	1
`charAt()`	1	1
`substring()`	1	1
`concat()`	N	N

Memory. $40 + 2N$ bytes for a virgin `String` of length N.

can use `byte[]` or `char[]` instead of `String` to save space

(but lose convenience of `String` data type)

The StringBuilder data type

StringBuilder data type. Sequence of characters (mutable).

Design Choice. Easier to update, can't cache or share array.

Underlying implementation. Resizing `char[]` array and length.

	`String`		`StringBuilder`	
operation	guarantee	extra space	guarantee	extra space
`length()`	1	1	1	1
`charAt()`	1	1	1	1
`substring()`	1	1	N	N
`concat()`	N	N	1 *	1 *

* amortized

Actually as of Java 1.7 this is O(n) for String as well. Before 1.7 the initial String and substring shared the backing array (no need to copy!)

Remark. `StringBuffer` data type is similar, but thread safe (and slower).

String vs. StringBuilder

Q. How to efficiently reverse a string?

A.

```
public static String reverse(String s)
{
   String rev = "";
   for (int i = s.length() - 1; i >= 0; i--)
      rev += s.charAt(i);
   return rev;
}
```

quadratic time

String concatenation creates a new String and all chars in backing array are copied to new one.

B.

```
public static String reverse(String s)
{
   StringBuilder rev = new StringBuilder();
   for (int i = s.length() - 1; i >= 0; i--)
      rev.append(s.charAt(i));
   return rev.toString();
}
```

linear time

The backing array is updated. Sometimes may need to expand the array but amortised cost is O(1)

String challenge: array of suffixes

Q. How to efficiently form array of suffixes?

input string

a	a	c	a	a	g	t	t	t	a	c	a	a	g	c
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

suffixes

0	a	a	c	a	a	g	t	t	t	a	c	a	a	g	c
1	a	c	a	a	g	t	t	t	a	c	a	a	g	c	
2	c	a	a	g	t	t	t	a	c	a	a	g	c		
3	a	a	g	t	t	t	a	c	a	a	g	c			
4	a	g	t	t	t	a	c	a	a	g	c				
5	g	t	t	t	a	c	a	a	g	c					
6	t	t	t	a	c	a	a	g	c						
7	t	t	a	c	a	a	g	c							
8	t	a	c	a	a	g	c								
9	a	c	a	a	g	c									
10	c	a	a	g	c										
11	a	a	g	c											
12	a	g	c												
13	g	c													
14	c														

String vs. StringBuilder

Q. How to efficiently form array of suffixes?

A.

```
public static String[] suffixes(String s)
{
   int N = s.length();
   String[] suffixes = new String[N];
   for (int i = 0; i < N; i++)
      suffixes[i] = s.substring(i, N);
   return suffixes;
}
```

linear time and
linear space

Since Strings are immutable, the backing array of larger String can be shared with substring. In Java 1.7 they changed it, now cost is the same as below!

B.

```
public static String[] suffixes(String s)
{
   int N = s.length();
   StringBuilder sb = new StringBuilder(s);
   String[] suffixes = new String[N];
   for (int i = 0; i < N; i++)
      suffixes[i] = sb.substring(i, N);
   return suffixes;
}
```

quadratic time and
quadratic space

The array of StringBuilder can change, so can't share with substring.

Longest common prefix

Q. How long to compute length of longest common prefix?

p	r	e	f	e	t	c	h
0	1	2	3	4	5	6	7
p	r	e	f	i	x		

```
public static int lcp(String s, String t)
{
   int N = Math.min(s.length(), t.length());
   for (int i = 0; i < N; i++)
      if (s.charAt(i) != t.charAt(i))
         return i;
   return N;
}
```

linear time (worst case)

sublinear time (typical case)

Running time. Proportional to length D of longest common prefix.

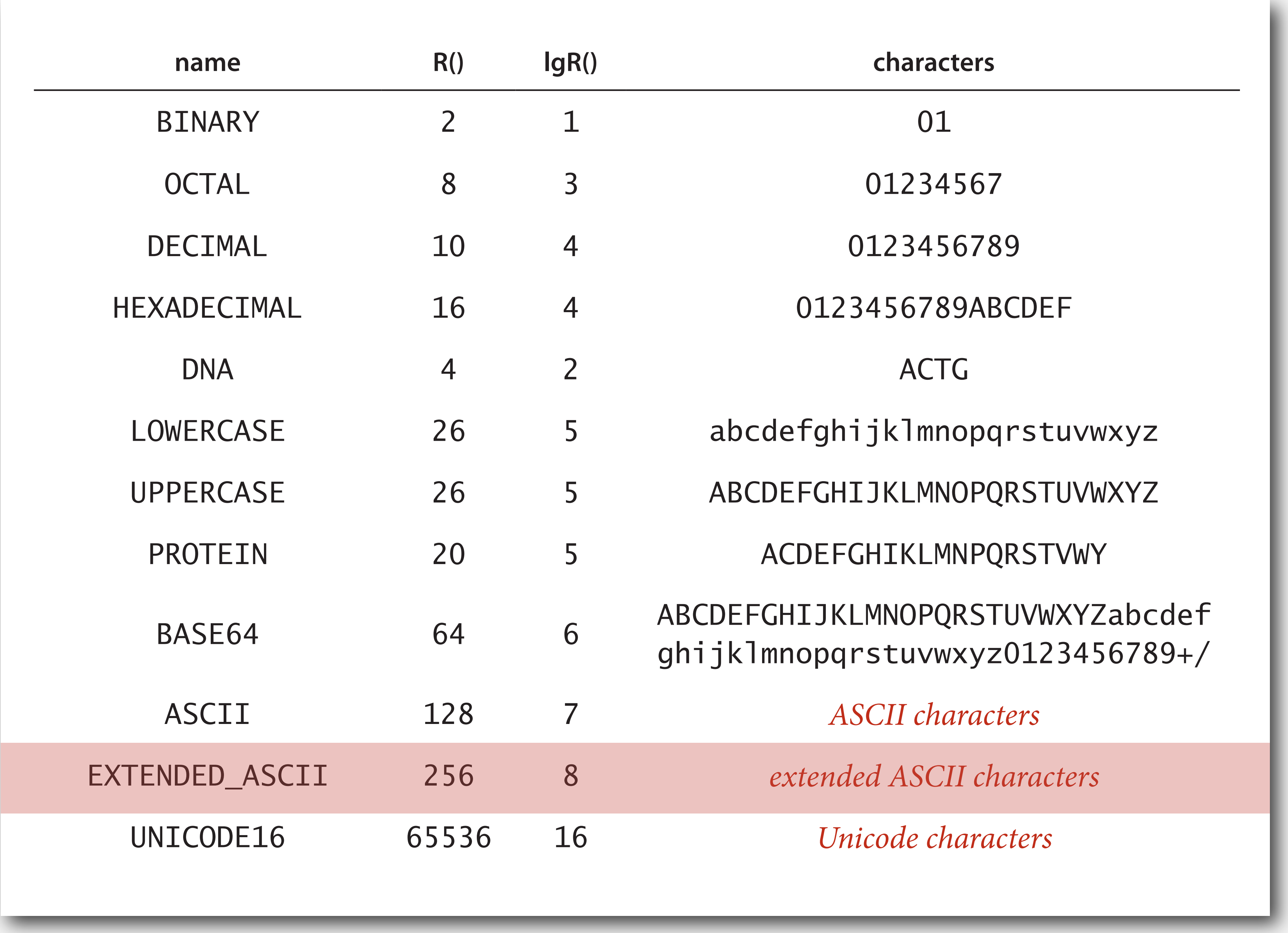
Remark. Also can compute `compareTo()` in sublinear time.

Alphabets

Digital key. Sequence of digits over fixed alphabet.

Radix. Number of digits R in alphabet.

Complexity of some algorithms will depend on this



name	R()	lgR()	characters
BINARY	2	1	01
OCTAL	8	3	01234567
DECIMAL	10	4	0123456789
HEXADECIMAL	16	4	0123456789ABCDEF
DNA	4	2	ACTG
LOWERCASE	26	5	abcdefghijklmnopqrstuvwxyz
UPPERCASE	26	5	ABCDEFGHIJKLMNOPQRSTUVWXYZ
PROTEIN	20	5	ACDEFGHIKLMNPQRSTVWY
BASE64	64	6	ABCDEFGHIJKLMNOPQRSTUVWXYZabcdef ghijklmnopqrstuvwxyz0123456789+/
ASCII	128	7	*ASCII characters*
EXTENDED_ASCII	256	8	*extended ASCII characters*
UNICODE16	65536	16	*Unicode characters*

TODAY

- Substring search
- Brute force
- Knuth-Morris-Pratt
- Boyer-Moore
- Rabin-Karp

Substring search

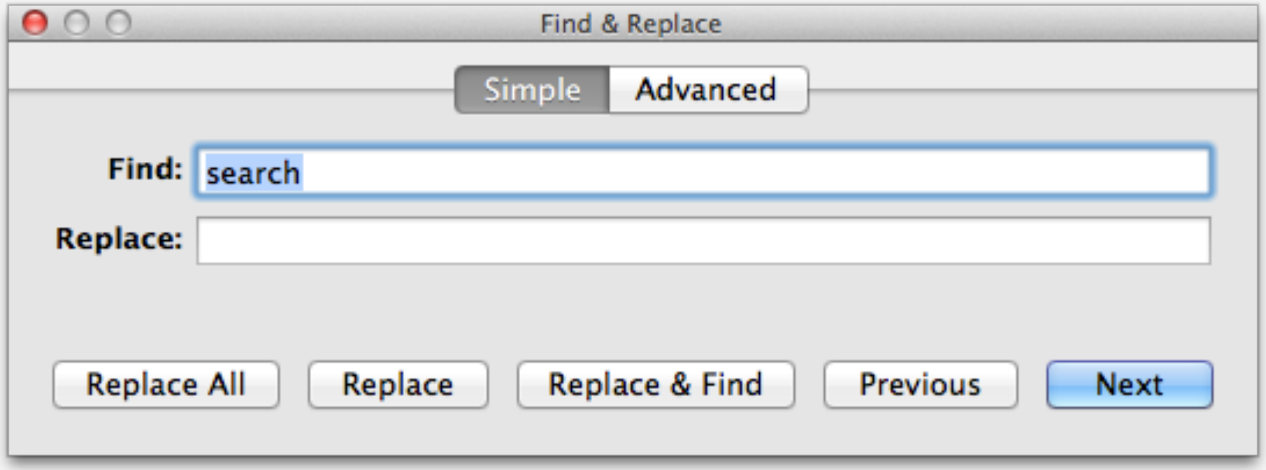
Goal. Find pattern of length M in a text of length N.

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



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

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

Identify patterns indicative of spam.

- `PROFITS`
- `L0SE WE1GHT`
- `There is no catch.`
- `This is a one-time mailing.`
- `This message is sent in compliance with spam regulations.`

SpamAssassin

Substring search applications

Electronic surveillance.

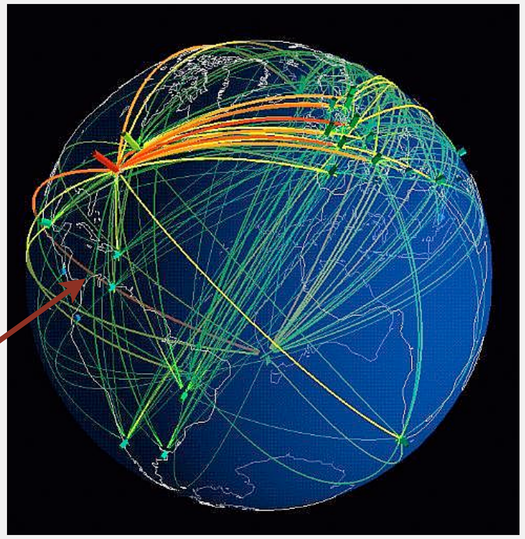
Need to monitor all internet traffic. (security)

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

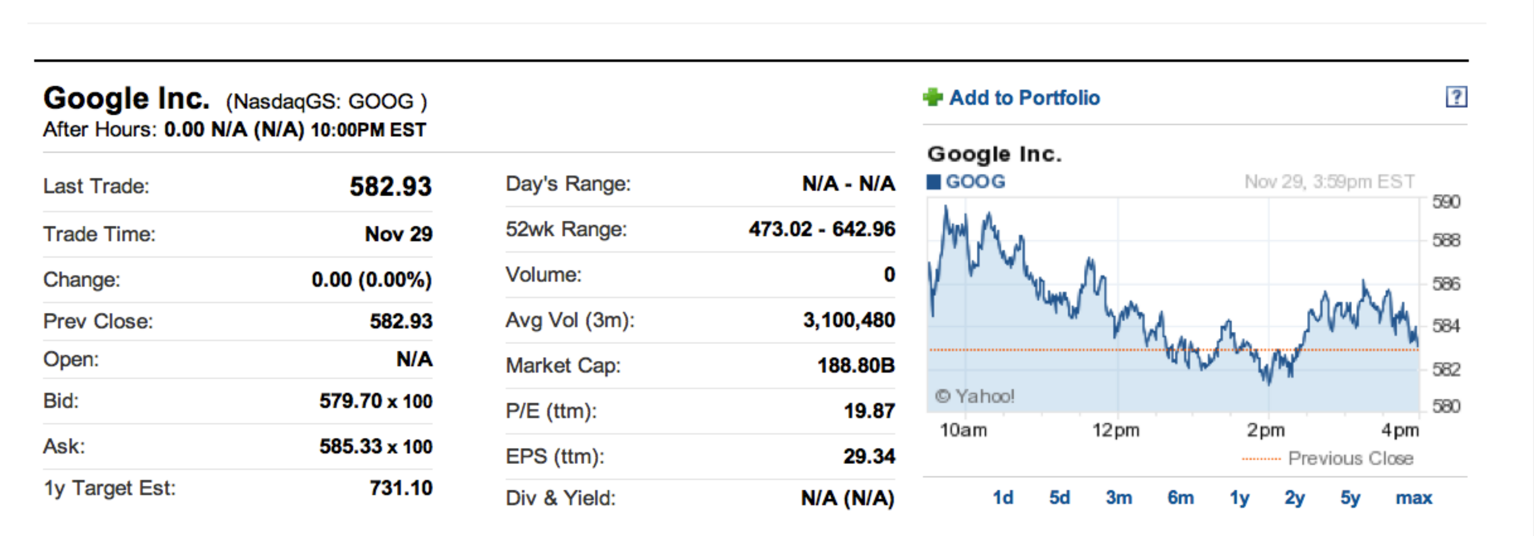


found

Substring search applications

Screen scraping. Extract relevant data from web page.

Ex. Find string delimited by `<b>` and `</b>` after first occurrence of pattern `Last Trade:`.



Google Inc. (NasdaqGS: GOOG)
After Hours: **0.00 N/A (N/A) 10:00PM EST**

Last Trade:	**582.93**	Day's Range:	**N/A - N/A**
Trade Time:	**Nov 29**	52wk Range:	**473.02 - 642.96**
Change:	**0.00 (0.00%)**	Volume:	**0**
Prev Close:	**582.93**	Avg Vol (3m):	**3,100,480**
Open:	**N/A**	Market Cap:	**188.80B**
Bid:	**579.70 x 100**	P/E (ttm):	**19.87**
Ask:	**585.33 x 100**	EPS (ttm):	**29.34**
1y Target Est:	**731.10**	Div & Yield:	**N/A (N/A)**

http://finance.yahoo.com/q?s=goog

```
...
<tr>
<td class= "yfnc_tablehead1"
width= "48%">
Last Trade:
</td>
<td class= "yfnc_tabledata1">
<big><b>452.92</b></big>
</td></tr>
<td class= "yfnc_tablehead1"
width= "48%">
Trade Time:
</td>
<td class= "yfnc_tabledata1">
...
```

Screen scraping: Java implementation

Java library. The `indexOf()` method in Java's string library returns the index of the first occurrence of a given string, starting at a given offset.

```
public class StockQuote
{
   public static void main(String[] args)
   {
      String name = "http://finance.yahoo.com/q?s=";
      In in = new In(name + args[0]);
      String text = in.readAll();
      int start    = text.indexOf("Last Trade:", 0);
      int from     = text.indexOf("<b>",  start);
      int to       = text.indexOf("</b>", from);
      String price = text.substring(from + 3, to);
      StdOut.println(price);
   }
}
```

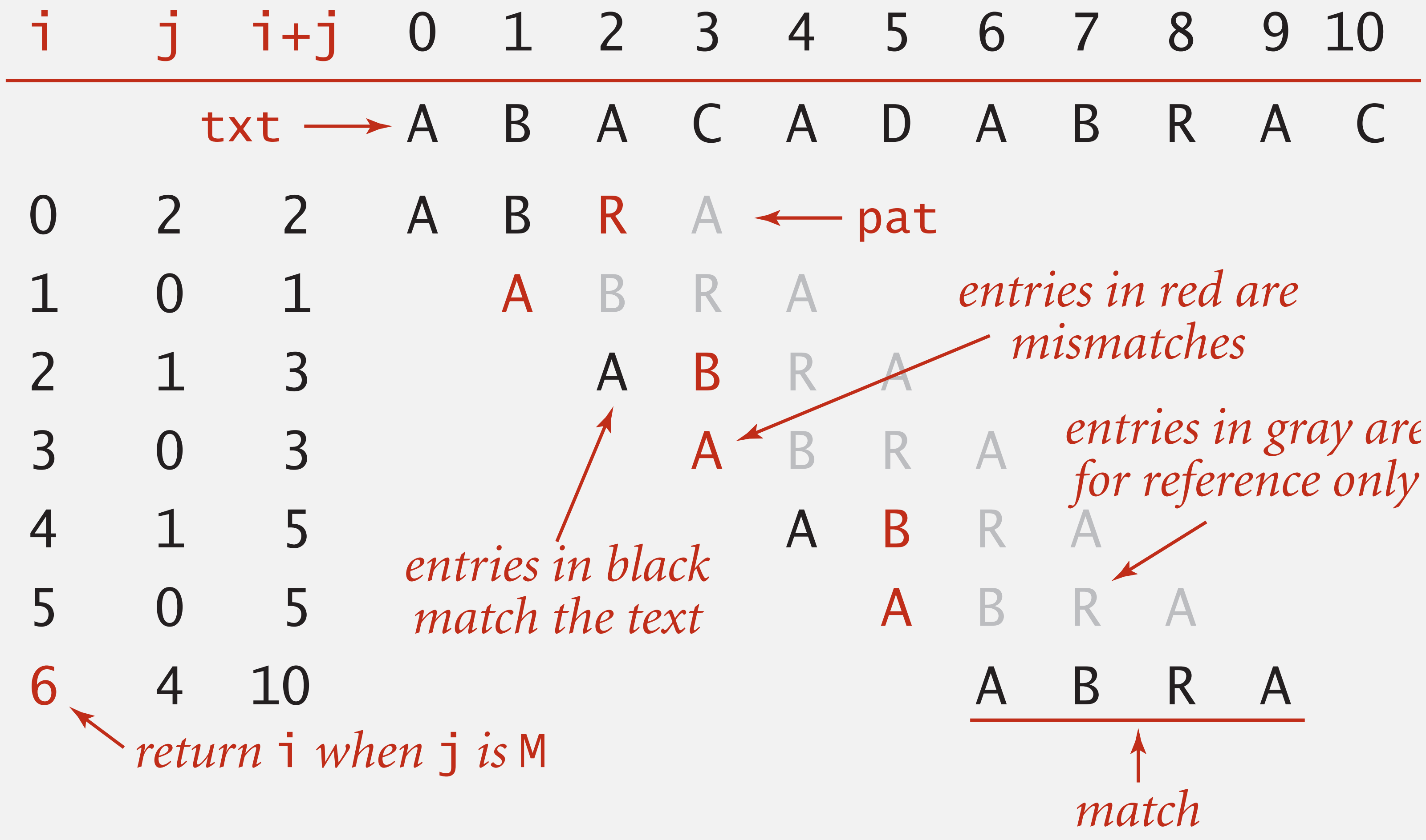
```
% java StockQuote goog
582.93

% java StockQuote msft
24.84
```

SUBSTRING SEARCH

- **Brute force**
- Knuth-Morris-Pratt
- Boyer-Moore
- Rabin-Karp

Brute-force substring search

Check for pattern starting at each text position.



