CS22012: Data Structures and Algorithms II

Substring Search

Ivana.Dusparic@scss.tcd.ie

Outline of Substring search algorithms

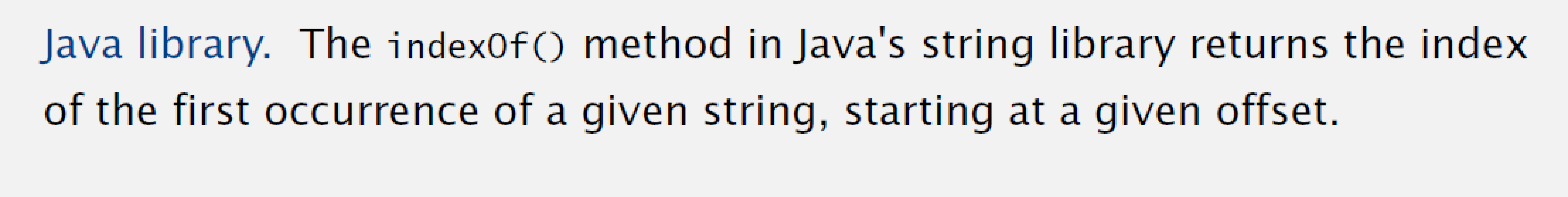
- Brute force
- KMP (Knuth-Morris-Pratt)
- Boyer-Moore
- Rabin-Karp
- Many many many others

- Suffix arrays
- LCP (longest common prefix) arrays

String matching algorithms

- Handbook of Exact String Matching Algorithms
- http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.133.4896&rep=rep1&type=pdf
- http://www-igm.univ-mlv.fr/~lecroq/string/

Java String 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.

- Which algorithm does String.IndexOf(String) use?
  - Naïve loop (brute force)
  - Why?
- String.contains()

Common interview questions

- Implement a needle-in-a-haystack
  - public int Search(String haystack, String needle)
- Implement strstr()
  - Find the first instance of a string in another string
- Longest common substring between 2 files
- Longest substring that's a palindrome
- Longest repeated substring
- Etc etc

Different to Pattern Matching

- Find a pattern, i.e. one of the specified set of substrings in a text
- Regular expression – notation to specify a set of strings
- For more info see 5.4 in Sedgewick and Wayne

Substring search - definition

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

typically N >> M

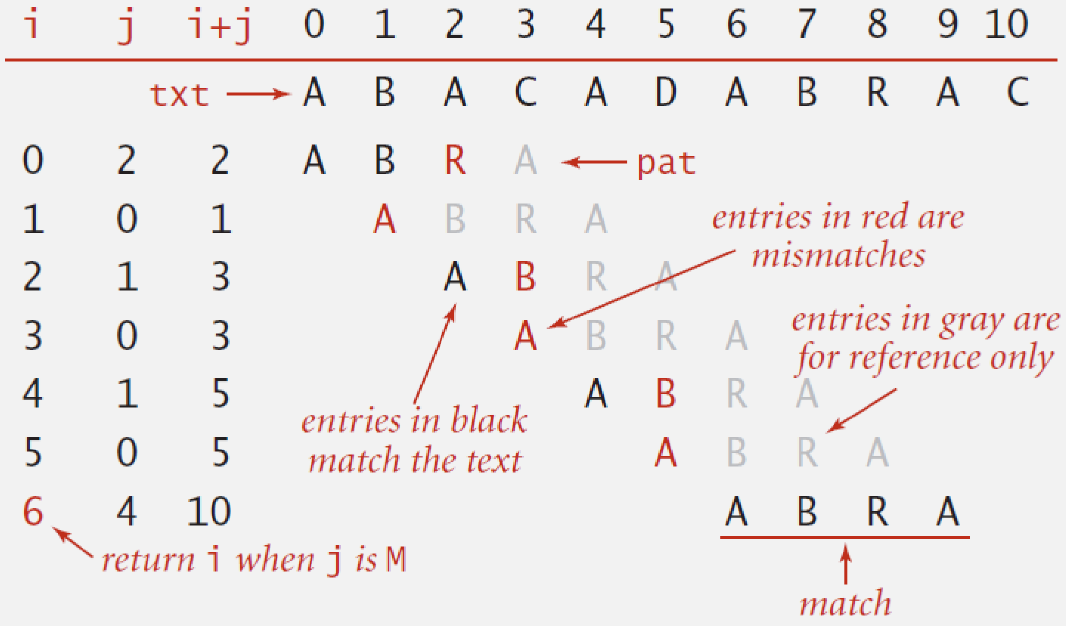
pattern → N E E D L E

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

match

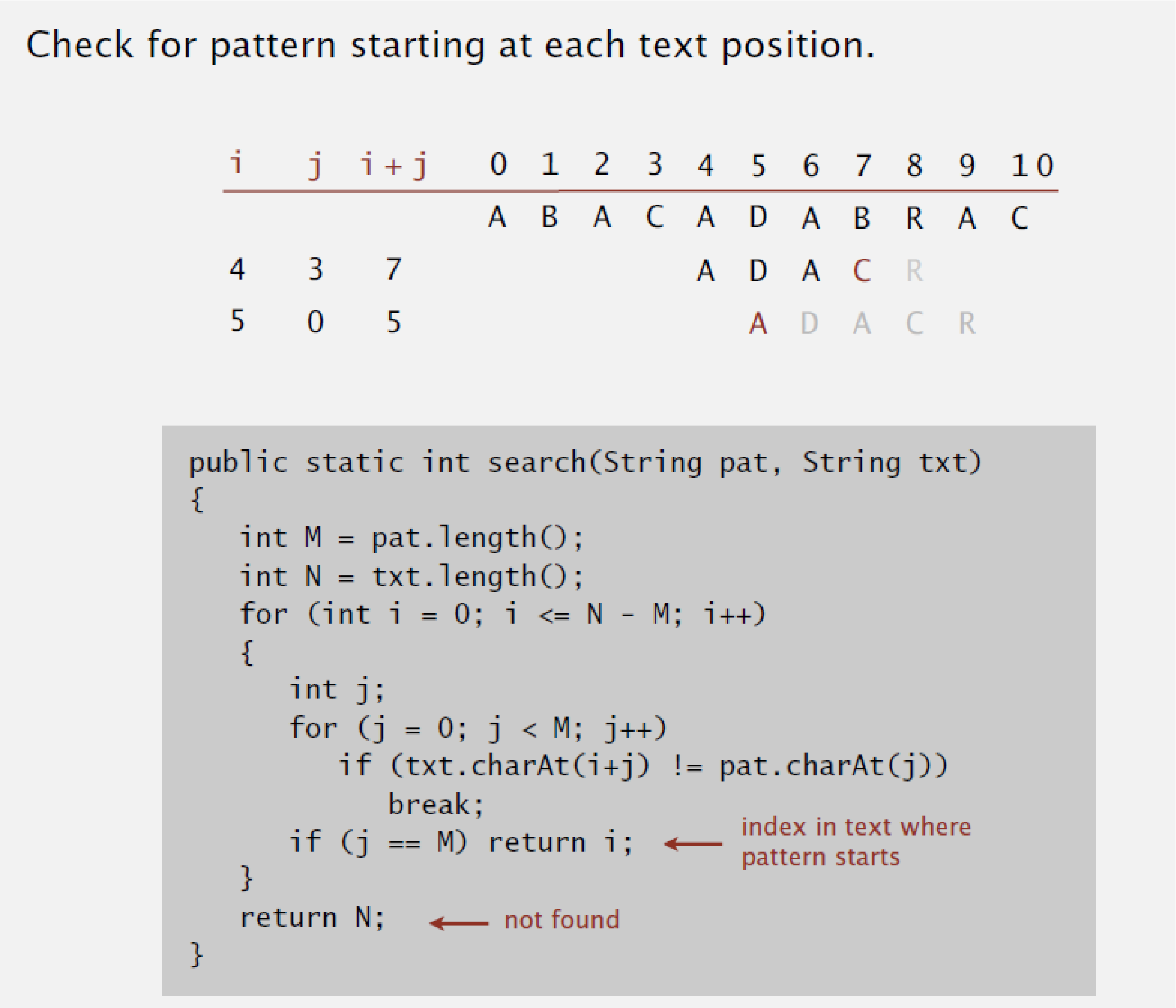
Substring search – brute force

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	

Substring search – brute force



Check for pattern starting at each text position.

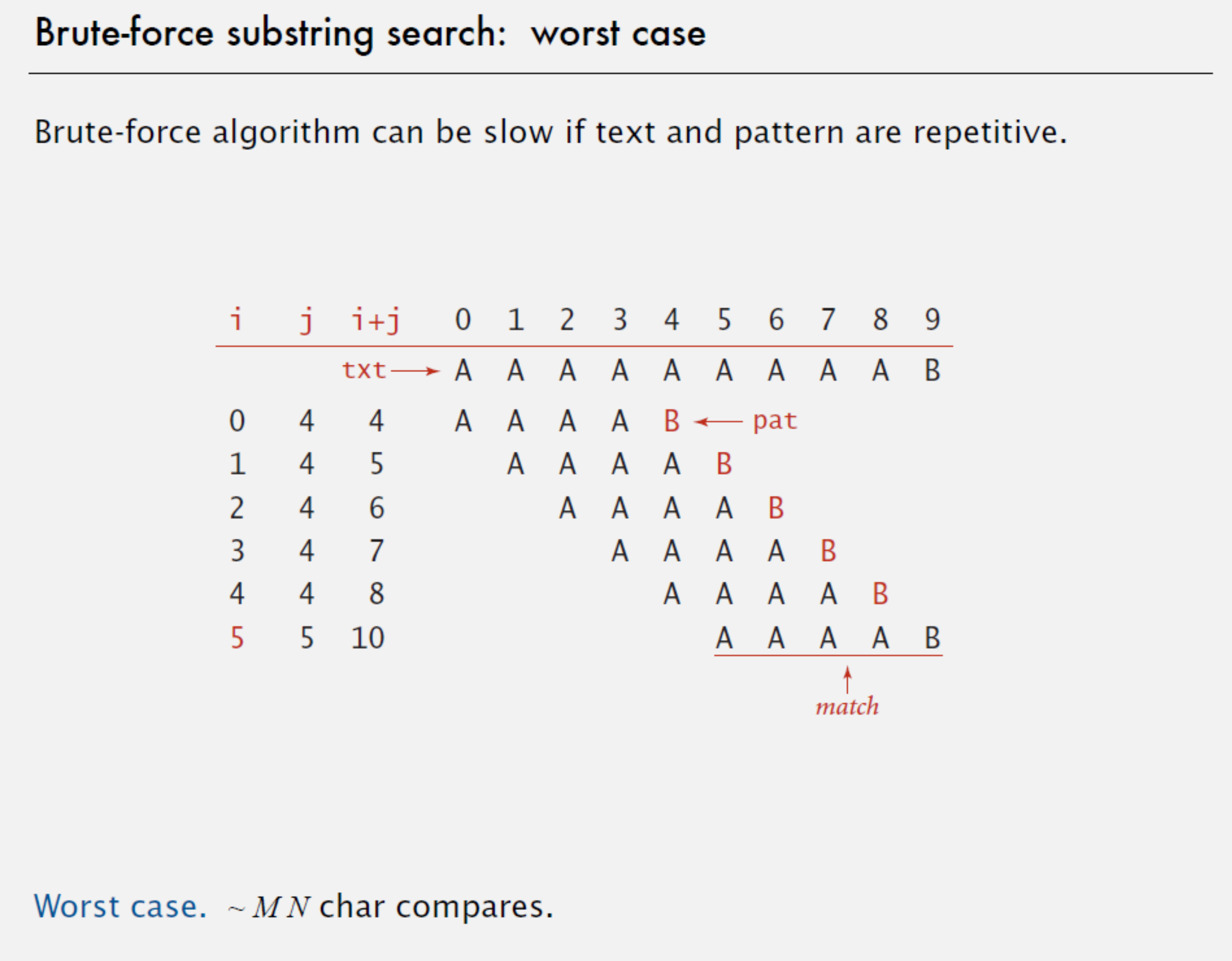
i	j	i+j	0	1	2	3	4	5	6	7	8	9	10
			A	B	A	C	A	D	A	B	R	A	C
4	3	7					A	D	A	C	R		
5	0	5						A	D	A	C	R	

```
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;
}
```

← index in text where pattern starts

← not found

Substring search – brute force



Brute-force substring search: worst case

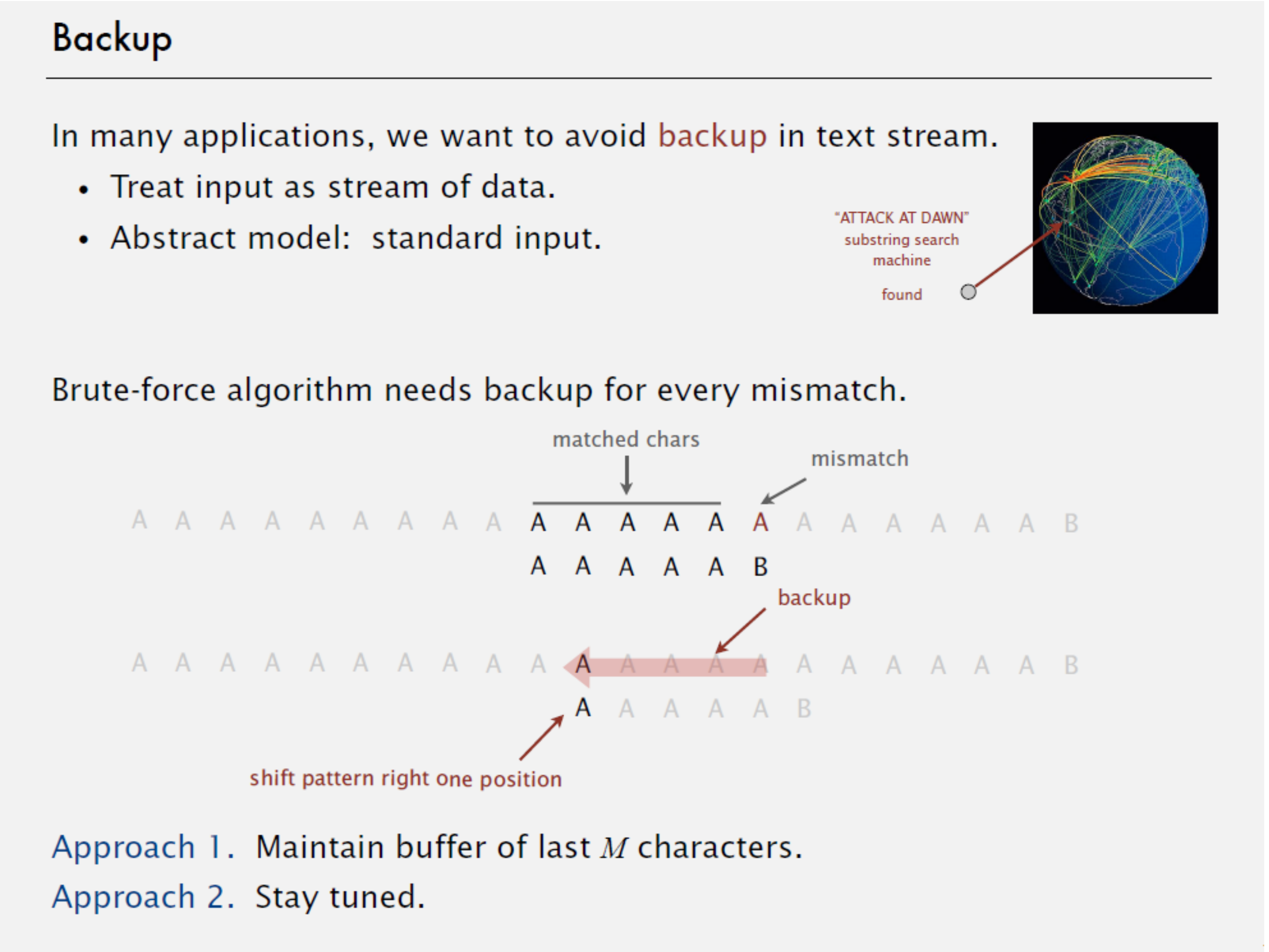
Brute-force algorithm can be slow if text and pattern are repetitive.

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.

Substring search – backup



Backup

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

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

Brute-force algorithm needs backup for every mismatch.

Approach 1. Maintain buffer of last M characters.

Approach 2. Stay tuned.

Substring search – explicit backup

Same sequence of char compares as previous implementation.