`i`	`j`	`i+j`	0	1	2	3	4	5	6	7	8	9	10
		`txt` →	A	B	A	C	A	D	A	B	R	A	C
0	2	2	A	B	R	A							
1	0	1		A	B	R	A						
2	1	3			A	B	R	A					
3	0	3				A	B	R	A				
4	1	5					A	B	R	A			
5	0	5						A	B	R	A		
6	4	10							A	B	R	A	

← `pat`

entries in red are mismatches

entries in gray are for reference only

entries in black match the text

return `i` *when* `j` *is* `M`

match

Brute-force substring search: Java implementation

Check for pattern starting at each text position.

i	j	i+j	0	1	2	3	4	5	6	7	8	9	10
			A	B	A	C	A	D	A	B	R	A	C
4	3	7					A	D	A	C	R		
5	0	5						A	D	A	C	R	

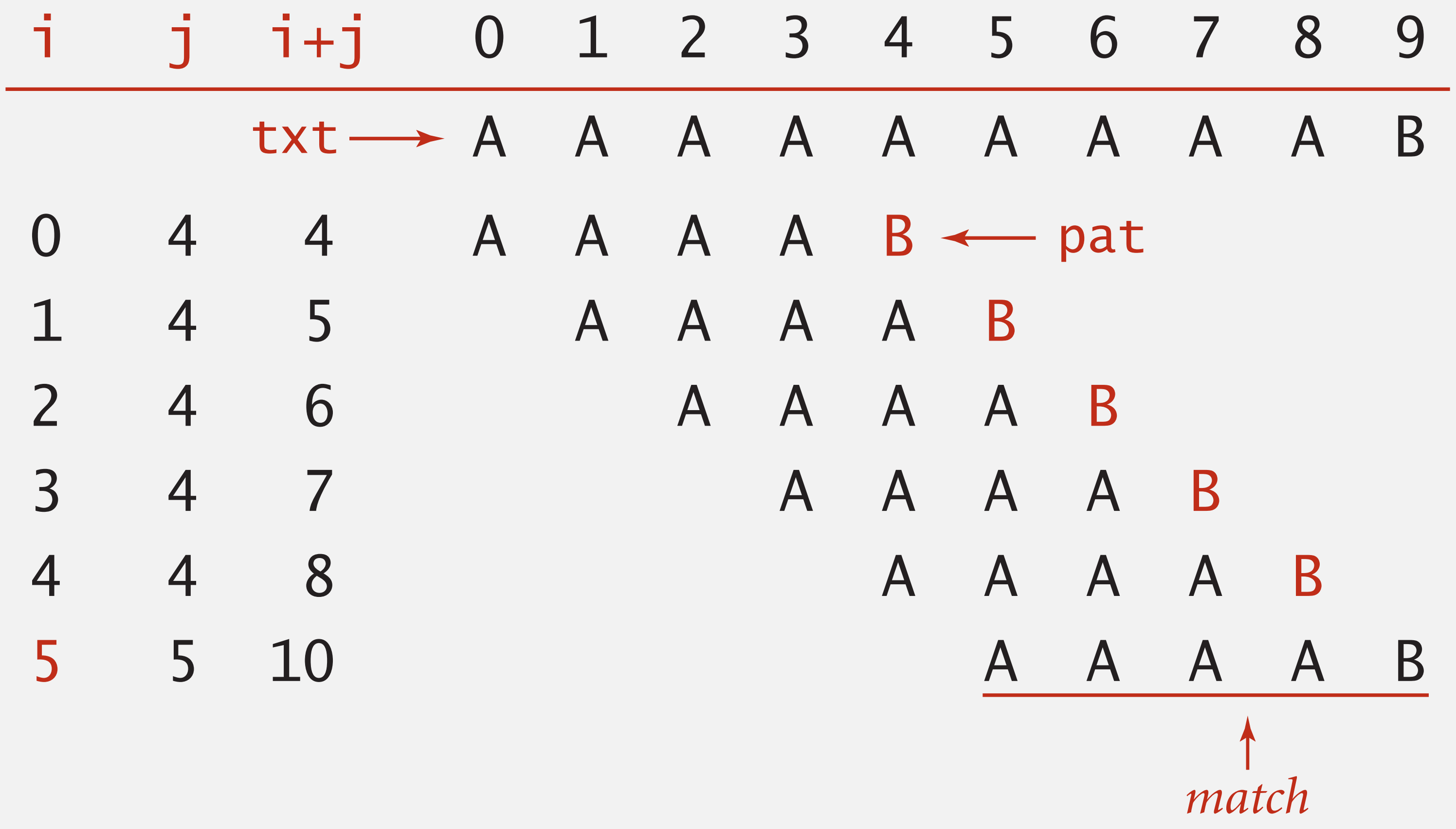
```
public static int search(String pat, String txt)
{
   int M = pat.length();
   int N = txt.length();
   for (int i = 0; i <= N - M; i++)
   {
      int j;
      for (j = 0; j < M; j++)
         if (txt.charAt(i+j) != pat.charAt(j))
            break;
      if (j == M) return i;
   }
   return N;
}
```

`if (j == M) return i;` ← index in text where pattern starts

`return N;` ← not found

Brute-force substring search: worst case

Brute-force algorithm can be slow if text and pattern are repetitive.



i	j	i+j	0	1	2	3	4	5	6	7	8	9
		txt →	A	A	A	A	A	A	A	A	A	B
0	4	4	A	A	A	A	B	← pat				
1	4	5		A	A	A	A	B				
2	4	6			A	A	A	A	B			
3	4	7				A	A	A	A	B		
4	4	8					A	A	A	A	B	
5	5	10						A	A	A	A	B

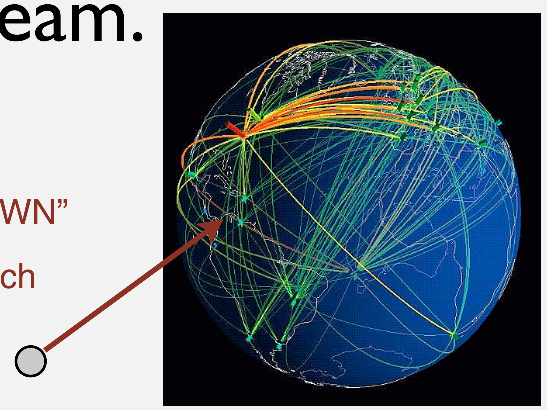
match

Worst case. $\sim M\,N$ char compares.

Backup

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

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

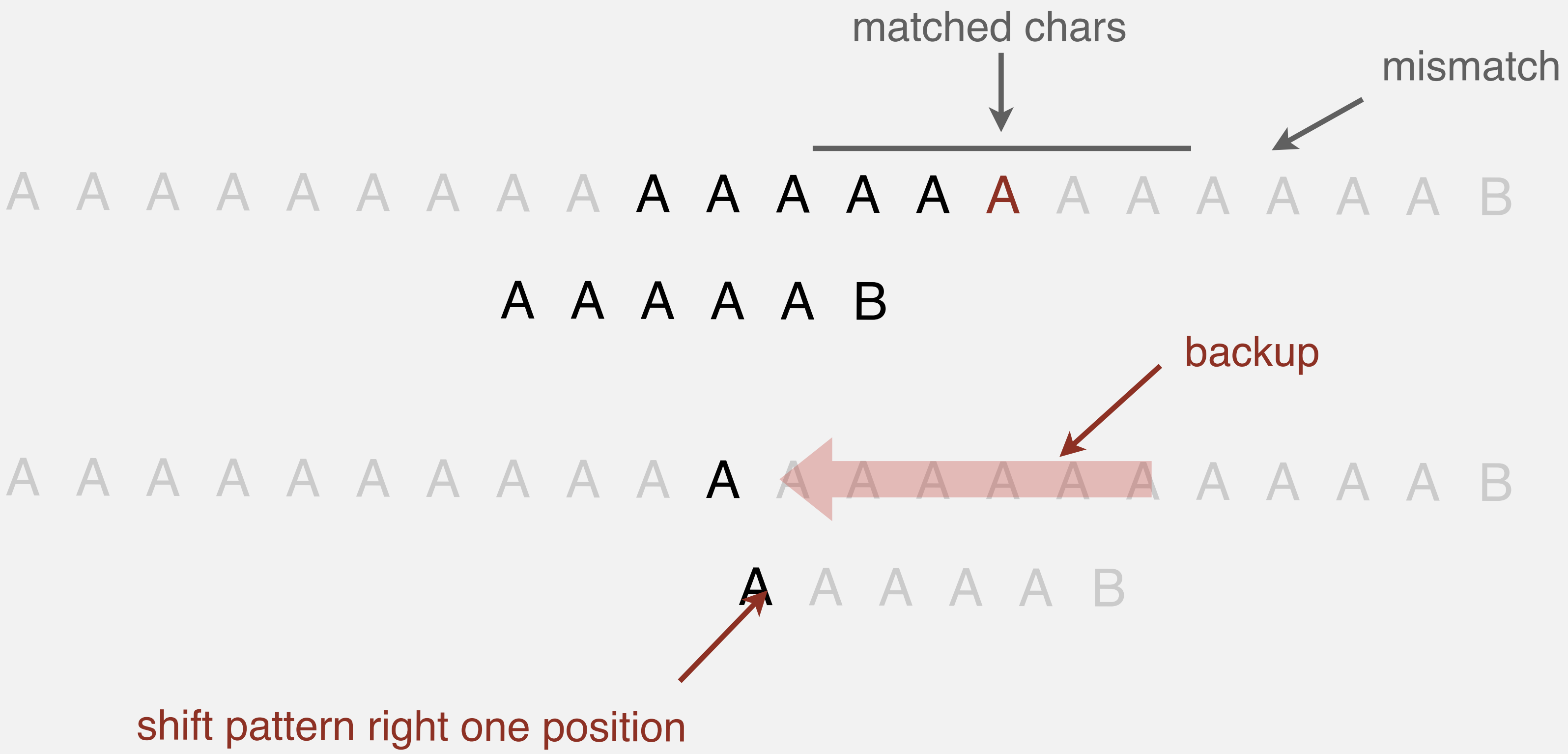


“ATTACK AT DAWN”

substring search machine

found

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. Stay tuned.

Brute-force substring search: alternate implementation

Same sequence of char compares as previous implementation.

- `i` points to end of sequence of already-matched chars in text.
- `j` stores number of already-matched chars (end of sequence in pattern).

i	j	0	1	2	3	4	5	6	7	8	9	10
		A	B	A	C	A	D	A	B	R	A	C
7	3					A	D	A	C	R		
5	0						A	D	A	C	R	

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

backup

Algorithmic challenges in substring search

Brute-force is not always good enough.

Theoretical challenge. Linear-time guarantee. ← fundamental algorithmic problem

Practical challenge. Avoid backup in text stream. ← often no room or time to save text

Now is the time for all people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for many good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for a lot of good people to come to the aid of their party. Now is the time for all of the good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for each good person to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for all good Republicans to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for many or all good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for all good Democrats to come to the aid of their party. Now is the time for all people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for many good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for a lot of good people to come to the aid of their party. Now is the time for all of the good people to come to the aid of their party. Now is the time for all good people to come to the aid of their attack at dawn party. Now is the time for each person to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for all good Republicans to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for many or all good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for all good Democrats to come to the aid of their party.

SUBSTRING SEARCH

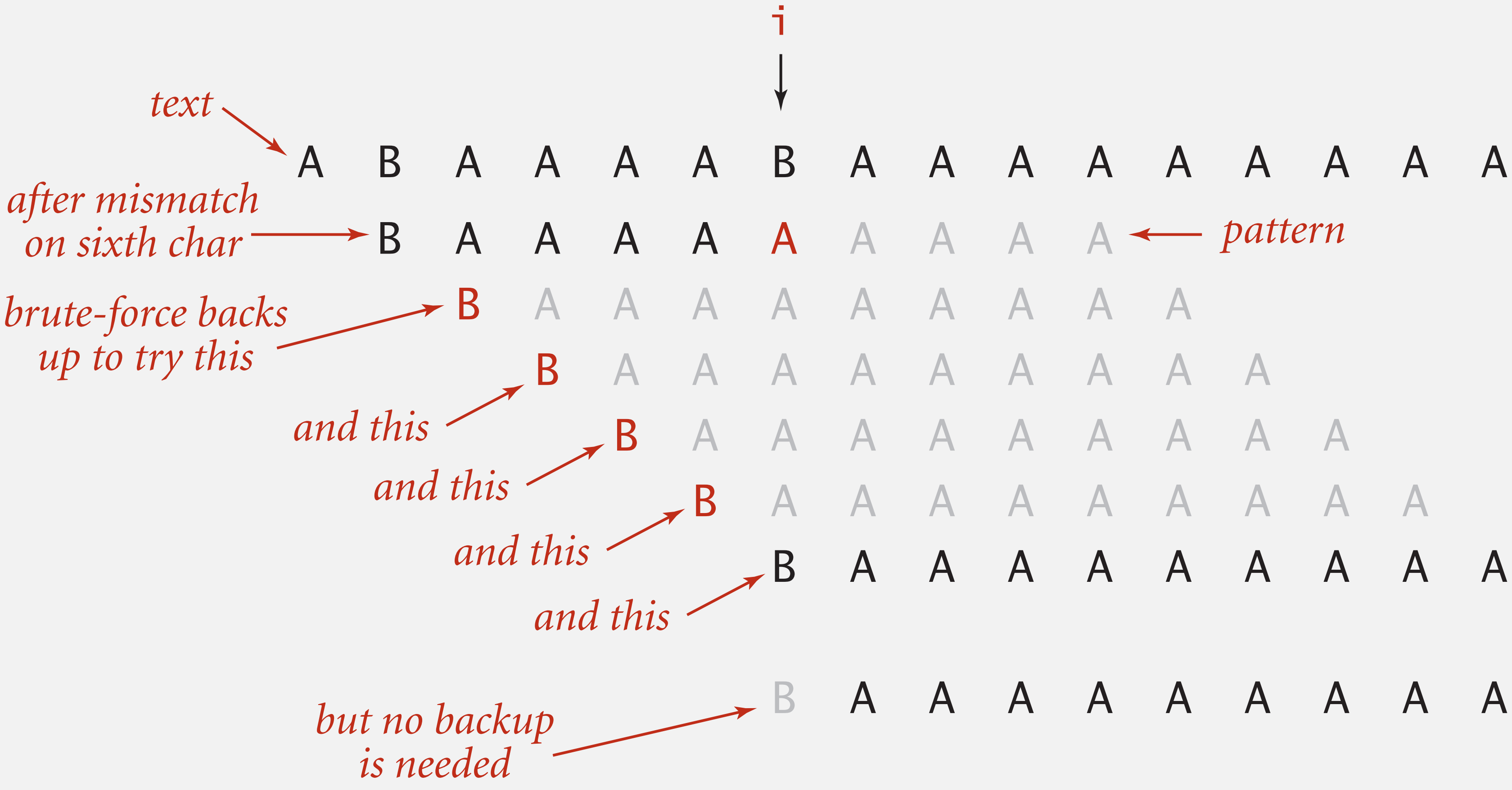
- Brute force
- **Knuth-Morris-Pratt**
- Boyer-Moore
- Rabin-Karp

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 6^{th} 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. (!)

Deterministic finite state automaton (DFA)

DFA is abstract string-searching machine.