- i points to end of sequence of already-matched chars in text.
- j stores # 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;
}
```

explicit 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.
```

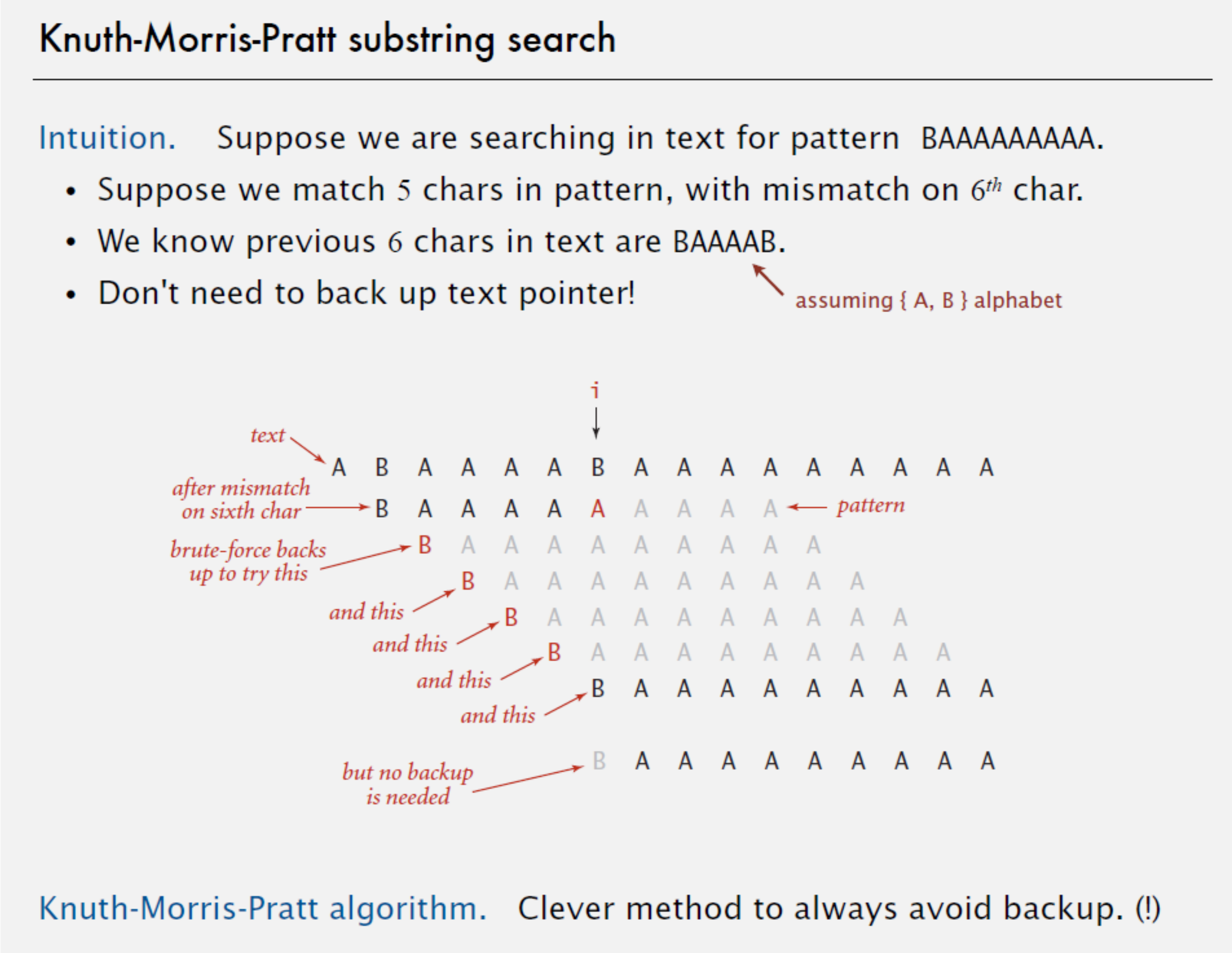
Knuth-Morris-Pratt (KMP)

KMP

- 1970 by Donald Knuth and Vaughan Pratt
- + Independently by James H. Morris.

- (Donald Knuth - The Art of Computer Programming - comprehensive monograph that covers many kinds of programming algorithms and their analysis – 4 volumes and counting)

KMP



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!

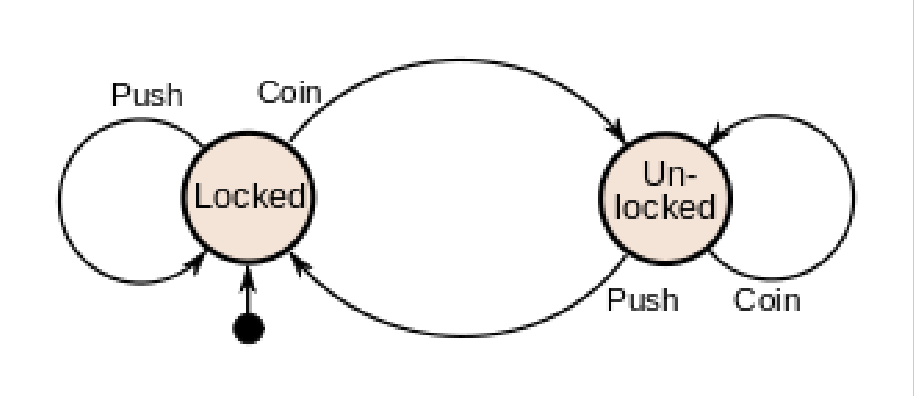
assuming { A, B } alphabet

Knuth-Morris-Pratt algorithm. Clever method to always avoid backup. (!)

KMP – avoid back up how? DFA

- DFA – Deterministic Final State Automaton
- Finite State Automaton/Finite State Machine
  - mathematical model of computation
  - an abstract machine that can be in exactly one of a finite number of states at any given time.
  - can change from one state to another in response to some external inputs
  - the change from one state to another is called a transition
  - defined by a list of its states, its initial state, and the conditions for each transition.
- Deterministic - produces a unique computation (or run) of the automaton for each input string
- DFA - finite-state machine that accepts and rejects strings of symbols

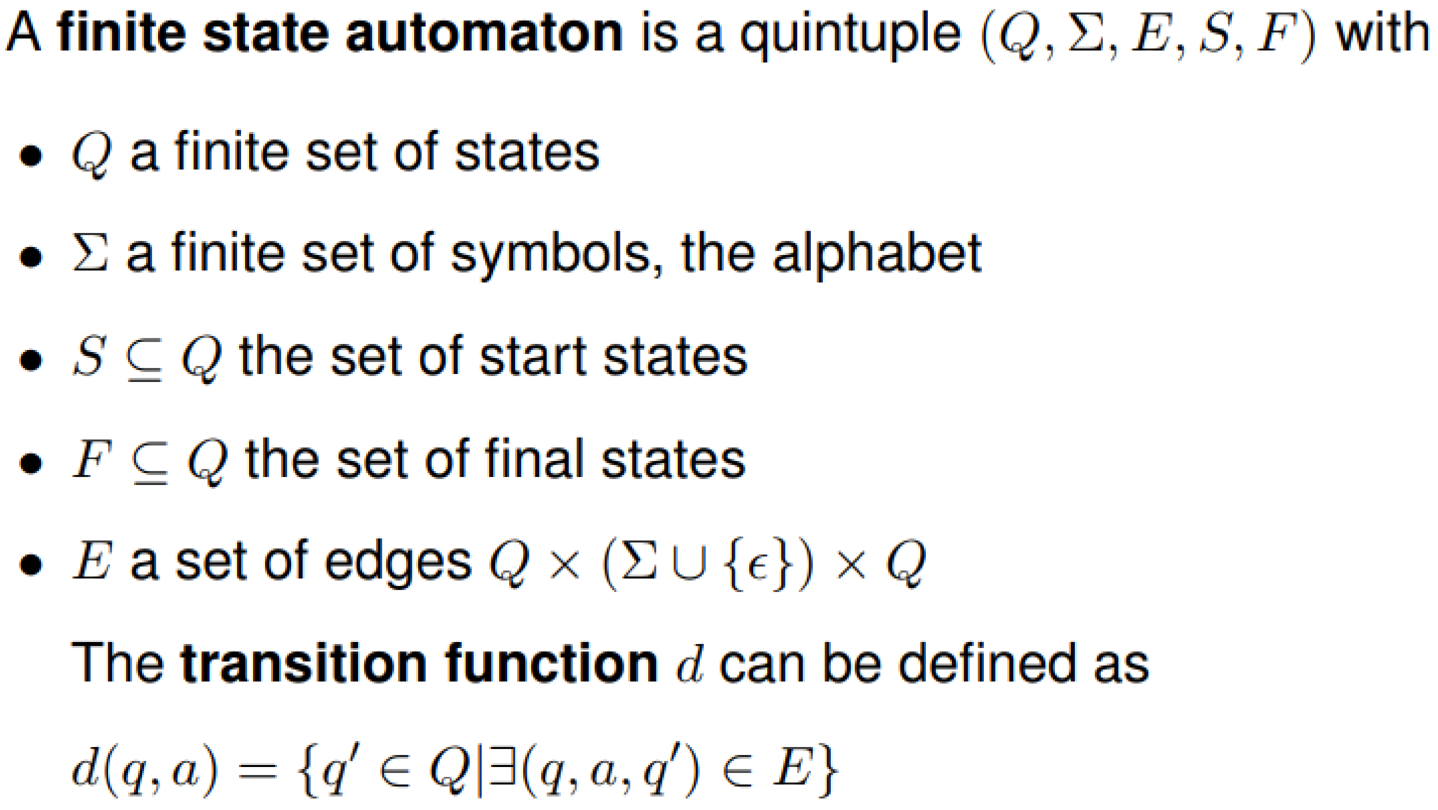
FSA – an example



State diagram for a turnstile

A turnstile

Finite State Machine – more formally



A **finite state automaton** is a quintuple (Q, Σ, E, S, F) with

- Q a finite set of states
- Σ a finite set of symbols, the alphabet
- $S \subseteq Q$ the set of start states
- $F \subseteq Q$ the set of final states
- E a set of edges $Q \times (\Sigma \cup \{\epsilon\}) \times Q$

  The **transition function** d can be defined as

  $$d(q, a) = \{q' \in Q | \exists (q, a, q') \in E\}$$

- Deterministic Finite Automata are always *complete*: they define a transition for each state and each input symbol.

DFA

DFA is abstract string-searching machine.

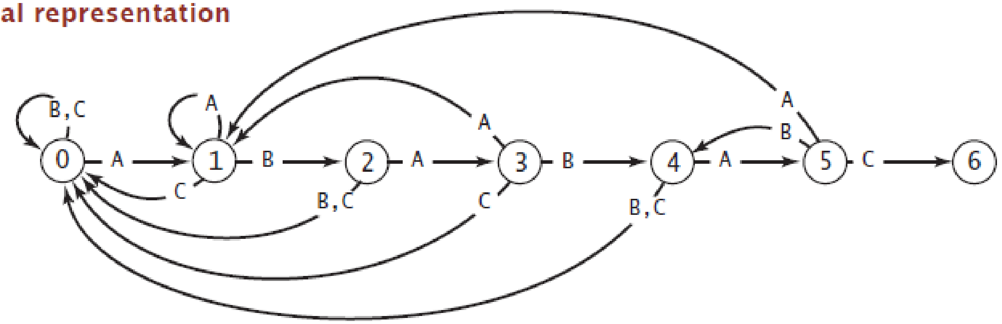
- Finite number of states (including start and halt).
- Exactly one transition for each char in alphabet.
- Accept if sequence of transitions leads to halt state.

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



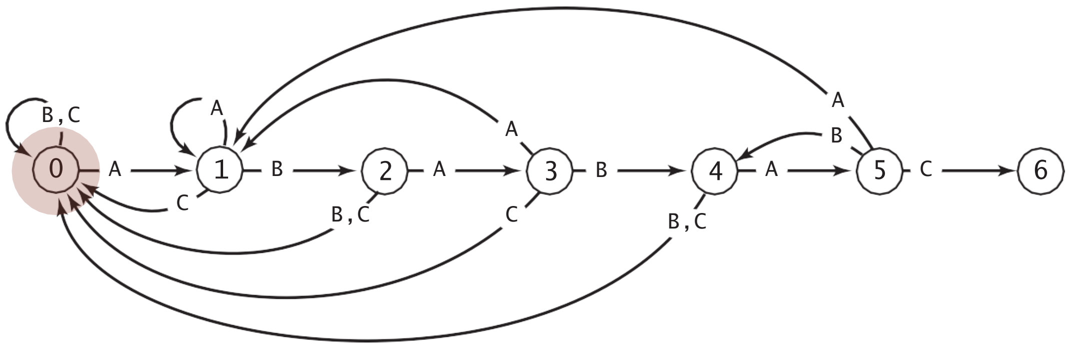
DFA simulation

DFA simulation

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

↑

	j	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



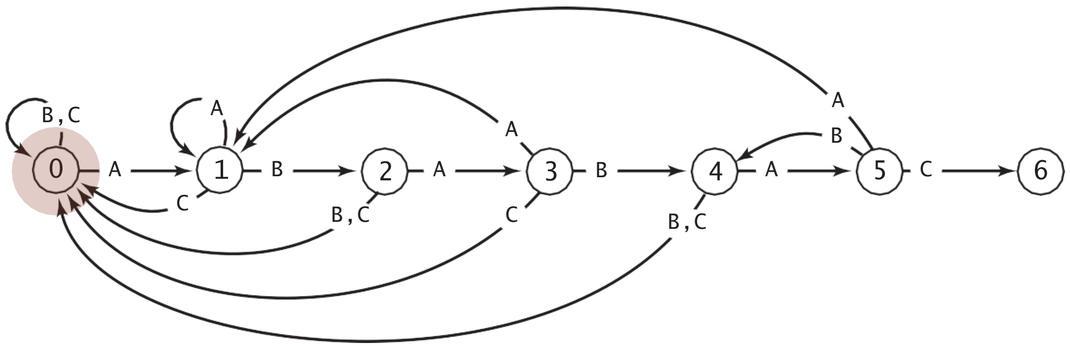
TRACE OF KMP SUBSTRING SEARCH (DFA SIMULATION) FOR A B A B A C