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

internal representation

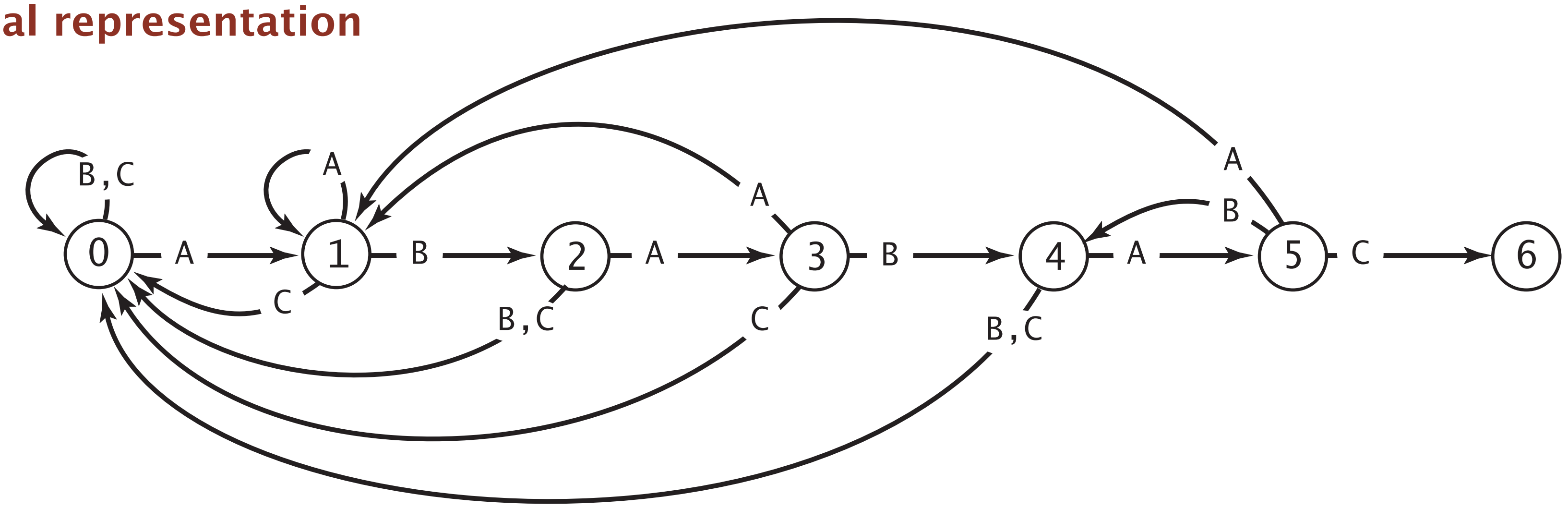
j		0	1	2	3	4	5
`pat.charAt(j)`		A	B	A	B	A	C
`dfa[][j]`	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6

If in state **`j`** reading char **`c`**:

if **`j`** is **6** halt and accept

- else move to state **`dfa[c][j]`**

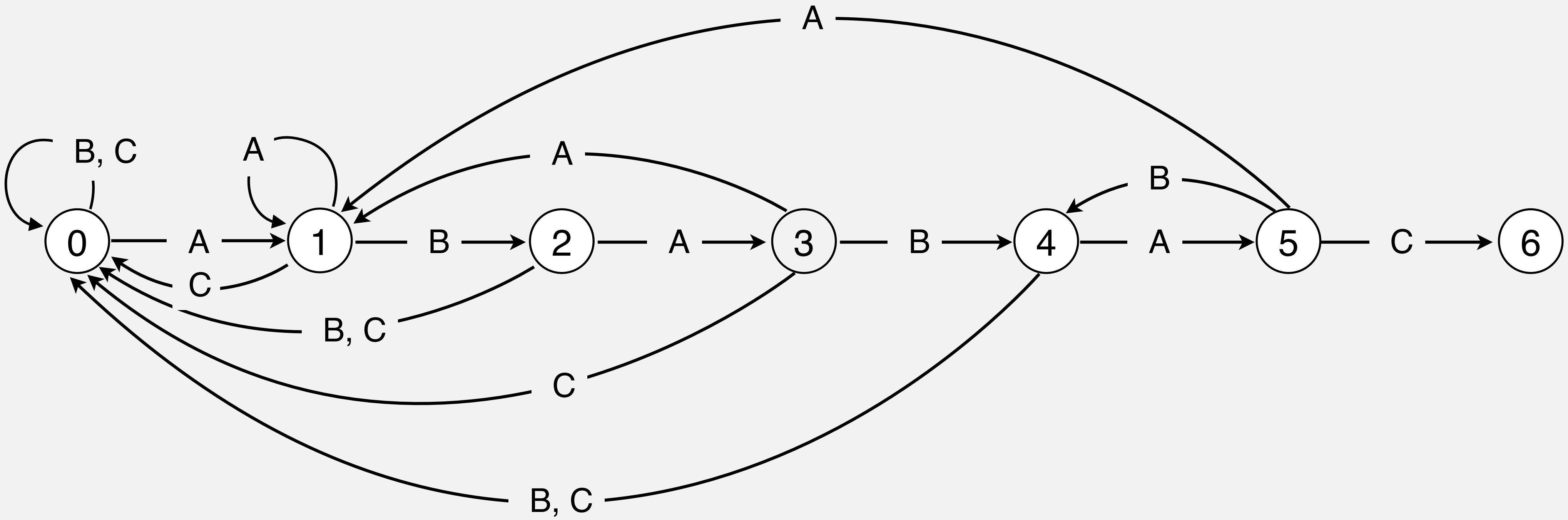
graphical representation



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

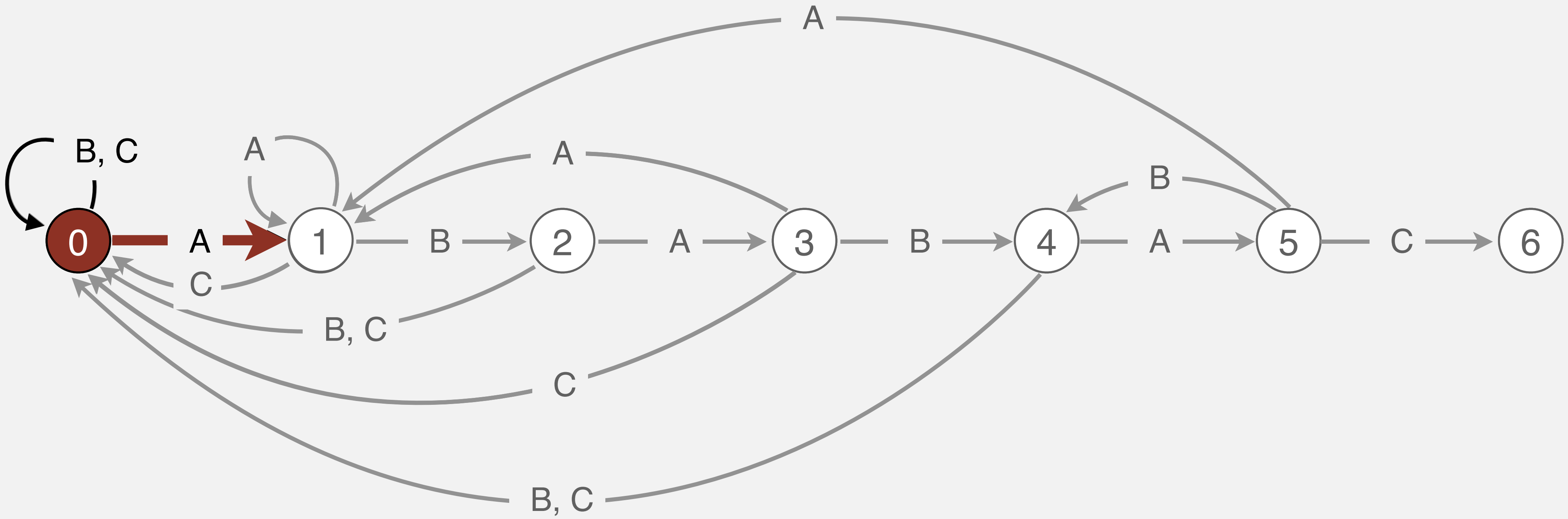


DFA simulation

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

↑

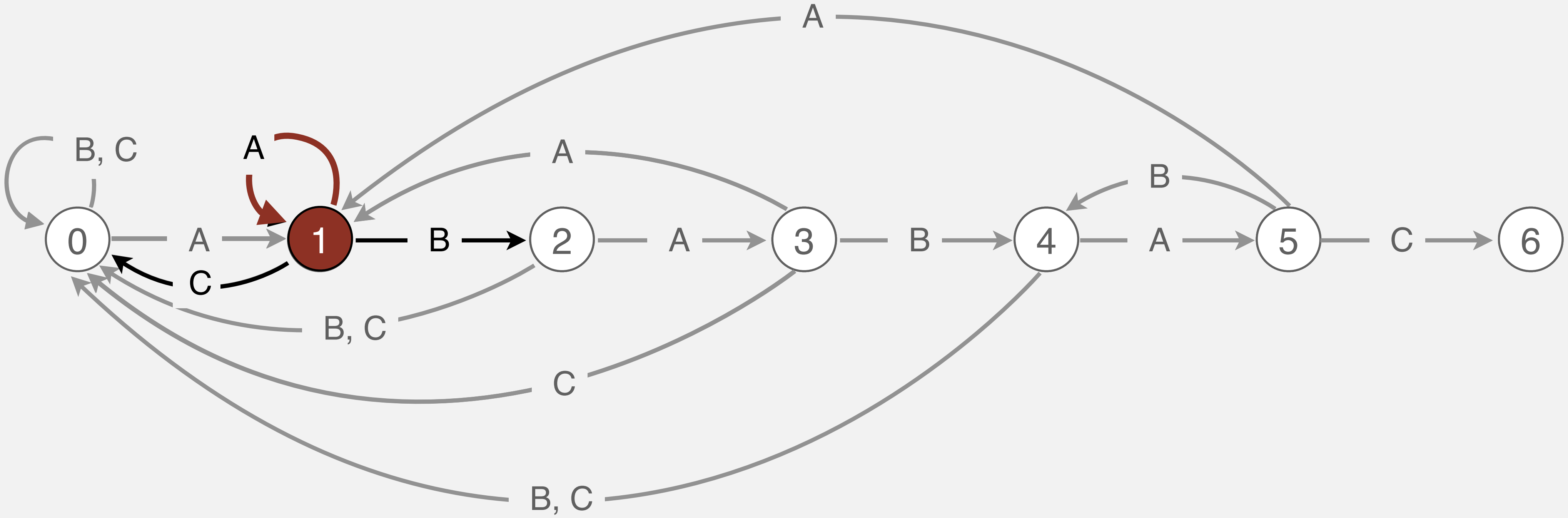
		0	1	2	3	4	5
`pat.charAt(j)`		A	B	A	B	A	C
`dfa[][j]`	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6



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

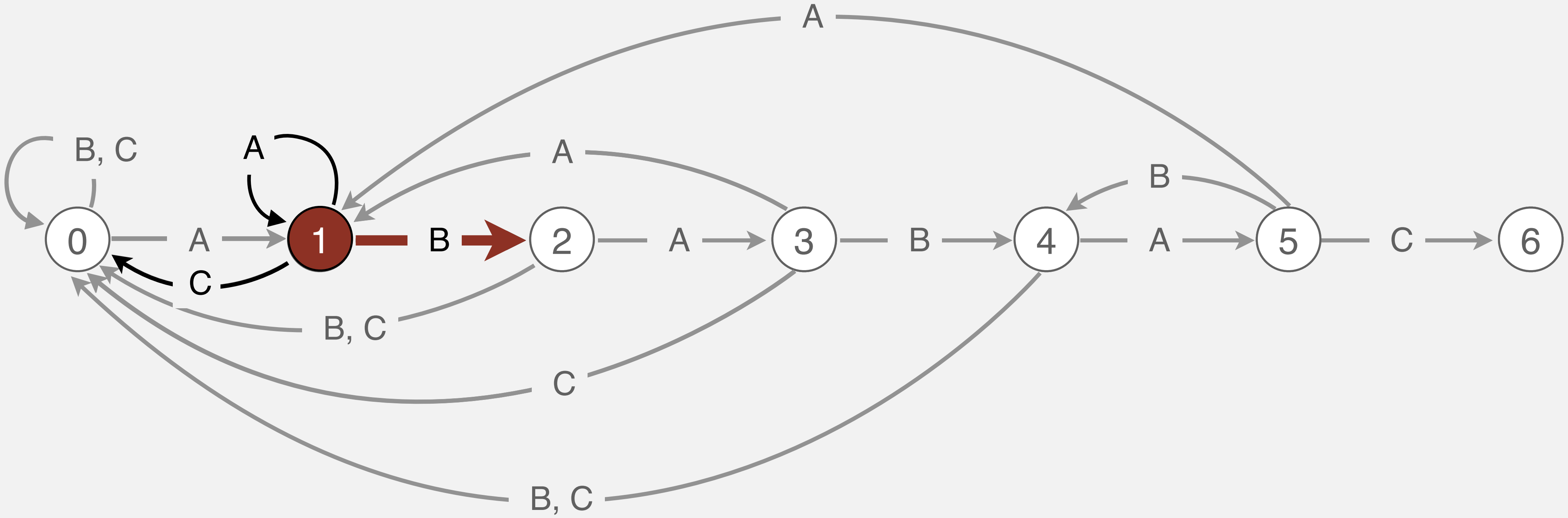


DFA simulation

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

↑

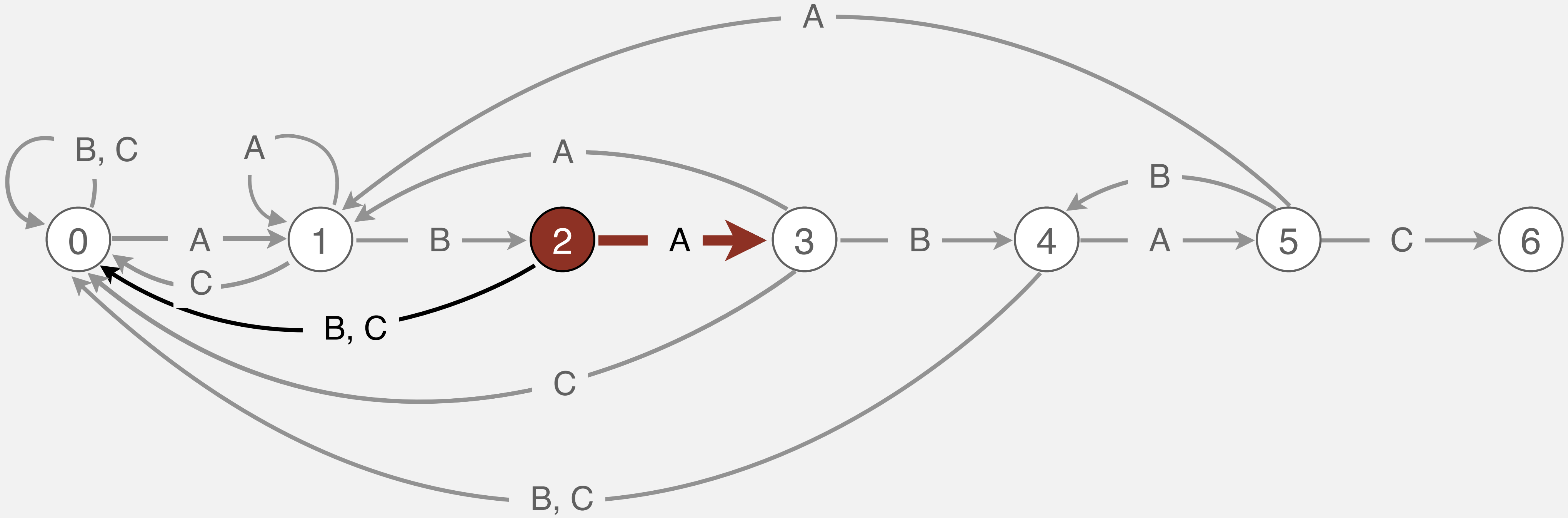
		0	1	2	3	4	5
`pat.charAt(j)`		A	B	A	B	A	C
`dfa[][j]`	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6



DFA simulation

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

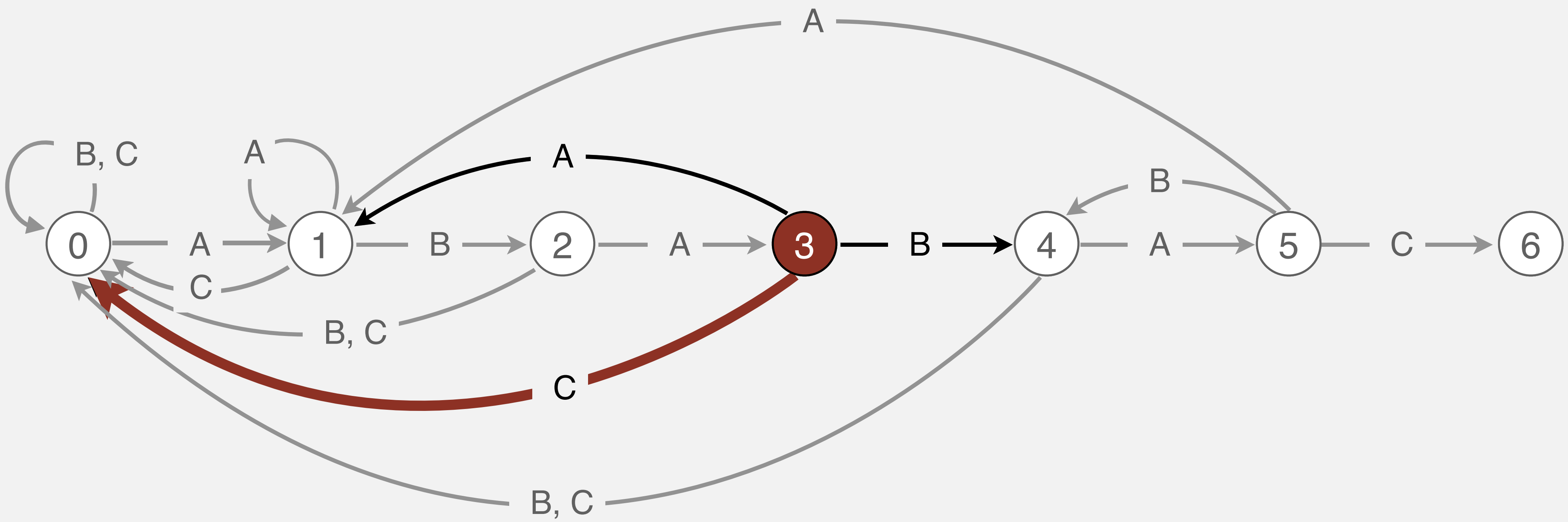
		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6



DFA simulation

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

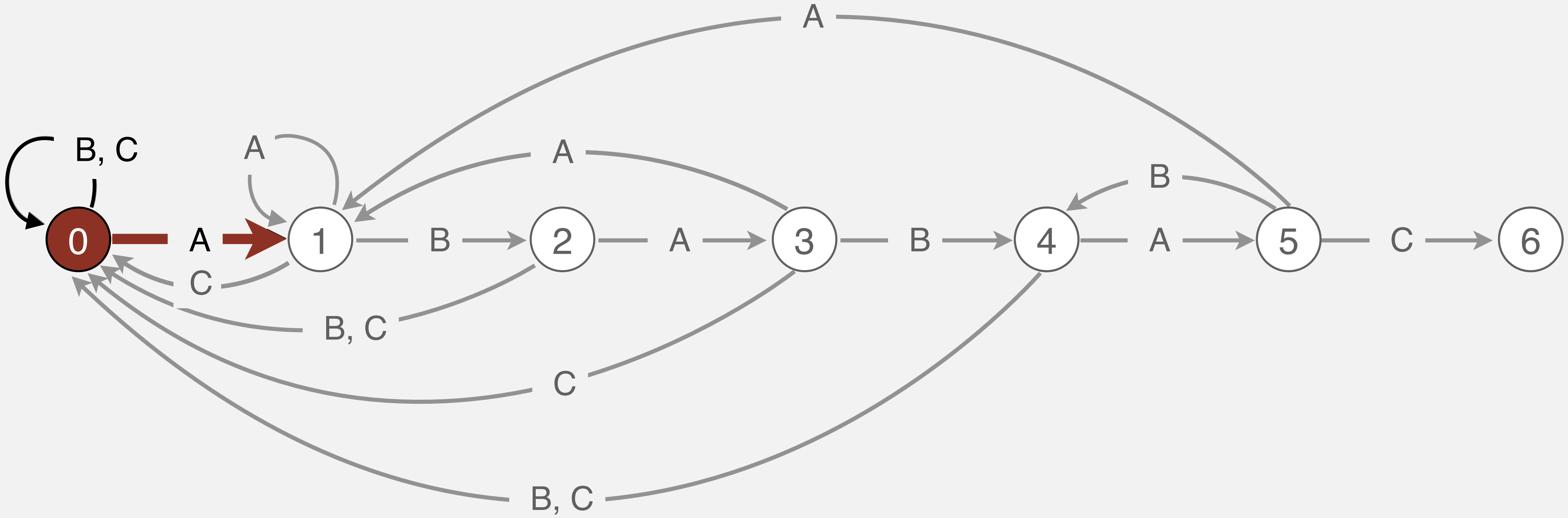
		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6