DFA simulation

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

↑

j		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

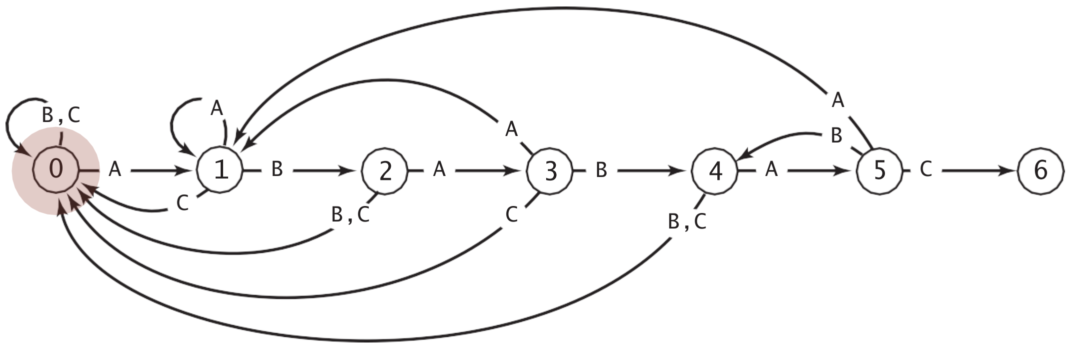


TRACE OF KMP SUBSTRING SEARCH (DFA SIMULATION) FOR A B A B A C

DFA simulation

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

j		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

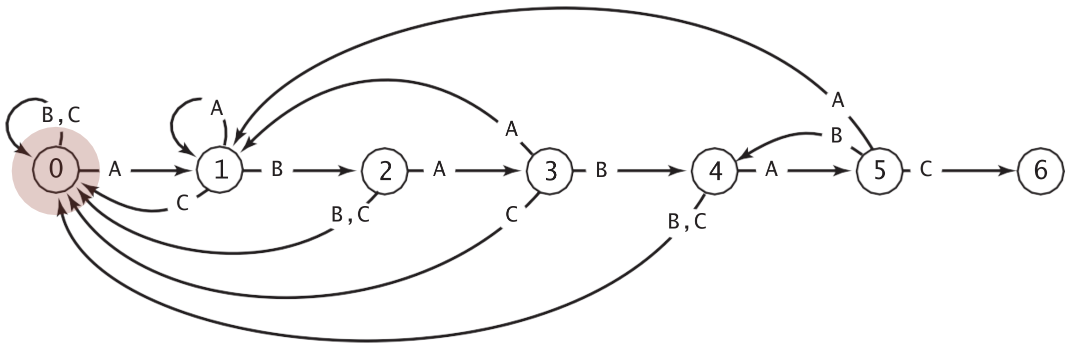


TRACE OF KMP SUBSTRING SEARCH (DFA SIMULATION) FOR A B A B A C

DFA simulation

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

	j	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



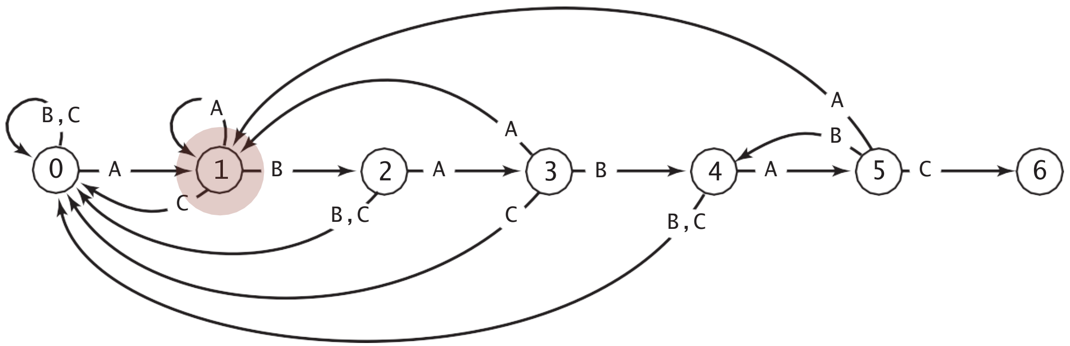
TRACE OF KMP SUBSTRING SEARCH (DFA SIMULATION) FOR A B A B A C

DFA simulation

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

↑

	j	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

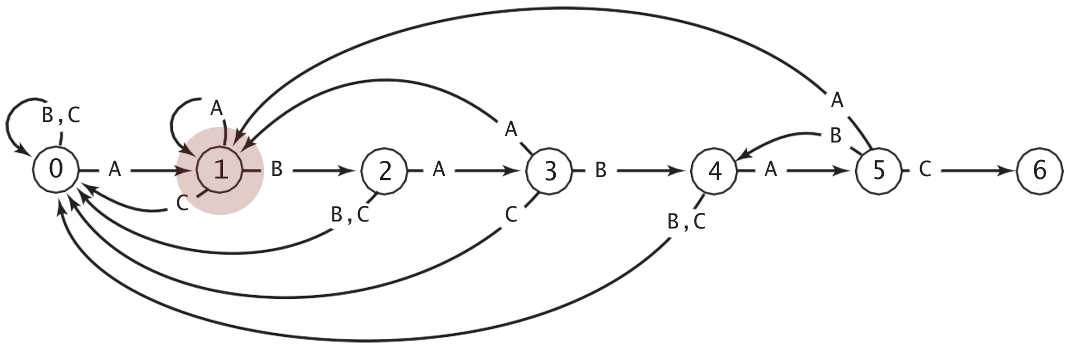


TRACE OF KMP SUBSTRING SEARCH (DFA SIMULATION) FOR A B A B A C

DFA simulation

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

	j	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

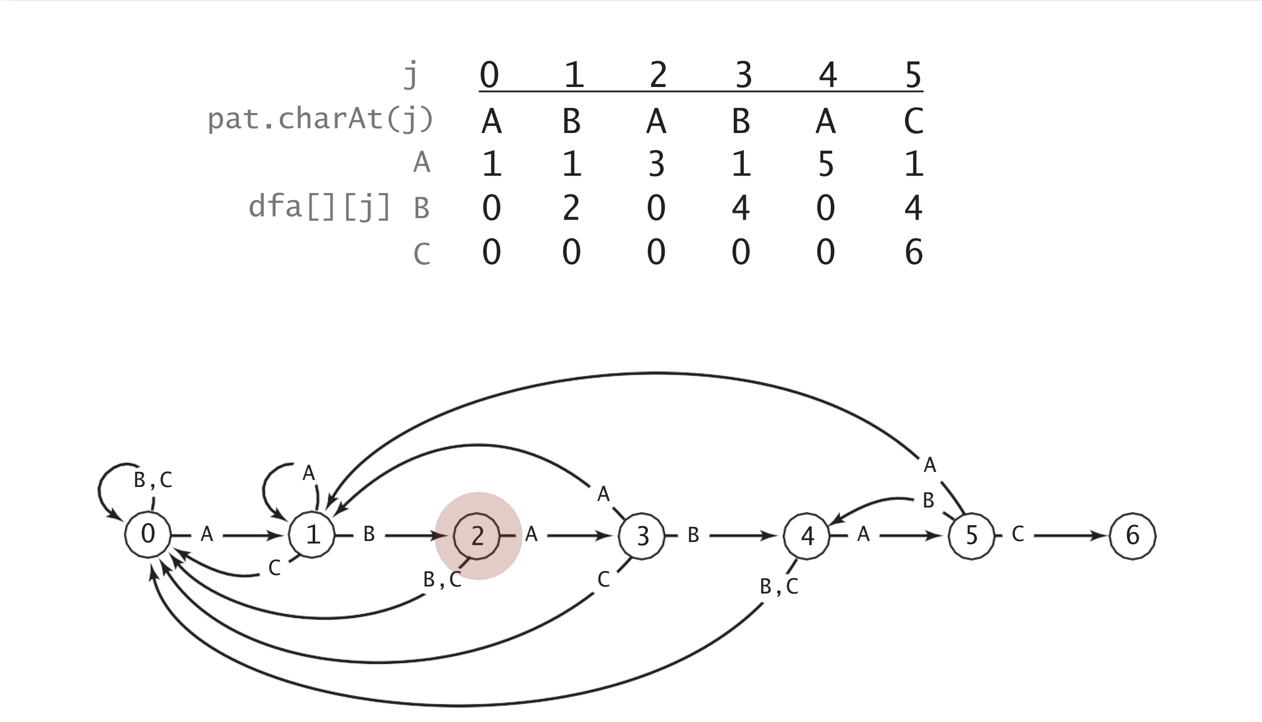


TRACE OF KMP SUBSTRING SEARCH (DFA SIMULATION) FOR A B A B A C

DFA simulation

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

↑



	j	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

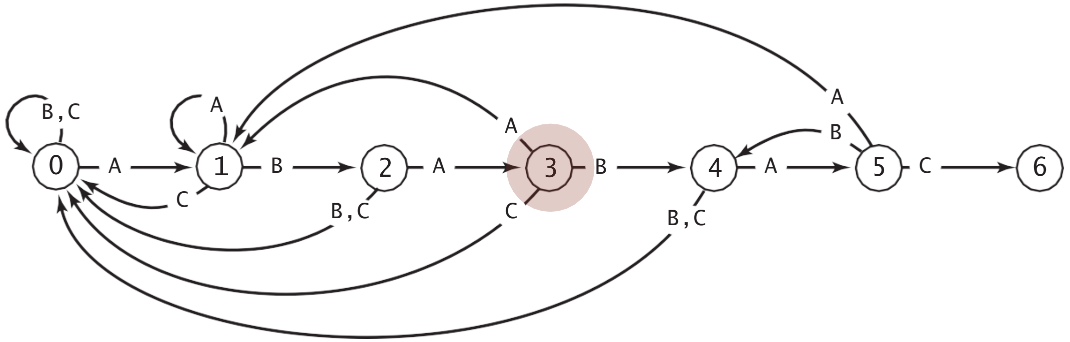
TRACE OF KMP SUBSTRING SEARCH (DFA SIMULATION) FOR A B A B A C

DFA simulation

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

↑

	j	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

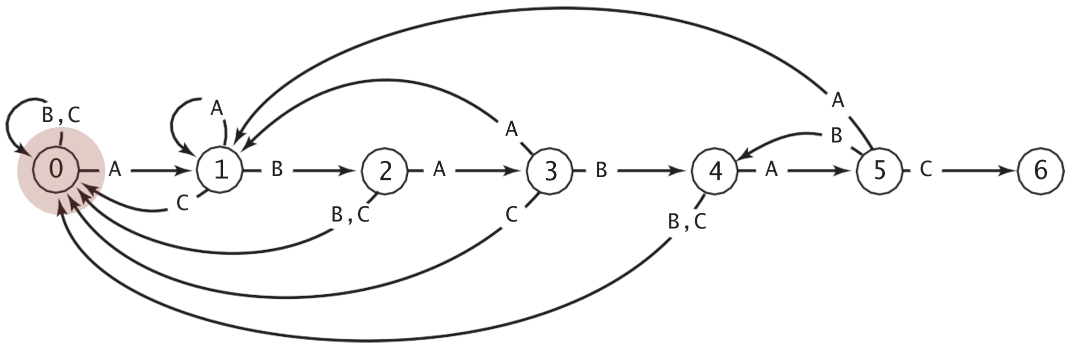


TRACE OF KMP SUBSTRING SEARCH (DFA SIMULATION) FOR A B A B A C

DFA simulation

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

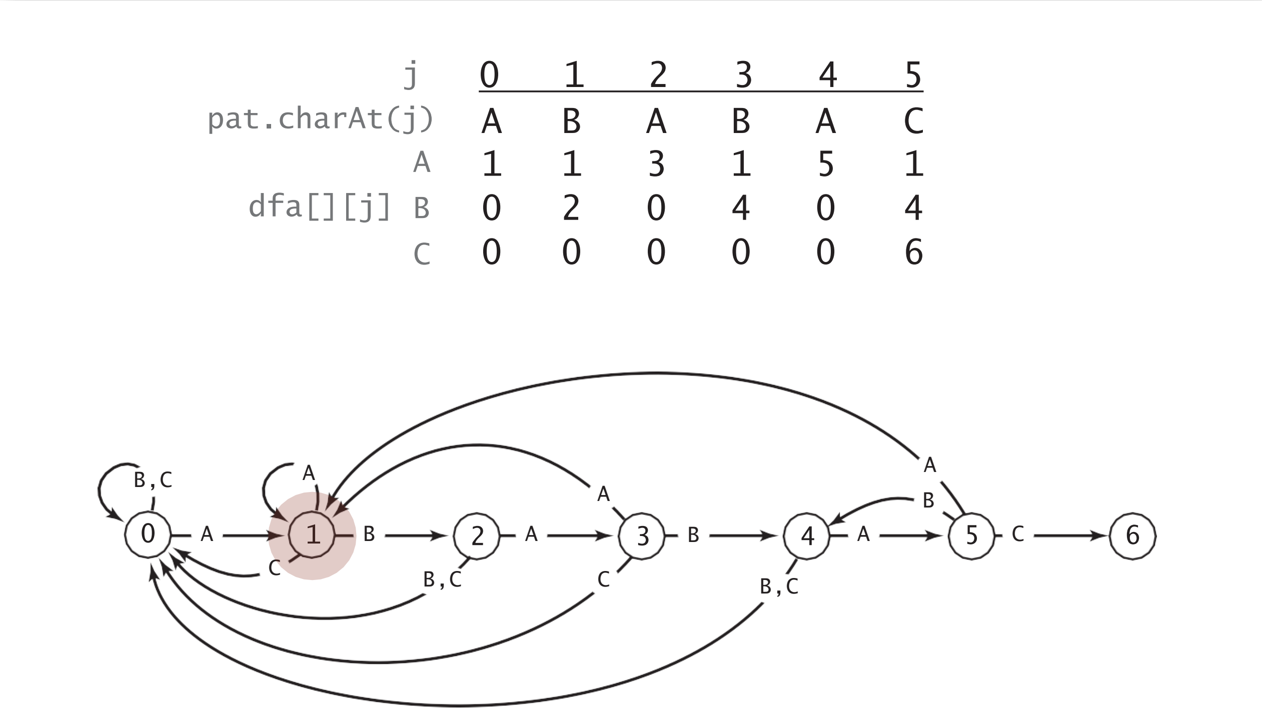
	j	0	1	2	3	4	5
	pat.charAt(j)	A	B	A	B	A	C
	A	1	1	3	1	5	1
dfa[][j]	B	0	2	0	4	0	4
	C	0	0	0	0	0	6



TRACE OF KMP SUBSTRING SEARCH (DFA SIMULATION) FOR A B A B A C

DFA simulation

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



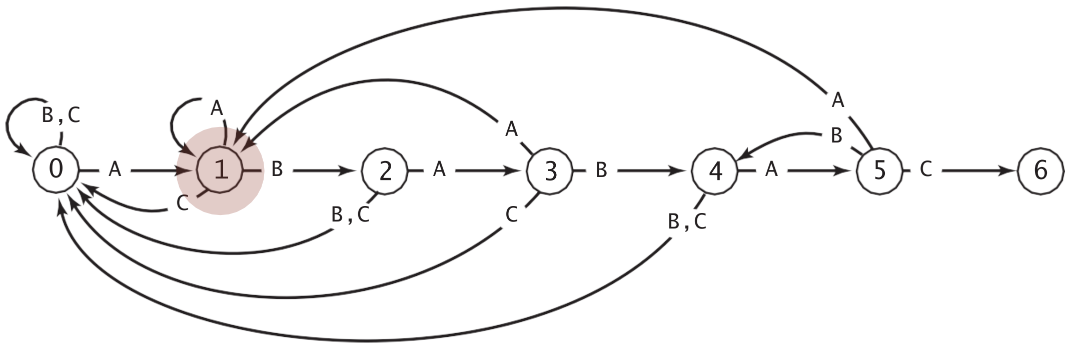
	j	0	1	2	3	4	5
	pat.charAt(j)	A	B	A	B	A	C
	A	1	1	3	1	5	1
dfa[][j]	B	0	2	0	4	0	4
	C	0	0	0	0	0	6

TRACE OF KMP SUBSTRING SEARCH (DFA SIMULATION) FOR A B A B A C

DFA simulation

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

	j	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



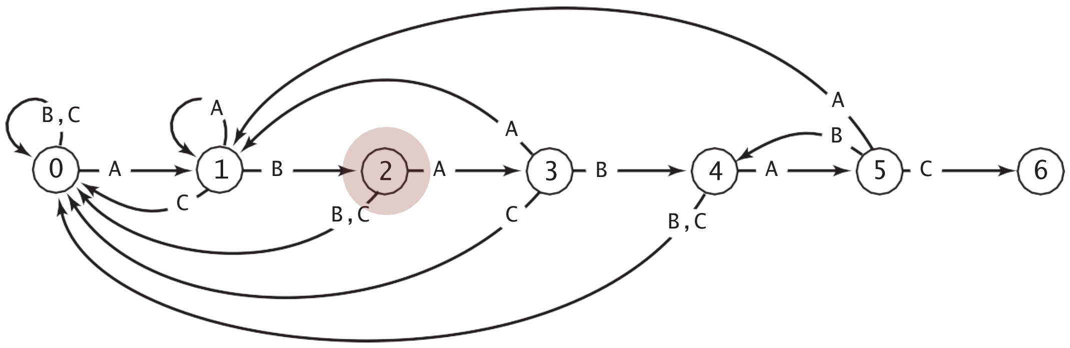
TRACE OF KMP SUBSTRING SEARCH (DFA SIMULATION) FOR A B A B A C

DFA simulation

B C B 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



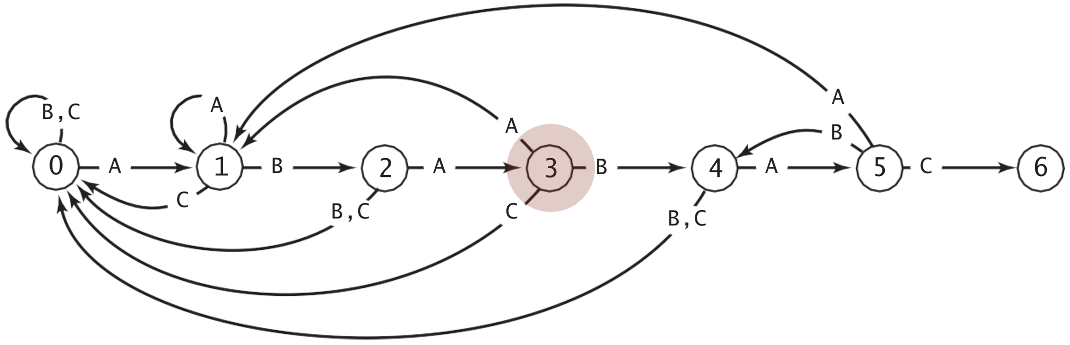
TRACE OF KMP SUBSTRING SEARCH (DFA SIMULATION) FOR A B A B A C