DFA simulation

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

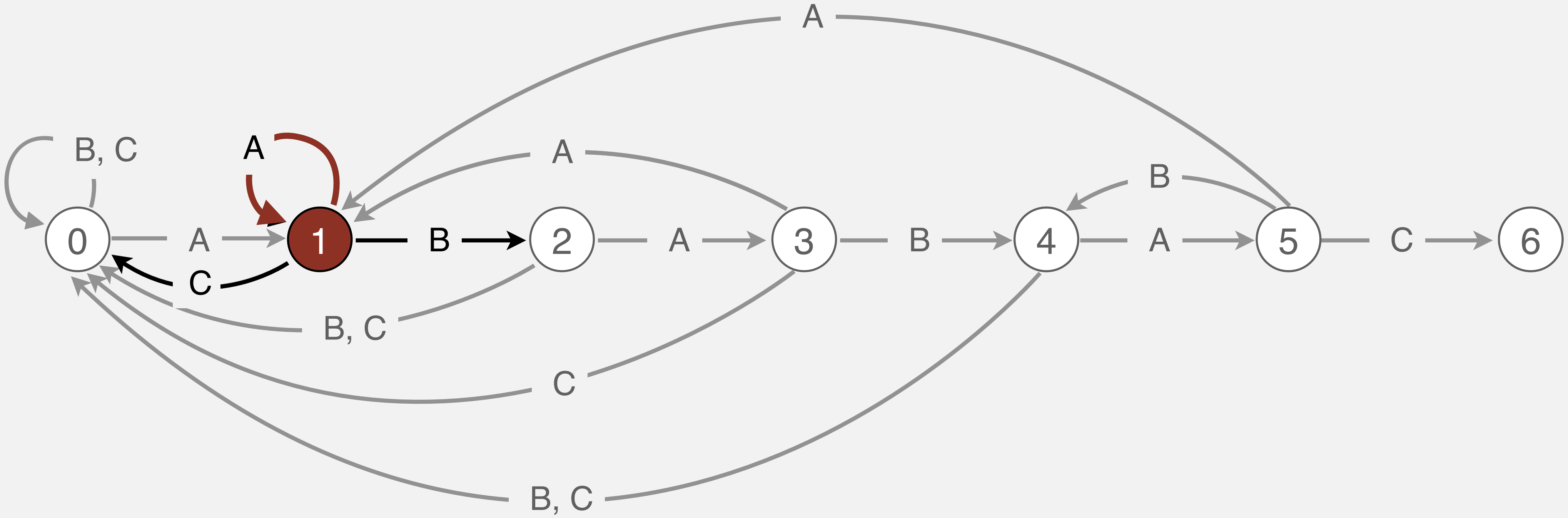
		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6



DFA simulation

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

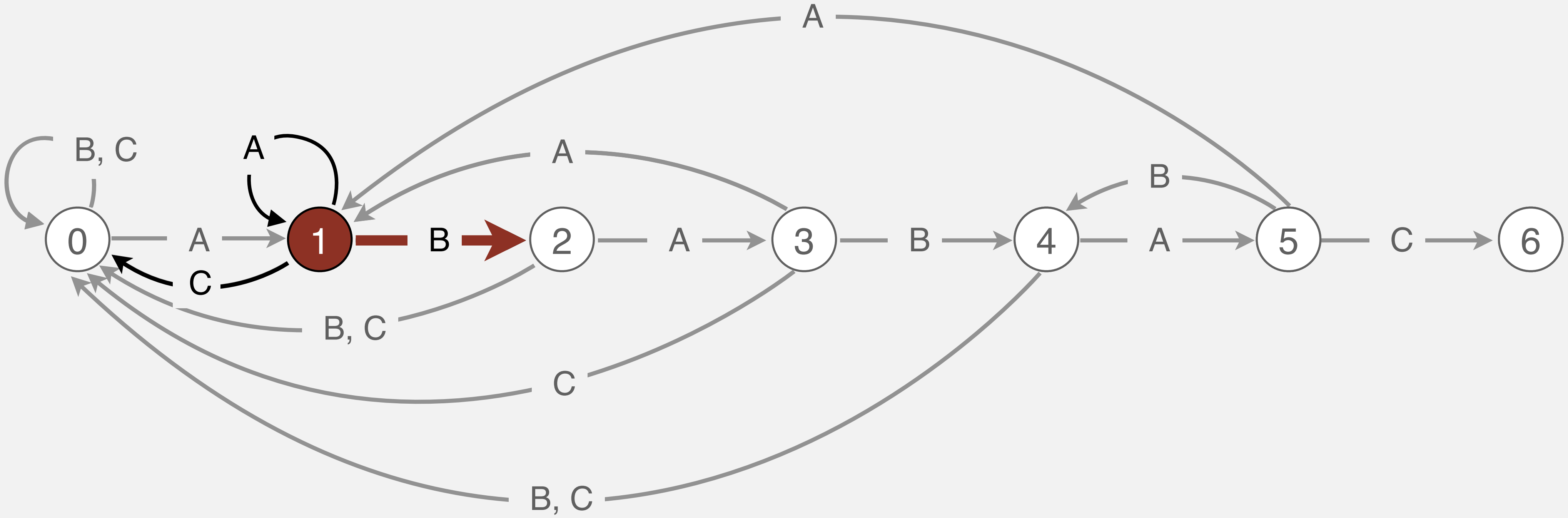
		0	1	2	3	4	5
`pat.charAt(j)`		A	B	A	B	A	C
`dfa[][j]`	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6



DFA simulation

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

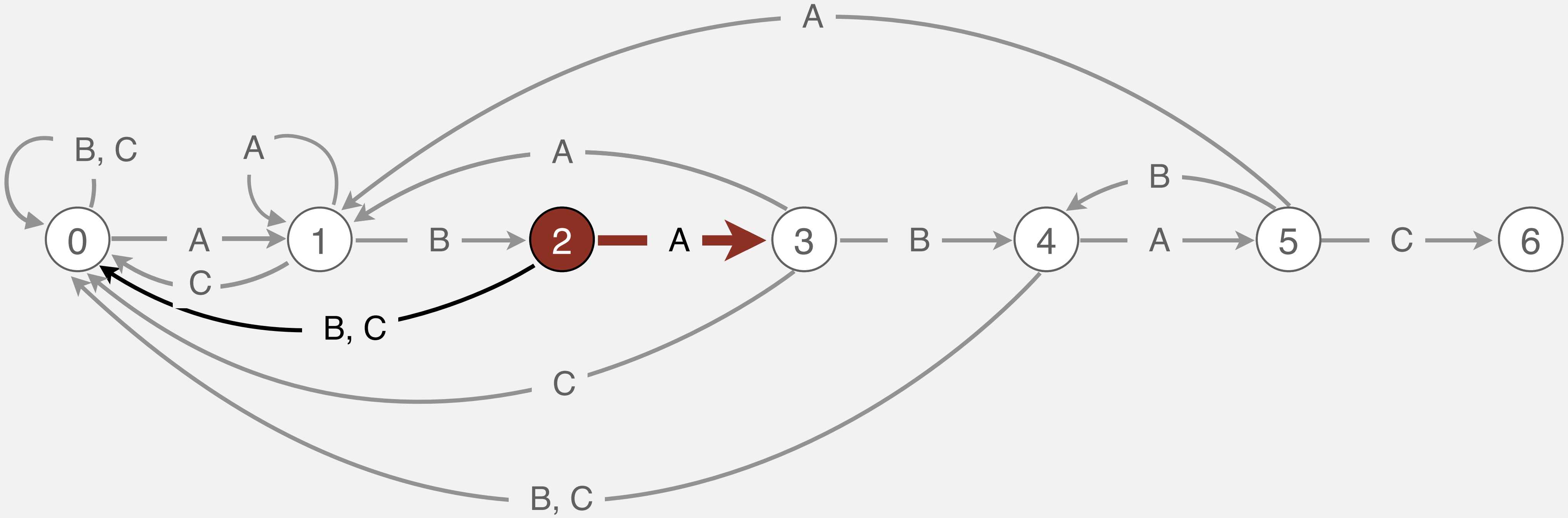
		0	1	2	3	4	5
`pat.charAt(j)`		A	B	A	B	A	C
	A	1	1	3	1	5	1
`dfa[][j]`	B	0	2	0	4	0	4
	C	0	0	0	0	0	6



DFA simulation

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

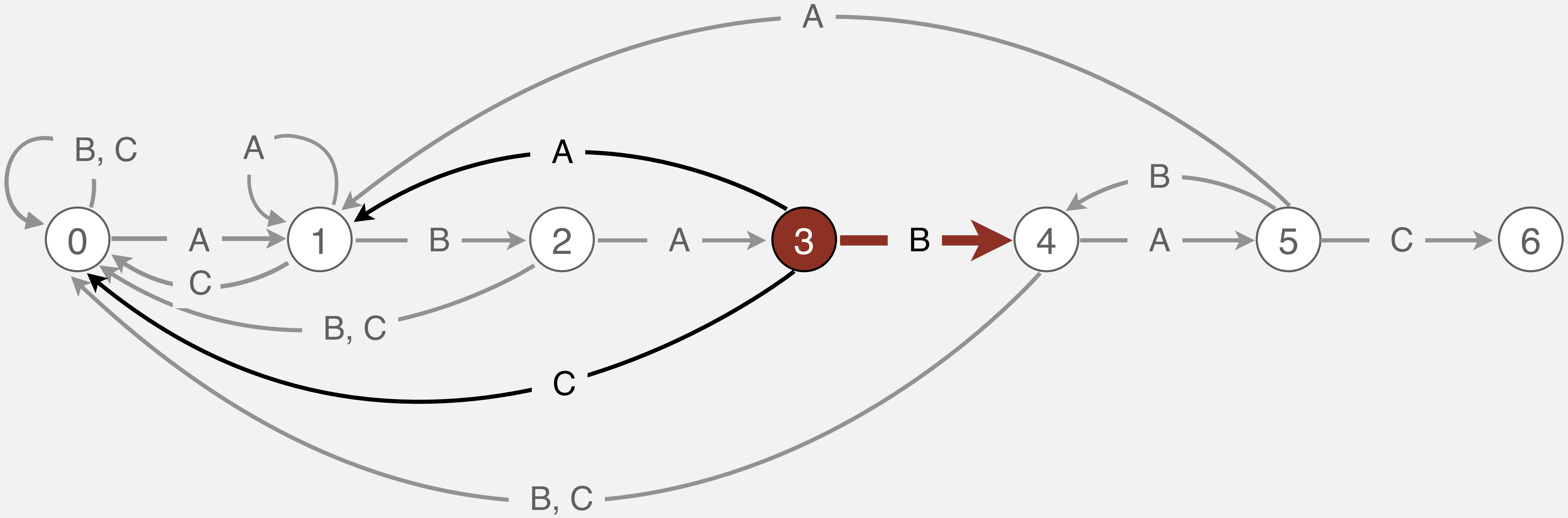
		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6



DFA simulation

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

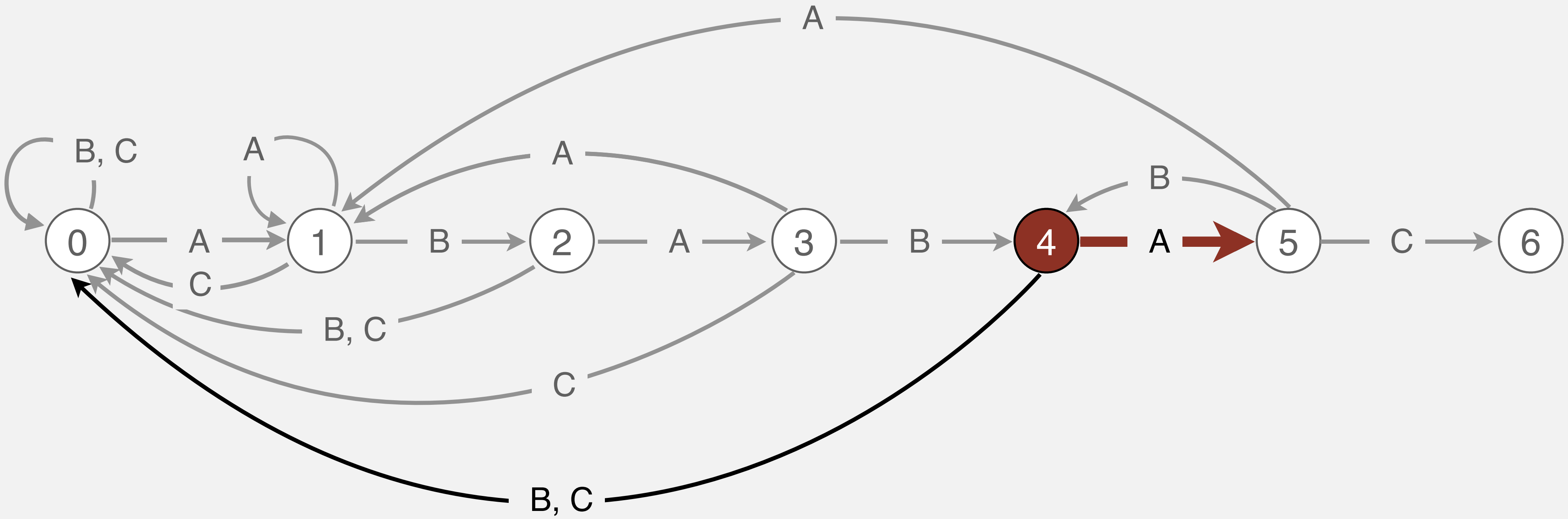
		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6



DFA simulation

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

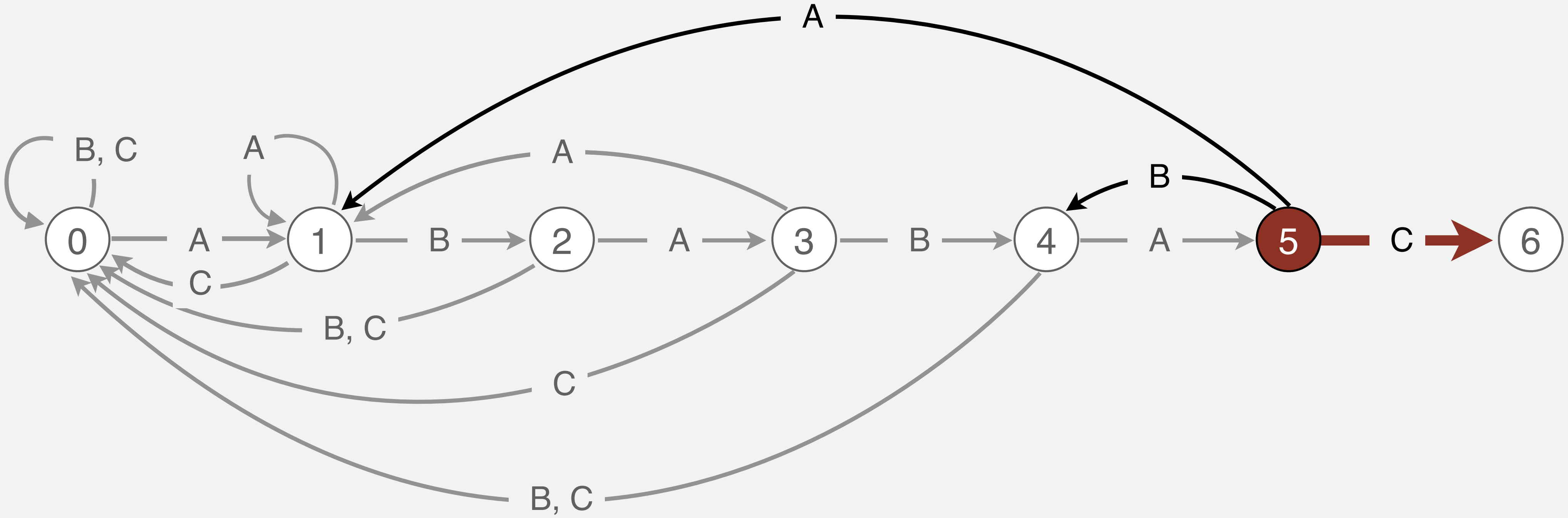
		0	1	2	3	4	5
`pat.charAt(j)`		A	B	A	B	A	C
`dfa[][j]`	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6



DFA simulation

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

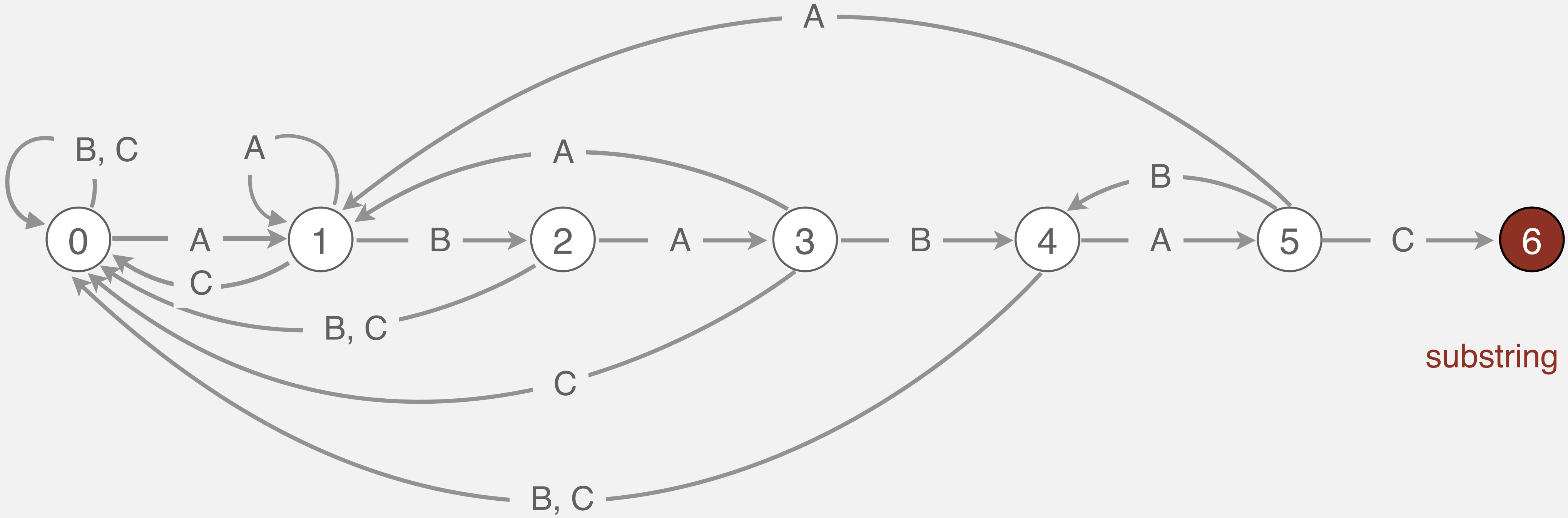
		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6