DFA simulation

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

↑

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

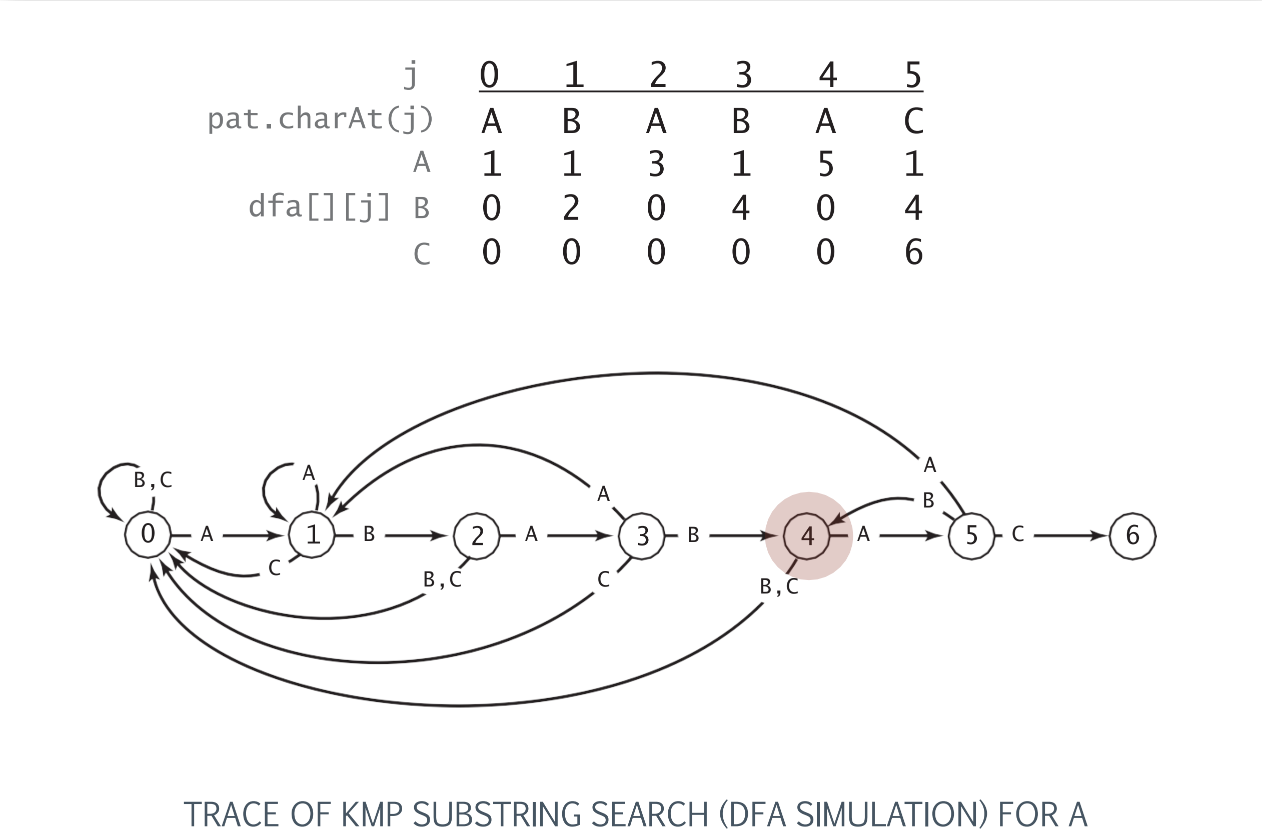


TRACE OF KMP SUBSTRING SEARCH (DFA SIMULATION) FOR A B A B A C

DFA simulation

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

↑



		0	1	2	3	4	5
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

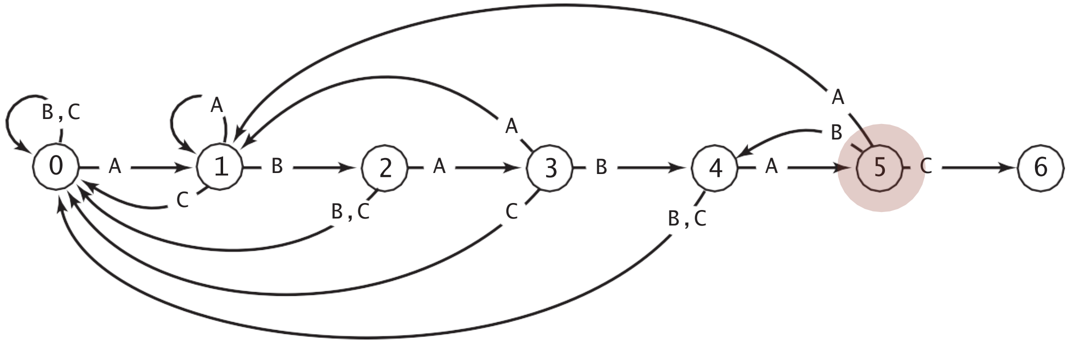
TRACE OF KMP SUBSTRING SEARCH (DFA SIMULATION) FOR A B A B A C

DFA simulation

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

↑

j		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



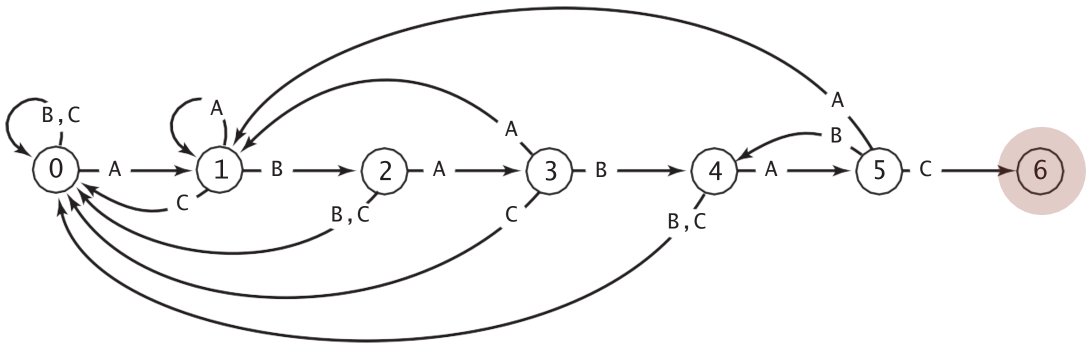
TRACE OF KMP SUBSTRING SEARCH (DFA SIMULATION) FOR A B A B A C

DFA simulation

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

↑

	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



TRACE OF KMP SUBSTRING SEARCH (DFA SIMULATION) FOR A B A B A C

DFA States – number of characters matched

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]`

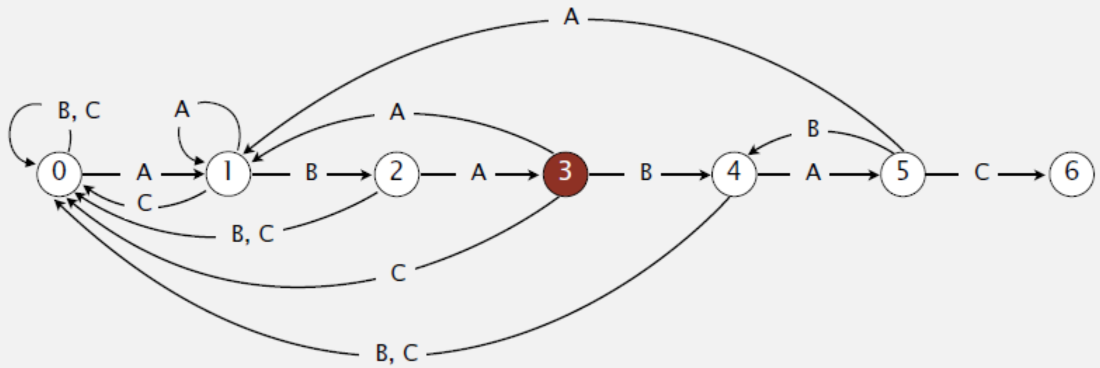
Ex. DFA is in state 3 after reading in `txt[0..6]`.

							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[]`



DFA simulation exercise

- Consider the following DFA for searching for a string "IVANA"
- For simplicity, we assume the alphabet contains only letters A, I, N, V
- DFA is therefore as follows

j	0	1	2	3	4
char?	I	V	A	N	A
A	0	0	3	0	5
I	1	1	1	1	1
N	0	0	0	4	0
V	0	2	0	0	0

Exercise 1 Simulating DFA:

- 1. Construct graphical representation of the DFA table
- 2. Write the trace of states when searching for a string “IVANA” in input “ANVAIVAAIVANAAN”
- 3. Vote on turning point for the correct trace

Exercise 1 Simulating DFA:

j	0	1	2	3	4
char?	I	V	A	N	A
A	0	0	3	0	0
I	1	1	1	1	1
N	0	0	0	4	0
V	0	2	0	0	5

Text ANVAIVAAIVANAAN

Search string IVANA

KMP 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]

DFA construction

Knuth-Morris-Pratt construction

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

j		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: advance to next state if `c == pat.charAt(j)`.