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

txt

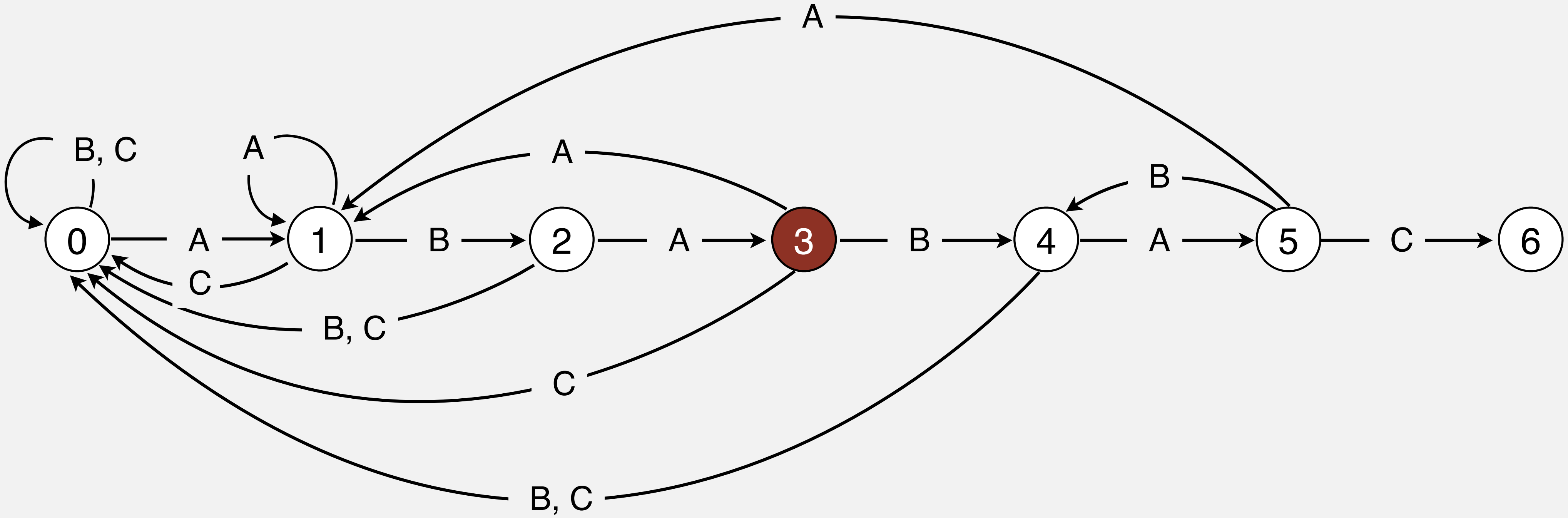
0	1	2	3	4	5	6	7	8
B	C	B	A	A	B	A	C	A

suffix of text[0..6]

pat

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

prefix of pat[]



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

Running time.

- Simulate DFA on text: at most N character accesses.
- Build DFA: how to do efficiently? [warning: tricky algorithm ahead]

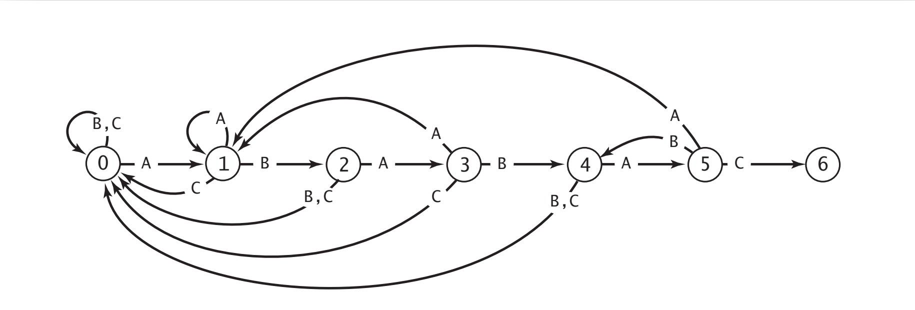
Knuth-Morris-Pratt substring search: Java implementation

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 construction

Include one state for each character in pattern (plus accept state).

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

Constructing the DFA for KMP substring search for A B A B A C

Knuth-Morris-Pratt construction

Match transition. If in state `j` and next char `c == pat.charAt(j)`, go to `j+1`.

first j characters of pattern have already been matched

next char matches

now first j+1 characters of pattern have been matched

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

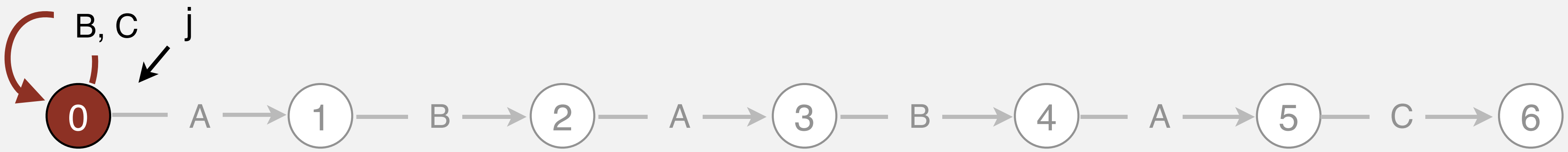
Constructing the DFA for KMP substring search for A B A B A C

Knuth-Morris-Pratt construction

Mismatch transition: back up if `c != pat.charAt(j)`.

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

Constructing the DFA for KMP substring search for A B A B A C

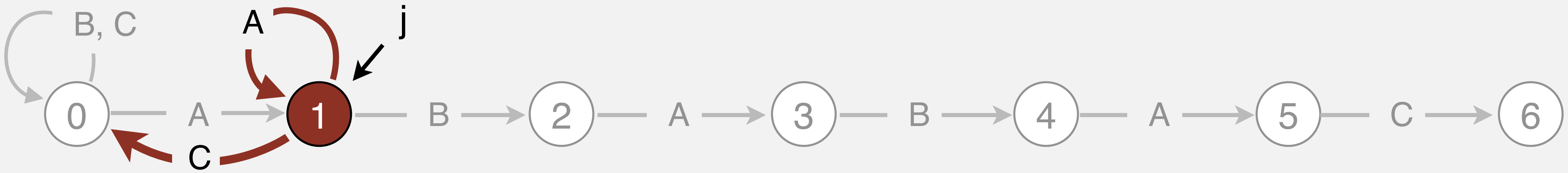


Knuth-Morris-Pratt construction

Mismatch transition: back up if `c != pat.charAt(j)`.

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

Constructing the DFA for KMP substring search for A B A B A C

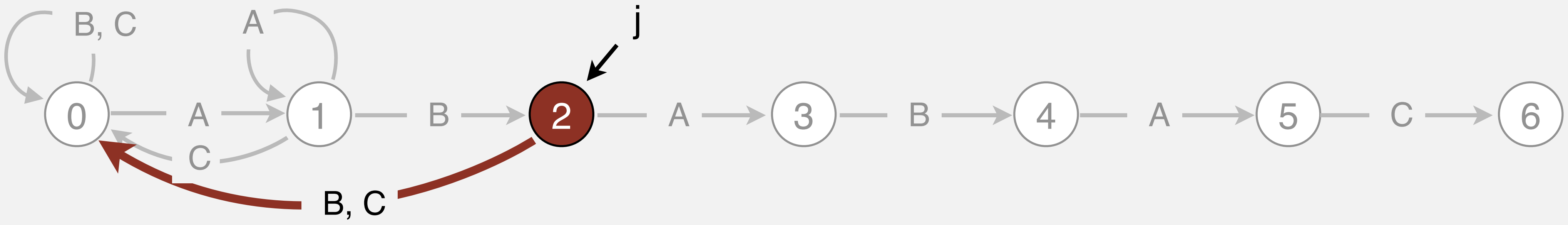


Knuth-Morris-Pratt construction

Mismatch transition: back up if `c != pat.charAt(j)`.

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

Constructing the DFA for KMP substring search for A B A B A C

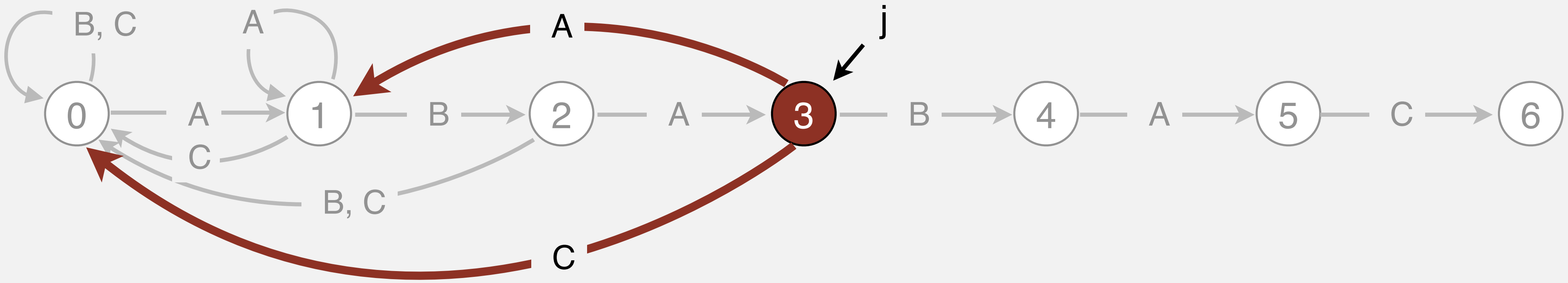


Knuth-Morris-Pratt construction

Mismatch transition: back up if `c != pat.charAt(j)`.

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

Constructing the DFA for KMP substring search for A B A B A C

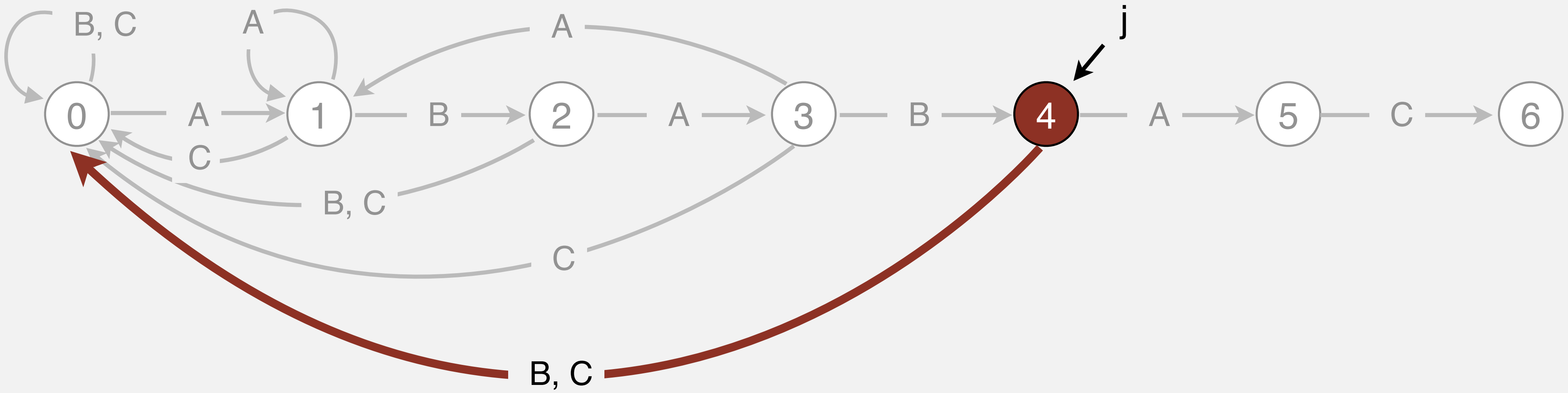


Knuth-Morris-Pratt construction

Mismatch transition: back up if `c != pat.charAt(j)`.

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

Constructing the DFA for KMP substring search for A B A B A C

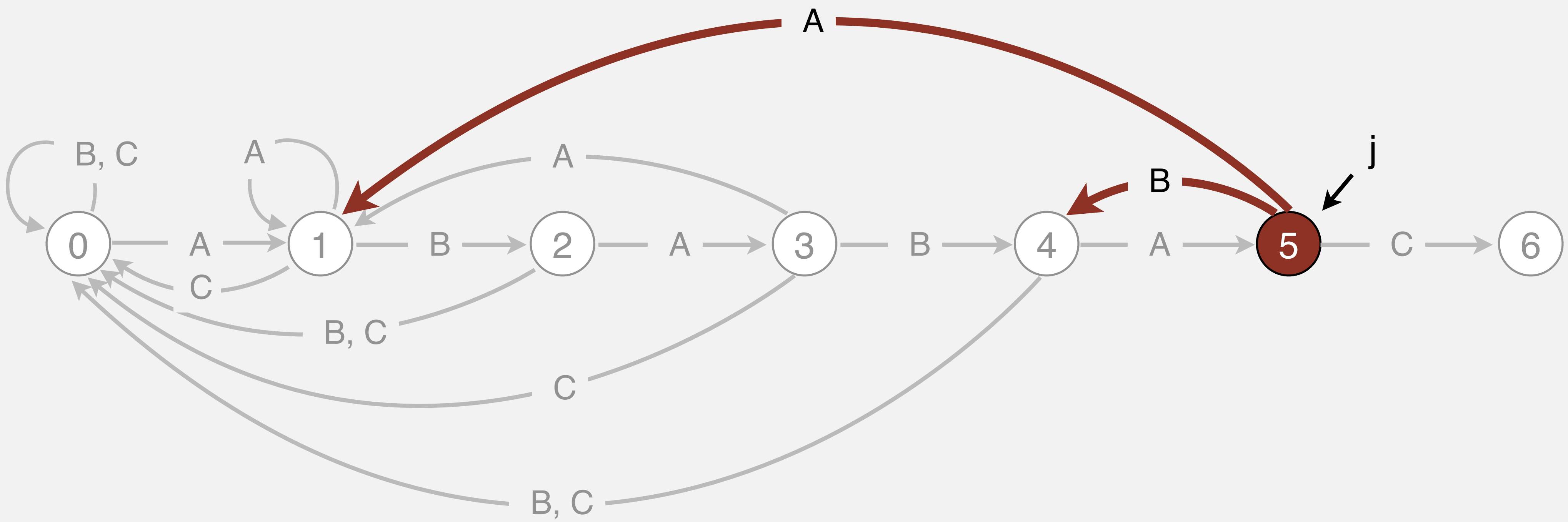


Knuth-Morris-Pratt construction

Mismatch transition: back up if `c != pat.charAt(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

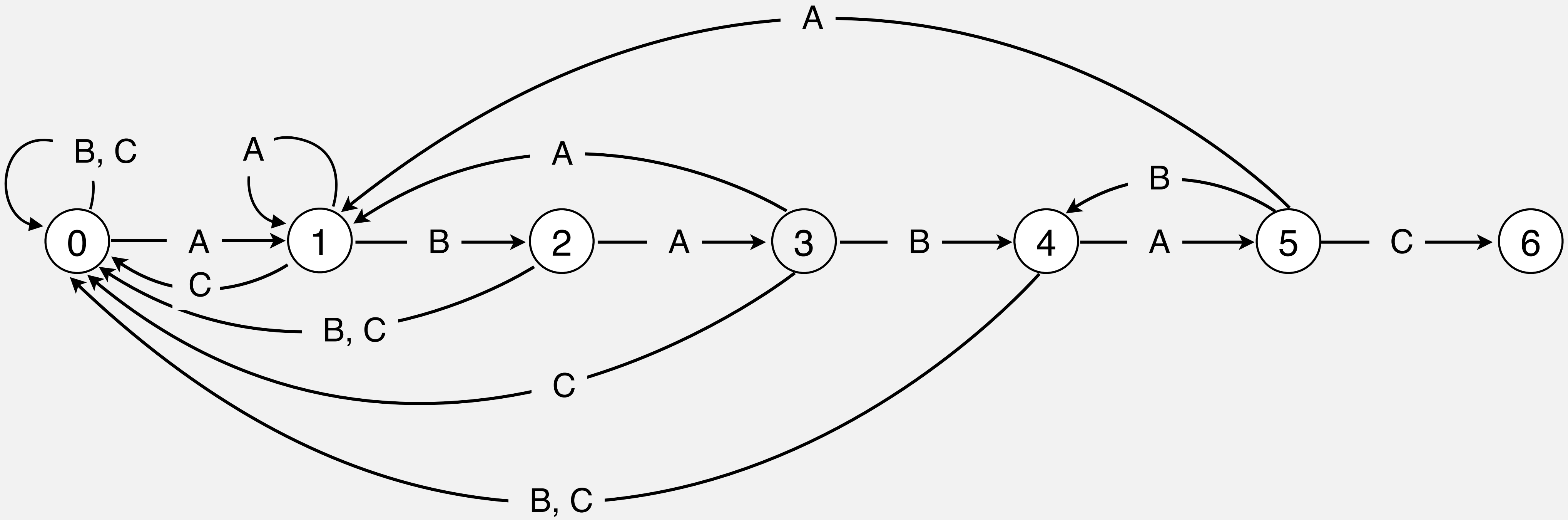
Constructing the DFA for KMP substring search for A B A B A C



Knuth-Morris-Pratt construction

		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

Constructing the DFA for KMP substring search for A B A B A C



How to build DFA from pattern?

Include one state for each character in pattern (plus accept state).

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

How to build DFA from pattern?

Match transition. If in state `j` and next char `c == pat.charAt(j)`, go to `j+1`.

first j characters of pattern have already been matched

next char matches

now first j+1 characters of pattern have been matched

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

How to build DFA from pattern?

Mismatch transition. If in state `j` and next char `c != pat.charAt(j)`, then the last `j-1` characters of input are `pat[1..j-1]`, followed by `c`.

To compute `dfa[c][j]`: Simulate `pat[1..j-1]` on DFA and take transition `c`.

still under construction (!)

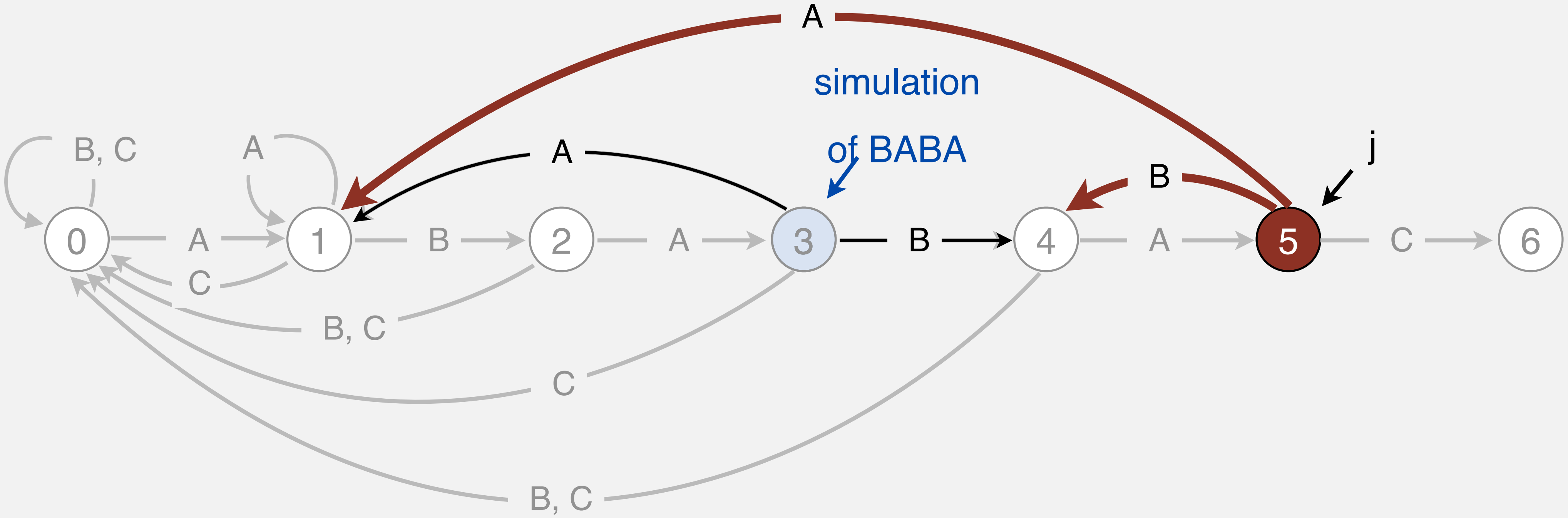
Running time. Seems to require *j* steps.

Ex. `dfa['A'][5] = 1;`   `dfa['B'][5] = 4`

simulate BABA;
take transition 'A'
= dfa['A'][3]

simulate BABA;
take transition 'B'
= dfa['B'][3]

j	0	1	2	3	4	5
`pat.charAt(j)`	A	B	A	B	A	C



How to build DFA from pattern?

Mismatch transition. If in state `j` and next char `c != pat.charAt(j)`, then the last `j-1` characters of input are `pat[1..j-1]`, followed by `c`.

state X

To compute `dfa[c][j]`: Simulate `pat[1..j-1]` on DFA and take transition `c`.

Running time. Takes only constant time if we maintain state X.

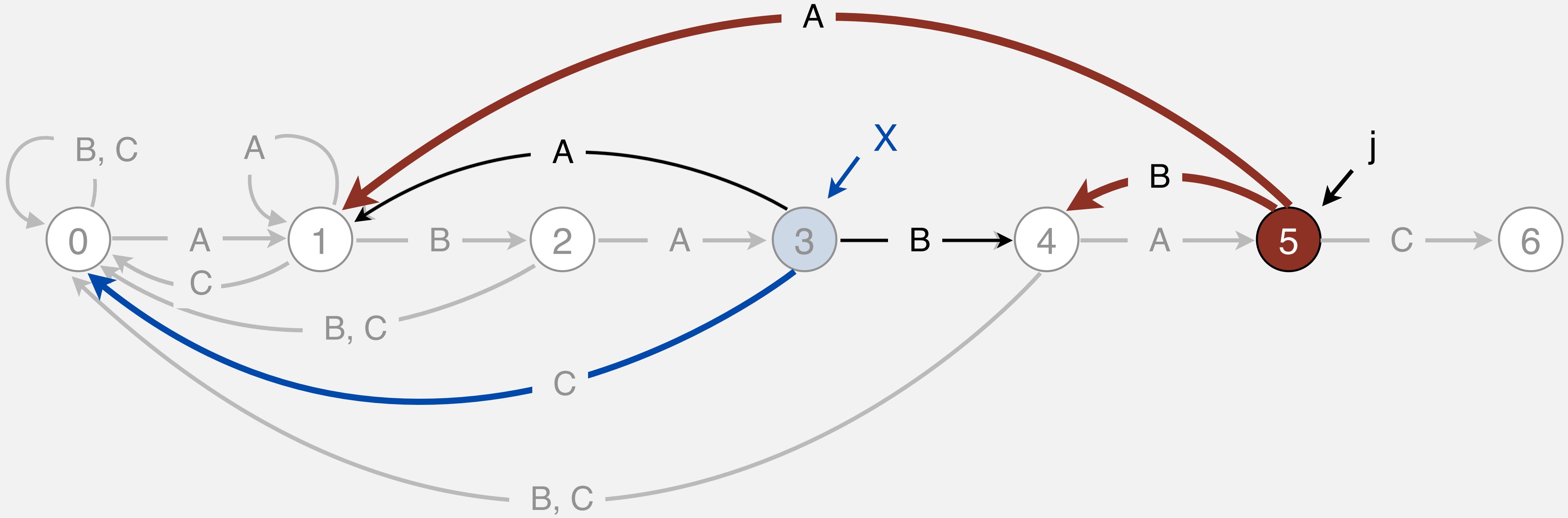
Ex. `dfa['A'][5] = 1;` `dfa['B'][5] = 4;` `X'= 0`

from state X,
take transition 'A'
= dfa['A'][X]

from state X,
take transition 'B'
= dfa['B'][X]

from state X,
take transition 'C'
= dfa['C'][X]

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



Knuth-Morris-Pratt construction (in linear time)

Include one state for each character in pattern (plus accept state).

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

Constructing the DFA for KMP substring search for A B A B A C

Knuth-Morris-Pratt construction (in linear time)

Match transition. For each state `j`, `dfa[pat.charAt(j)][j] = j+1`.

first j characters of pattern have already been matched

now first j+1 characters of pattern have been matched

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

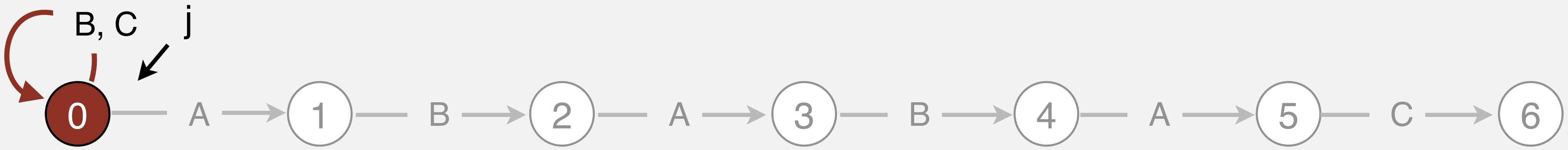
Constructing the DFA for KMP substring search for A B A B A C

Knuth-Morris-Pratt construction (in linear time)

Mismatch transition. For state `0` and char `c != pat.charAt(j)`, set `dfa[c][0] = 0`.

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

Constructing the DFA for KMP substring search for A B A B A C



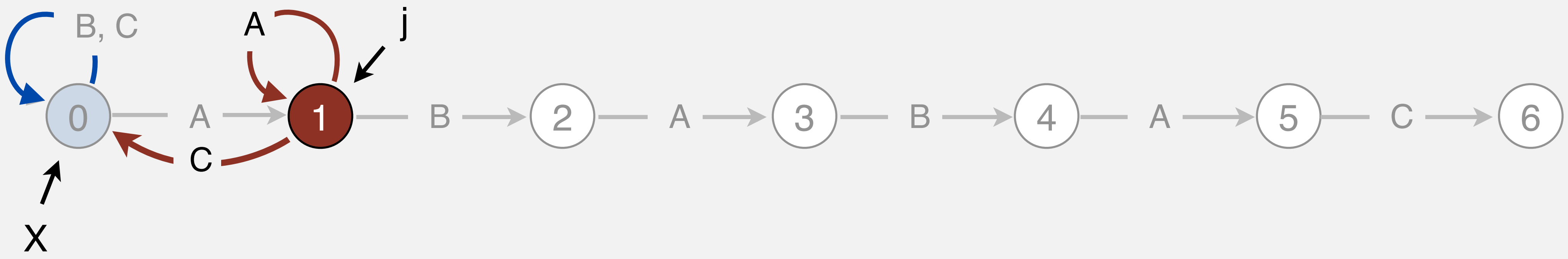
Knuth-Morris-Pratt construction (in linear time)

Mismatch transition. For each state `j` and char `c != pat.charAt(j)`, set `dfa[c][j] = dfa[c][X]`; then update `X = dfa[pat.charAt(j)][X]`.

X = simulation of empty string

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

Constructing the DFA for KMP substring search for A B A B A C



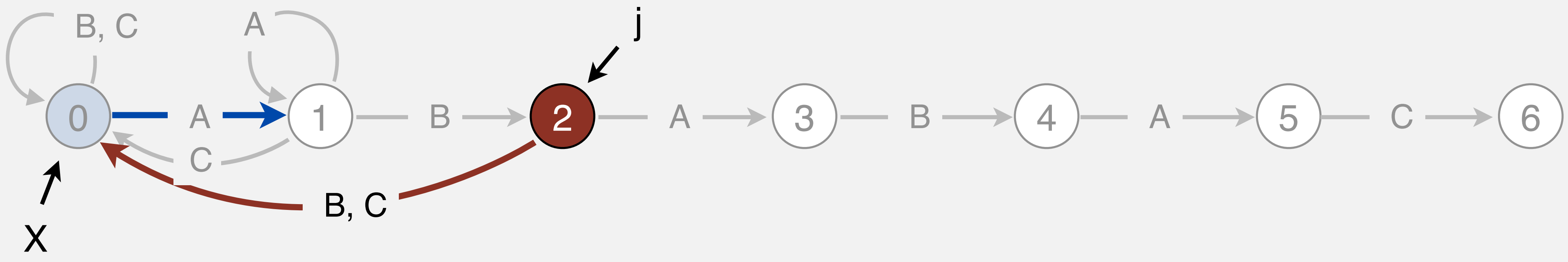
Knuth-Morris-Pratt construction (in linear time)

Mismatch transition. For each state `j` and char `c != pat.charAt(j)`, set `dfa[c][j] = dfa[c][X]`; then update `X = dfa[pat.charAt(j)][X]`.

X = simulation of B

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

Constructing the DFA for KMP substring search for A B A B A C



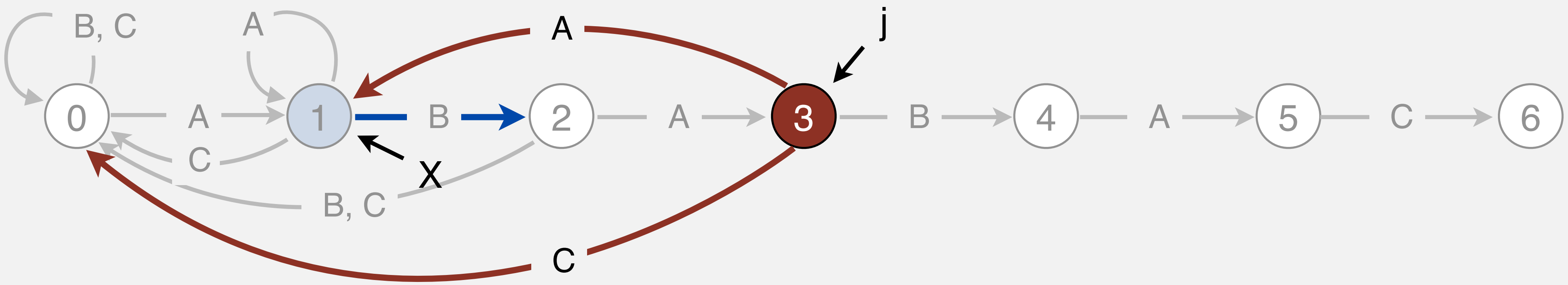
Knuth-Morris-Pratt construction (in linear time)

Mismatch transition. For each state `j` and char `c != pat.charAt(j)`, set `dfa[c][j] = dfa[c][X]`; then update `X = dfa[pat.charAt(j)][X]`.

X = simulation of B A

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

Constructing the DFA for KMP substring search for A B A B A C



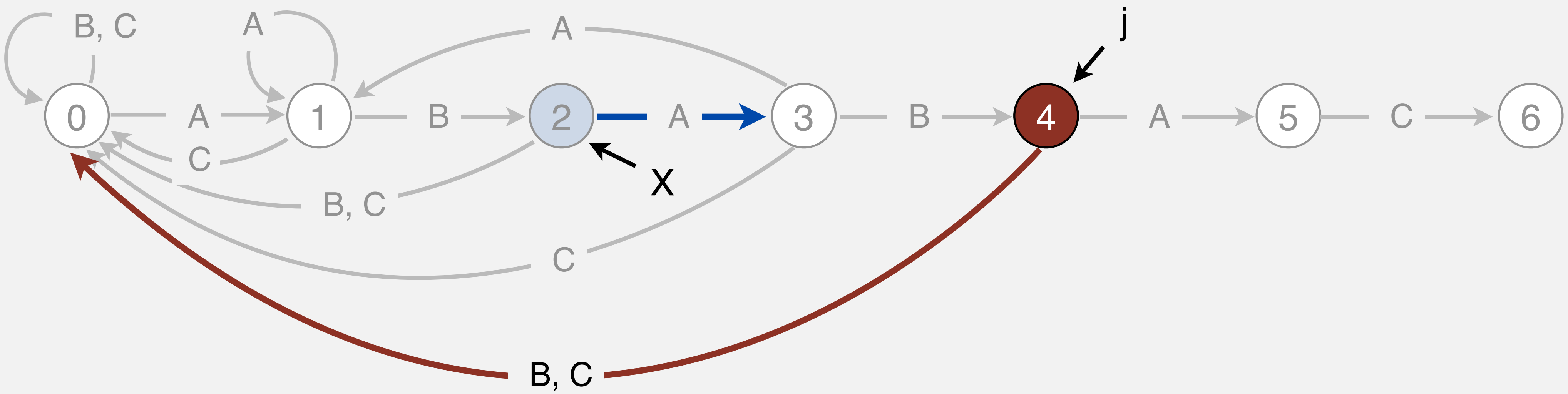
Knuth-Morris-Pratt construction (in linear time)

Mismatch transition. For each state `j` and char `c != pat.charAt(j)`, set `dfa[c][j] = dfa[c][X]`; then update `x = dfa[pat.charAt(j)][X]`.

X = simulation of B A B

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

Constructing the DFA for KMP substring search for A B A B A C



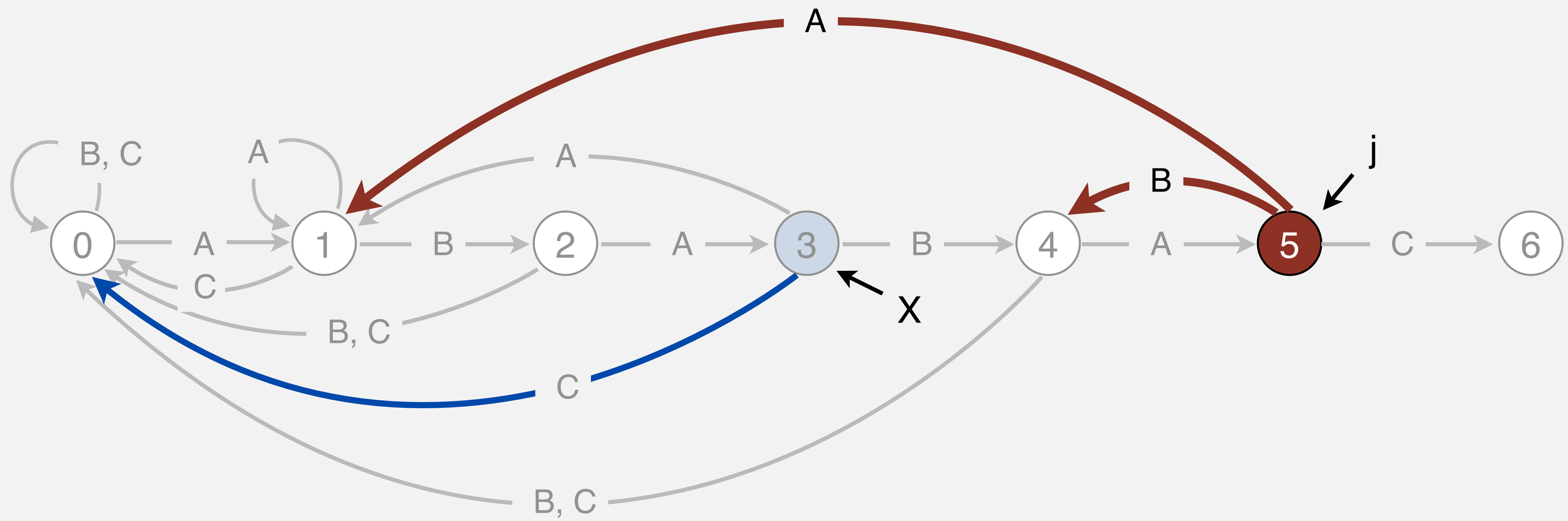
Knuth-Morris-Pratt construction (in linear time)

Mismatch transition. For each state `j` and char `c != pat.charAt(j)`, set `dfa[c][j] = dfa[c][X]`; then update `X = dfa[pat.charAt(j)][X]`.

X = simulation of B A B A

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

Constructing the DFA for KMP substring search for A B A B A C



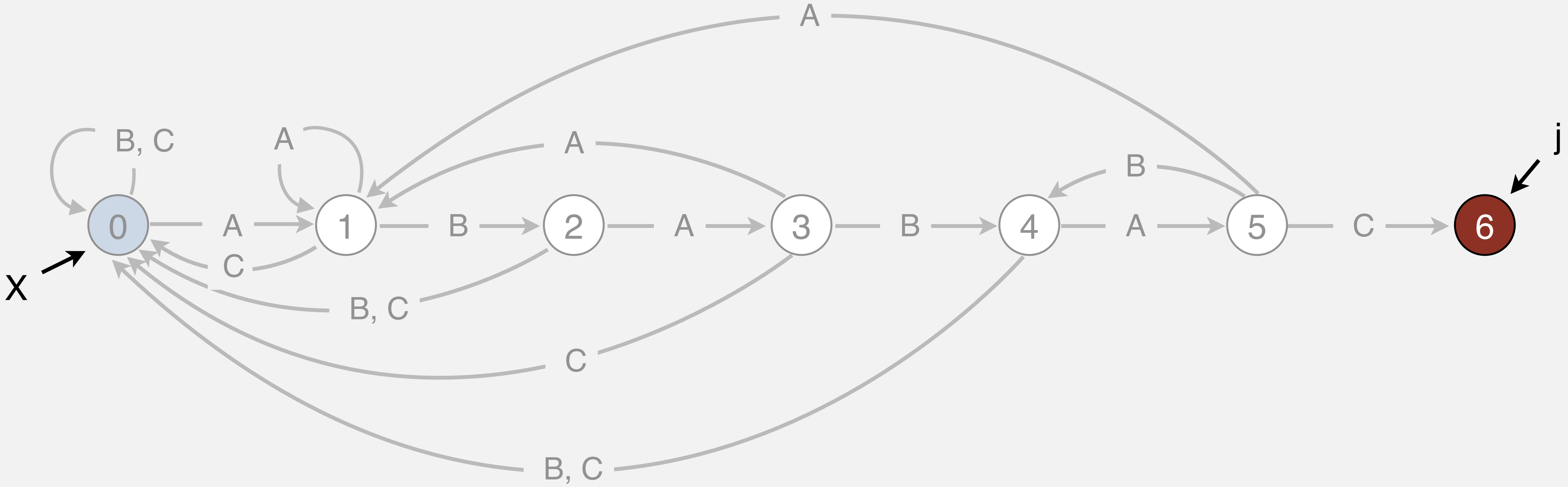
Knuth-Morris-Pratt construction (in linear time)

Mismatch transition. For each state `j` and char `c != pat.charAt(j)`, set `dfa[c][j] = dfa[c][X]`; then update `X = dfa[pat.charAt(j)][X]`.

X = simulation of B A B A C

		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

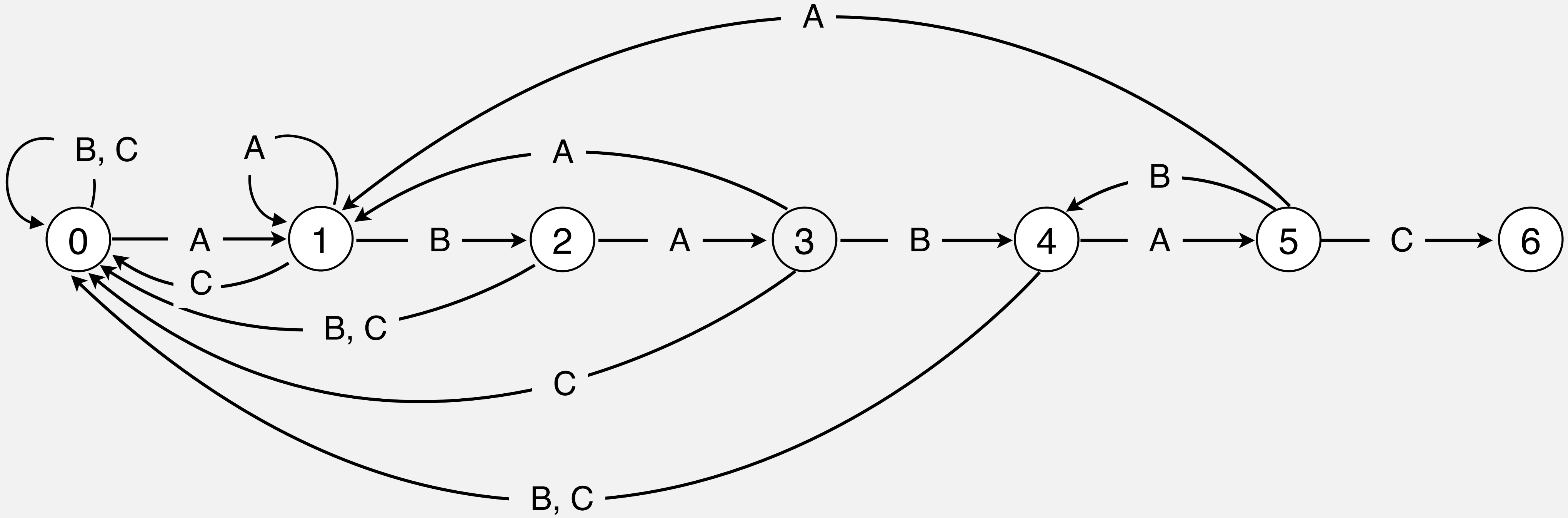
Constructing the DFA for KMP substring search for A B A B A C



Knuth-Morris-Pratt construction (in linear time)

		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

Constructing the DFA for KMP substring search for A B A B A C



Constructing the DFA for KMP substring search: Java implementation

For each state `j`:

- Copy `dfa[][X]` to `dfa[][j]` for mismatch case.
- Set `dfa[pat.charAt(j)][j]` to `j+1` for match case.
- Update `X`.

```
public KMP(String pat)
{
   this.pat = pat;
   M = pat.length();
   dfa = new int[R][M];
   dfa[pat.charAt(0)][0] = 1;
   for (int X = 0, j = 1; j < M; j++)
   {
      for (int c = 0; c < R; c++)
         dfa[c][j] = dfa[c][X];
      dfa[pat.charAt(j)][j] = j+1;
      X = dfa[pat.charAt(j)][X];
   }
}
```

copy mismatch cases

set match case

update restart state

Running time. M character accesses (but space proportional to $R\ M$).

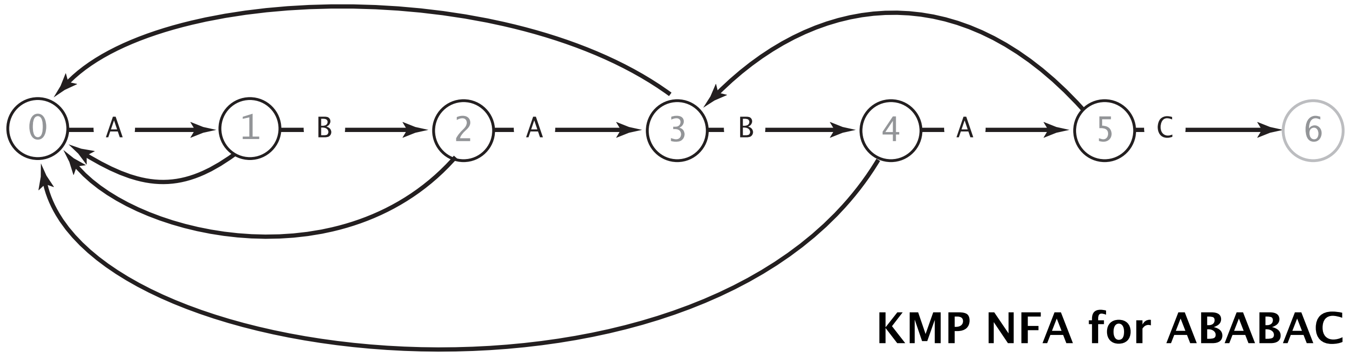
KMP substring search analysis

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 char accessed once when constructing the DFA; each text char accessed once (in the worst case) when simulating the DFA.

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

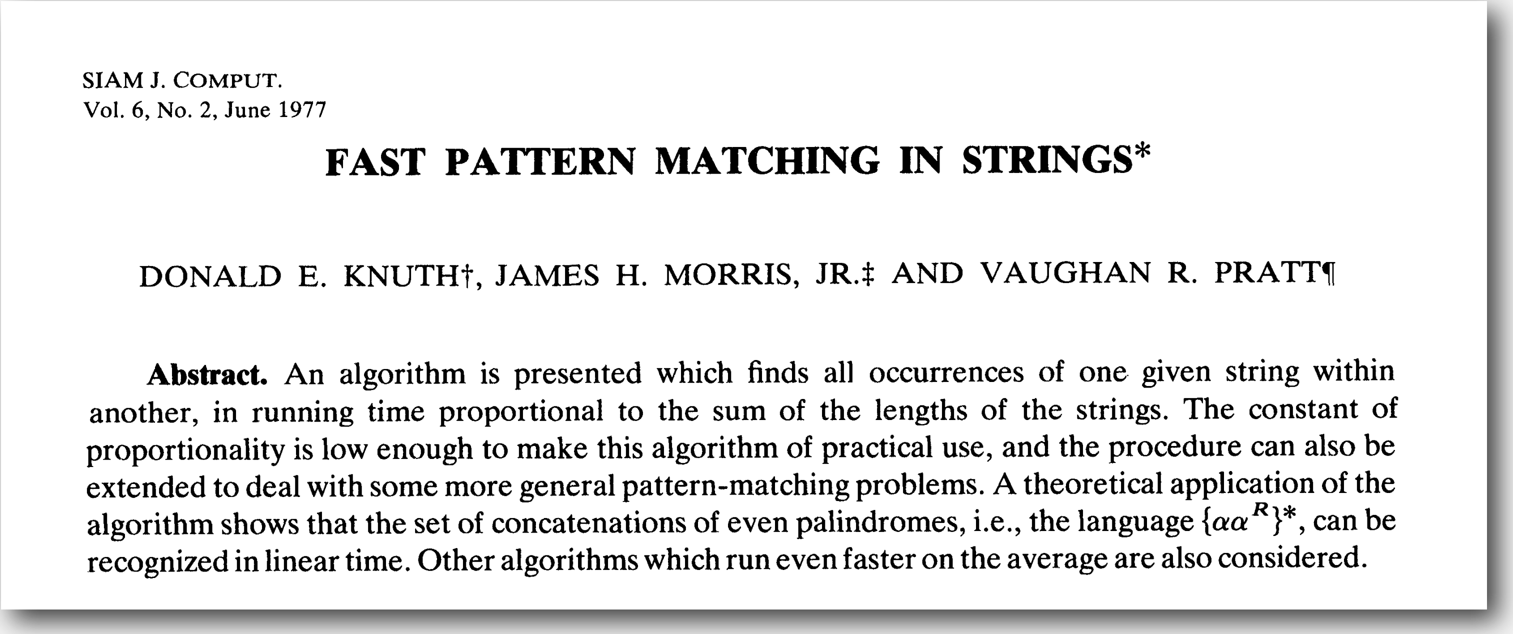
Larger alphabets. Improved version of KMP constructs `nfa[]` in time and space proportional to M.



KMP NFA for ABABAC

Knuth-Morris-Pratt: brief history

- Independently discovered by two theoreticians and a hacker.
  - Knuth: inspired by esoteric theorem, discovered linear-time 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

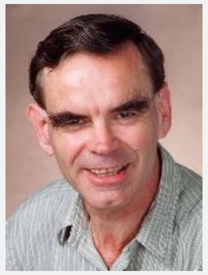
FAST PATTERN MATCHING IN STRINGS*

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

SUBSTRING SEARCH

- Brute force
- Knuth-Morris-Pratt
- **Boyer-Moore**
- Rabin-Karp

Boyer Moore Intuition

- Scan the text with a window of M chars (length of pattern)

Pattern in Text (M)

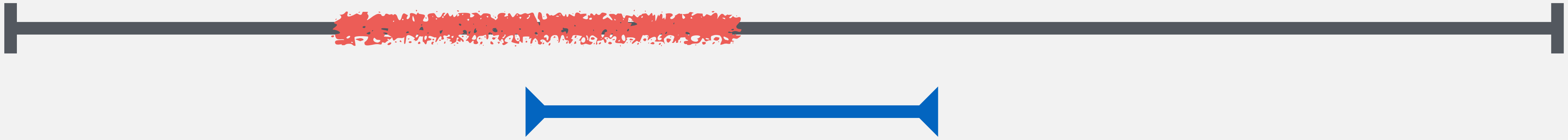
Text

Scan Window (M)

- Case 1: Scan Window is exactly on top of the searched pattern

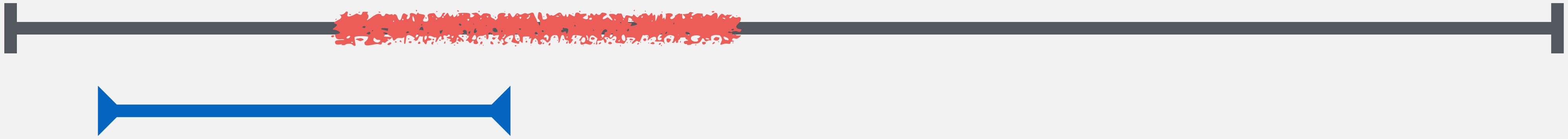
  - Starting from one end check if all characters are equal. (We must check!)

- Case 2: Scan Window starts after the pattern starts.

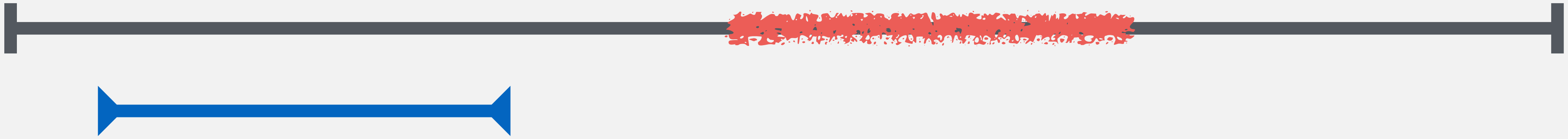


Boyer Moore Intuition (2)

- Case 3: Scan Window starts before the pattern starts



- Case 4: Independent



- In case 4, simply shift window M characters
- Avoid Case 2
- Convert Case 3 to Case 1, by shifting appropriately

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.
  - First we check the character in index pattern.length()-1
  - It is N which is not E, so we know that first 5 characters is not a match. Shift text 5 characters
  - S != E so shift 5, E == E so we can check for the pattern.length()-2, L!=N, skip 4.

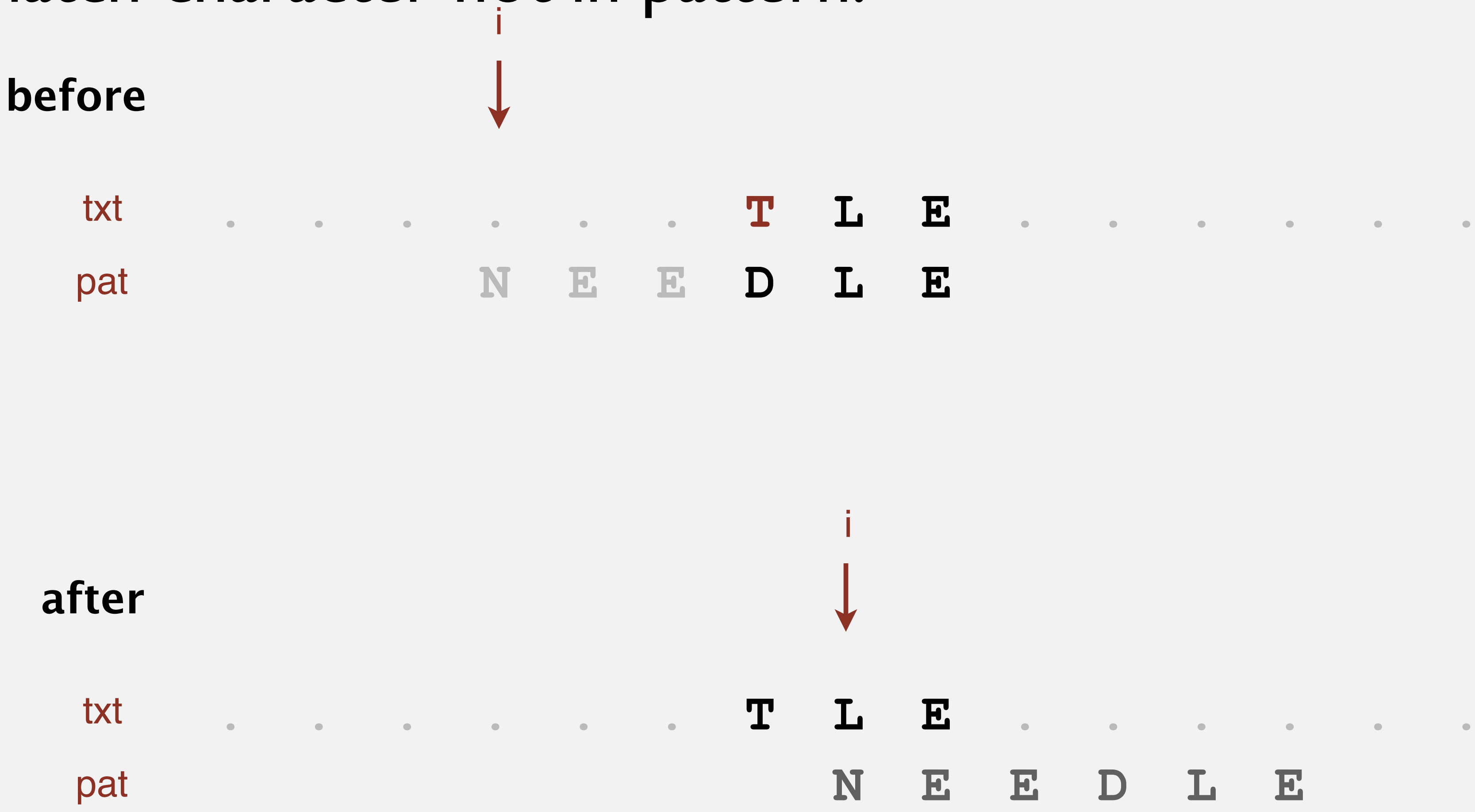
i	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
text →		F	I	N	D	I	N	A	H	A	Y	S	T	A	C	K	N	E	E	D	L	E	I	N	A
0	5	N	E	E	D	L	E ← *pattern*																		
5	5						N	E	E	D	L	E													
11	4												N	E	E	D	L	E							
15	0																N	E	E	D	L	E			

return i = 15

Boyer-Moore: mismatched character heuristic

Q. How much to skip?

Case 1. Mismatch character not in pattern.

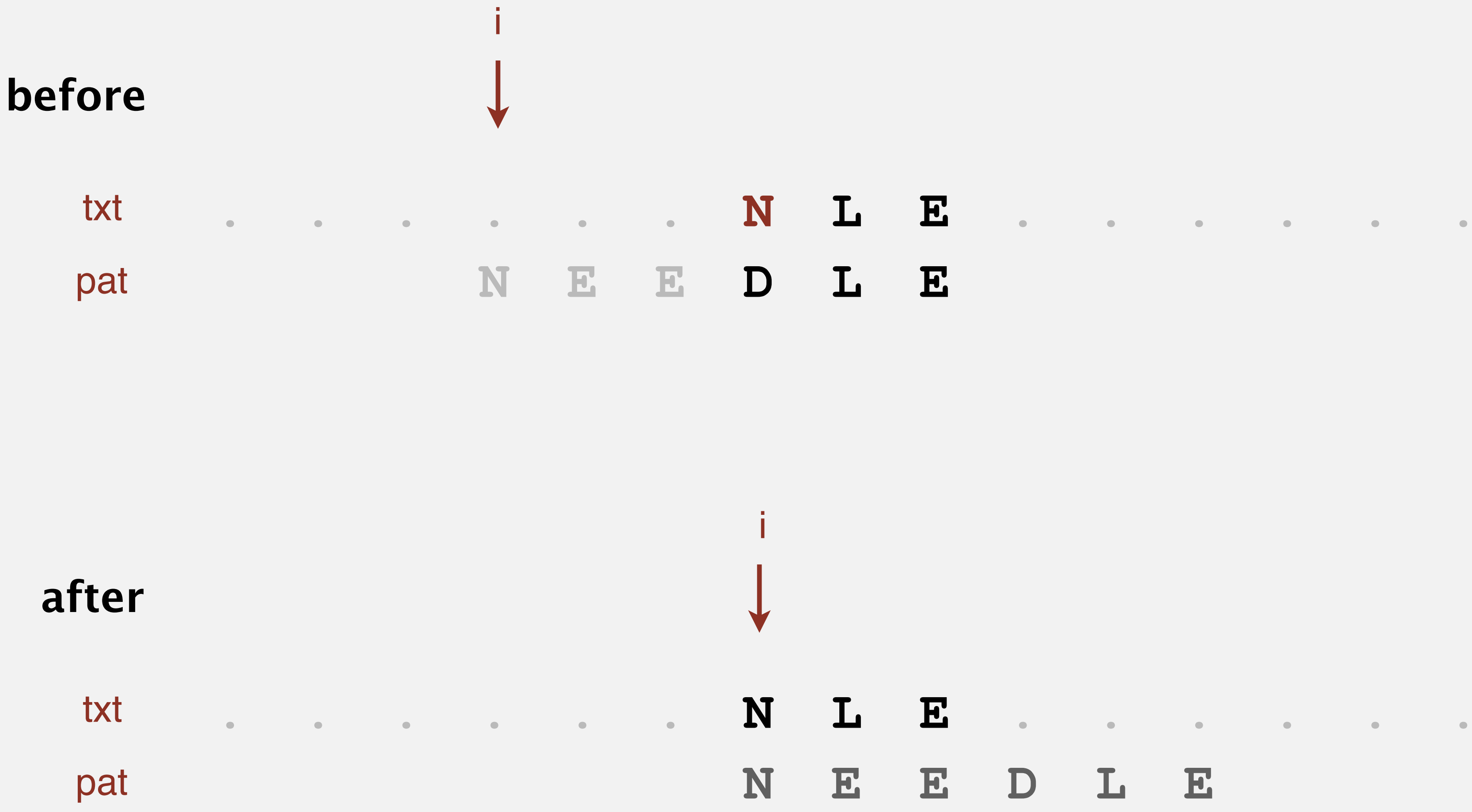


mismatch character 'T' not in pattern: increment i one character beyond 'T'

Boyer-Moore: mismatched character heuristic

Q. How much to skip?

Case 2a. Mismatch character in pattern.



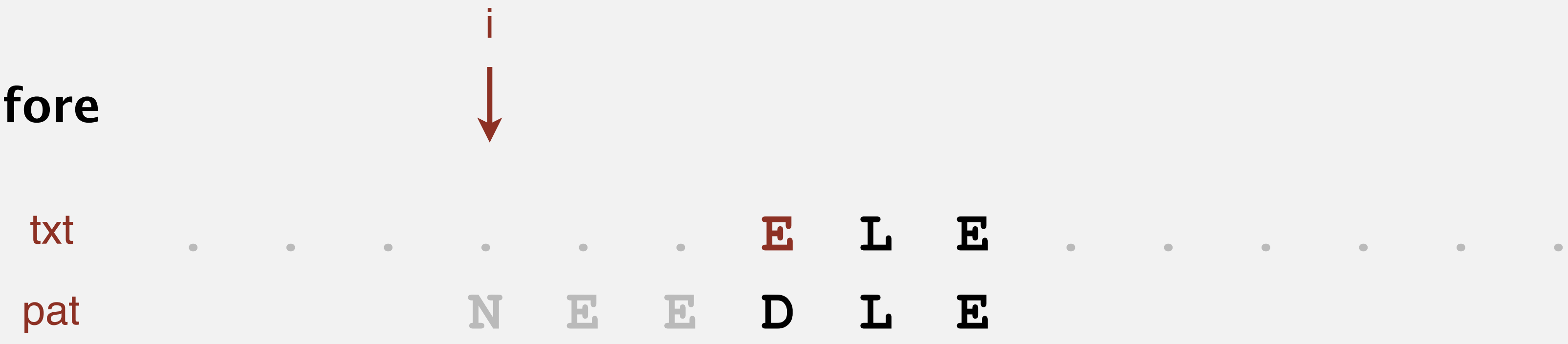
mismatch character 'N' in pattern: align text 'N' with rightmost pattern 'N'

Boyer-Moore: mismatched character heuristic

Q. How much to skip?

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

before



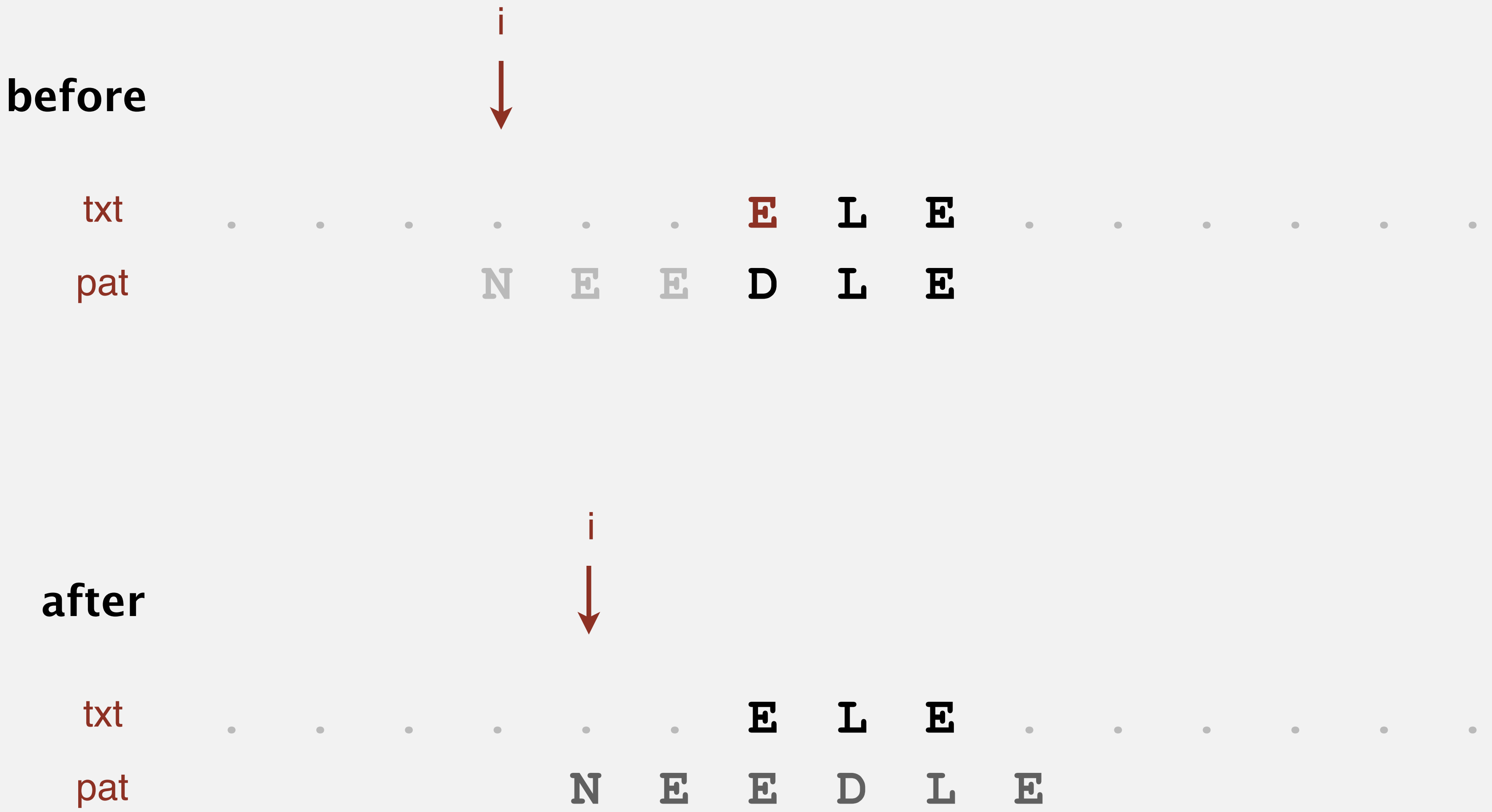
aligned with rightmost E?

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).



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).

```
right = new int[R];
for (int c = 0; c < R; c++)
   right[c] = -1;
for (int j = 0; j < M; j++)
   right[pat.charAt(j)] = j;
```

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

Boyer-Moore skip table computation

Boyer-Moore: Java implementation

```
public int search(String txt)
{
    int N = txt.length();
    int M = pat.length();
    int skip;
    for (int i = 0; i <= N-M; i += skip)
    {
        skip = 0;
        for (int j = M-1; j >= 0; j--)
        {
            if (pat.charAt(j) != txt.charAt(i+j))
            {
                skip = Math.max(1, j - right[txt.charAt(i+j)]);
                break;
            }
        }
        if (skip == 0) return i;
    }
    return N;
}
```

compute skip value

in case other term is nonpositive

match

Another Example

SEARCH FOR: XXXX

A X A X A X A X X X A X A X X X X A A A

If the window scan points to an unrecognised character, we can skip past that character. For this example, for the initial step we first match X at the end, when check for previous character (A) which is not in the string we skip 3 steps. The X at the end, we matched can still be the first character of the pattern, so we do not skip that.

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.

sublinear!

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

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 $\sim 3\,N$ by adding a KMP-like rule to guard against repetitive patterns.

SUBSTRING SEARCH

- Brute force
- Knuth-Morris-Pratt
- Boyer-Moore
- **Rabin-Karp**

Rabin-Karp fingerprint search

Basic idea = modular hashing.

- Compute a hash of pattern characters 0 to M - 1.
- For each i, compute a hash of text characters i to $M + i$ - 1.
- 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
                                                                match
 5                           9   2   6   5   3   % 997 = 929     ↙
 6 ←— return i = 6               2   6   5   3   5   % 997 = 613
```

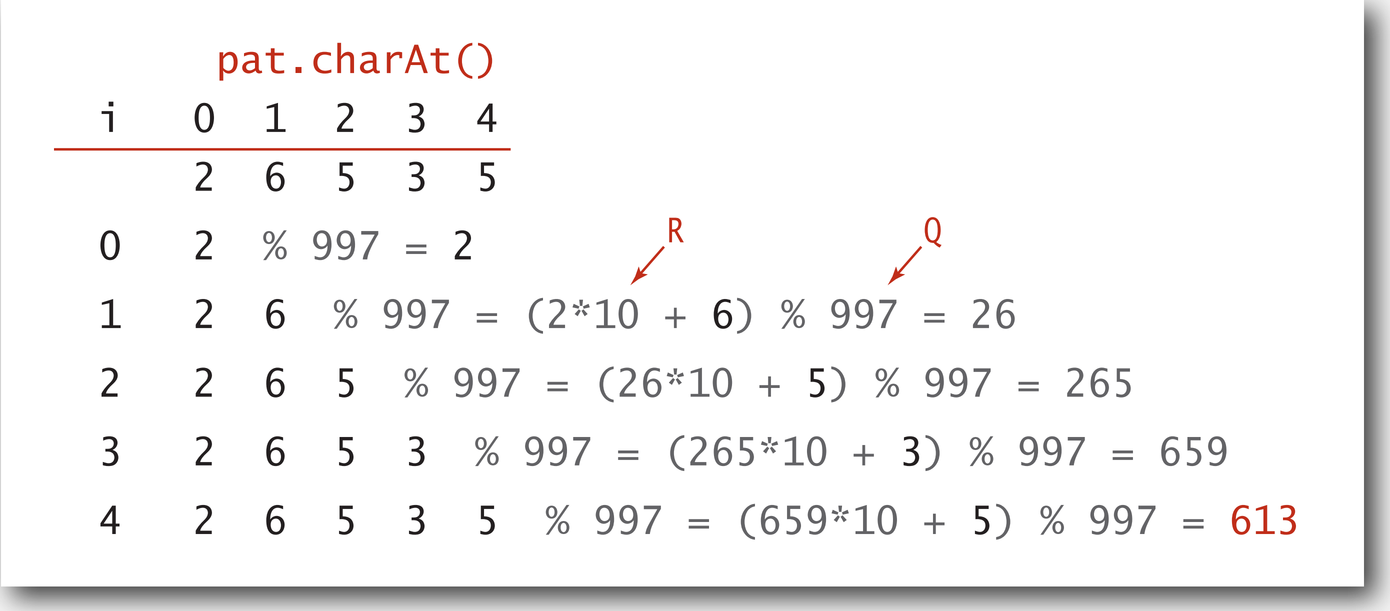
Efficiently computing the hash function

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

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

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
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 = (R * h + key.charAt(j)) % Q;
   return h;
}
```

Efficiently computing the hash function

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

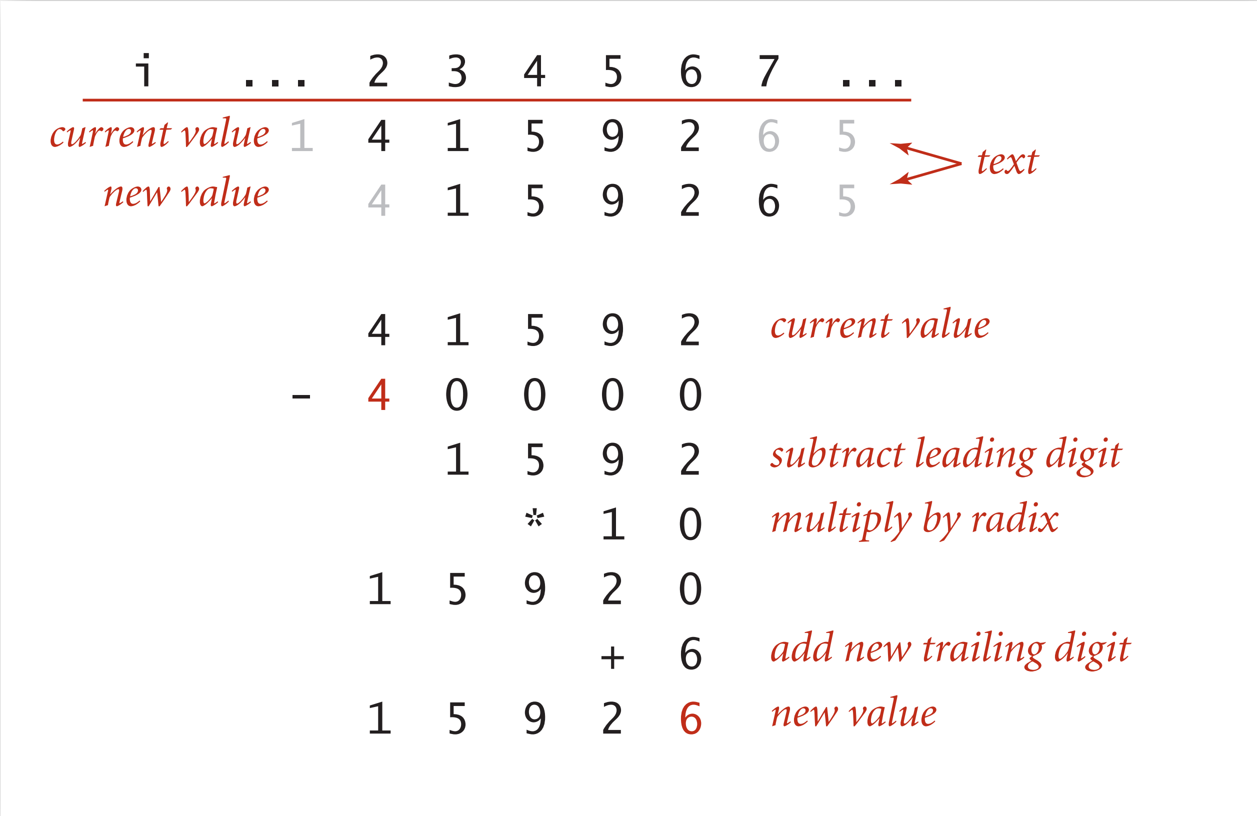
$$x_i = t_i R^{M-1} + t_{i+1} R^{M-2} + \ldots + t_{i+M-1} R^0$$

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

Key property. Can update hash function in constant time!

$$x_{i+1} = (\underset{\substack{\uparrow \\ \text{current} \\ \text{value}}}{x_i} - \underset{\substack{\uparrow \\ \text{subtract} \\ \text{leading digit}}}{t_i} R^{M-1}) \underset{\substack{\uparrow \\ \text{multiply} \\ \text{by radix}}}{R} + \underset{\substack{\uparrow \\ \text{add new} \\ \text{trailing digit}}}{t_{i+M}}$$

(can precompute R^{M-2})



i	...	2	3	4	5	6	7	...	
current value	1	4	1	5	9	2	6	5	*text*
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 substring search example

```
 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  % 997 = 3
                                 Q
 1   3  1  % 997 = (3*10 + 1) % 997 = 31
 2   3  1  4  % 997 = (31*10 + 4) % 997 = 314
 3   3  1  4  1  % 997 = (314*10 + 1) % 997 = 150
 4   3  1  4  1  5  % 997 = (150*10 + 5) % 997 = 508
                                                 RM   R
 5      1  4  1  5  9  % 997 = ((508 + 3*(997 - 30))*10 + 9) % 997 = 201
 6         4  1  5  9  2  % 997 = ((201 + 1*(997 - 30))*10 + 2) % 997 = 715
 7            1  5  9  2  6  % 997 = ((715 + 4*(997 - 30))*10 + 6) % 997 = 971
 8               5  9  2  6  5  % 997 = ((971 + 1*(997 - 30))*10 + 5) % 997 = 442
                                                                                   match
 9                  9  2  6  5  3  % 997 = ((442 + 5*(997 - 30))*10 + 3) % 997 = 929
10 ← return i-M+1 = 6  2  6  5  3  5  % 997 = ((929 + 9*(997 - 30))*10 + 5) % 997 = 613
```

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 RM;         // R^(M-1) % Q

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

      RM = 1;
      for (int i = 1; i <= M-1; i++)
         RM = (R * RM) % 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;
}
```

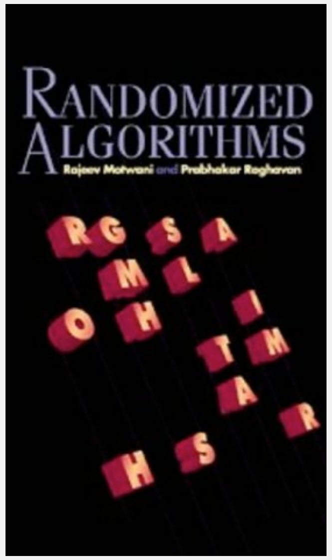
check for hash collision
using rolling hash function

Las Vegas version. 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 as 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 $M N$).

Rabin-Karp fingerprint search

Advantages.

- Extends to 2d patterns.
- Extends to finding multiple patterns.

Disadvantages.

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

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)*	$7N$	$7N$	*no*	*yes* †	1
	Las Vegas	$7N$ †	$7N$	*yes*	*yes*	1

† *probabilisitic guarantee, with uniform hash function*