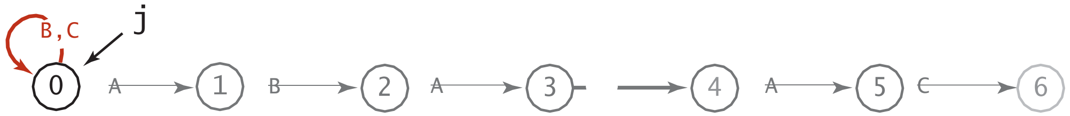
	j	0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
	A	1		3		5	
dfa[][j]	B		2		4		
	C						6

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)`.

		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

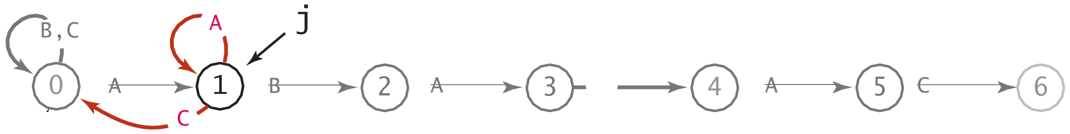


Constructing the DFA for KMP substring search for A B A B A C

Knuth-Morris-Pratt construction

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

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

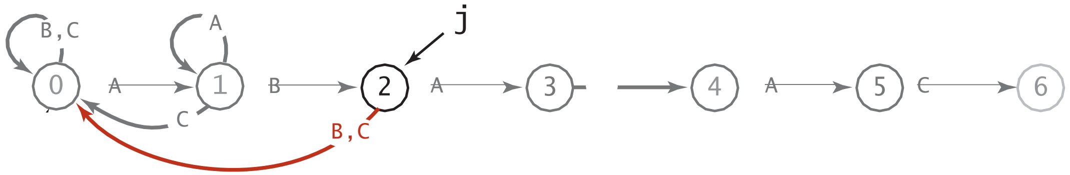


Constructing the DFA for KMP substring search for A B A B A C

Knuth-Morris-Pratt construction

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

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

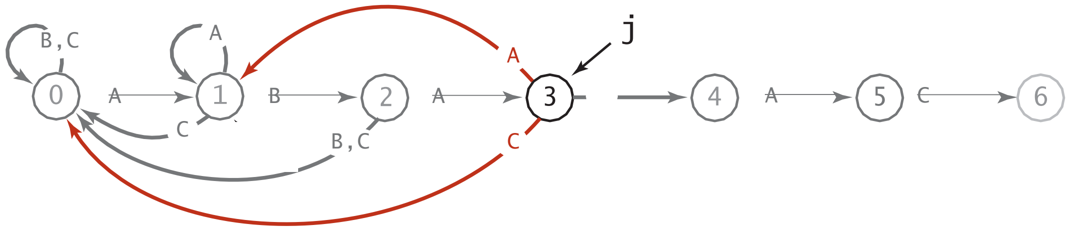


Constructing the DFA for KMP substring search for A B A B A C

Knuth-Morris-Pratt construction

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

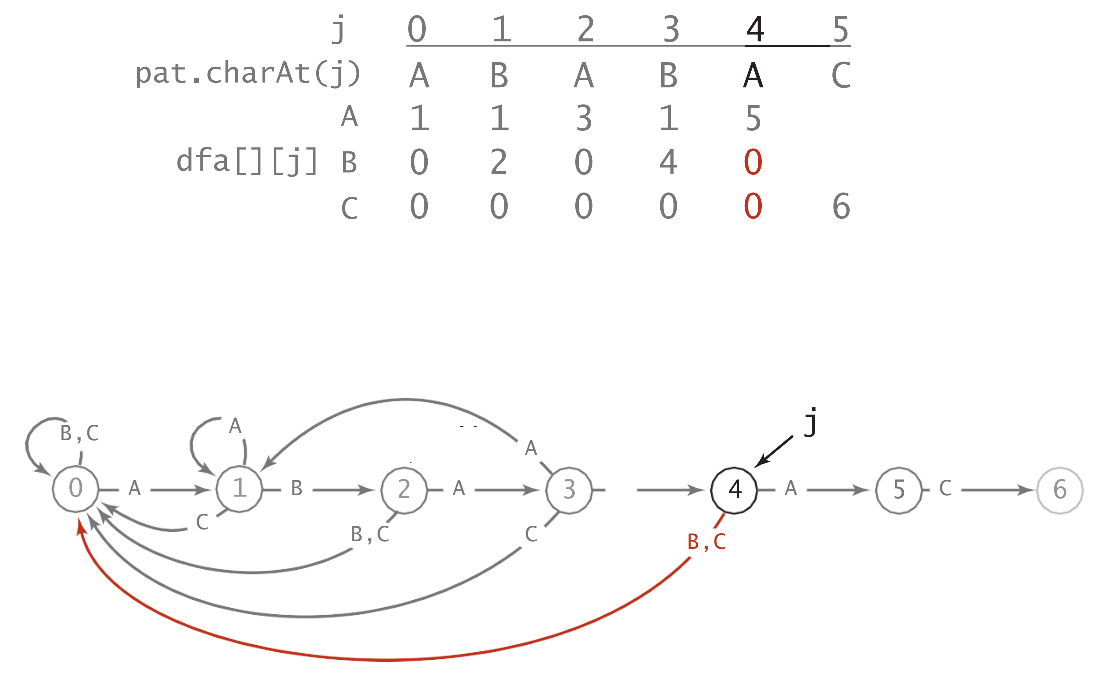
		0	1	2	3	4	5
j		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
	A	1	1	3	1	5	
dfa[][j]	B	0	2	0	4		
	C	0	0	0	0		6



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

Knuth-Morris-Pratt construction

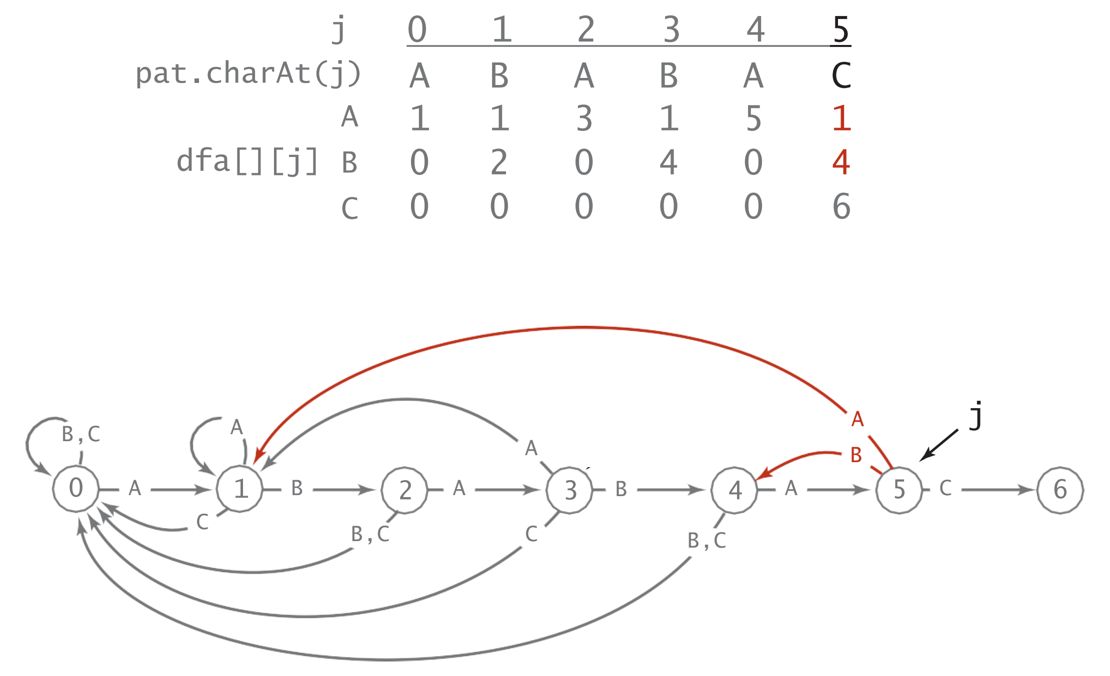
Mismatch transition: back up if `c != pat.charAt(j)`.



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

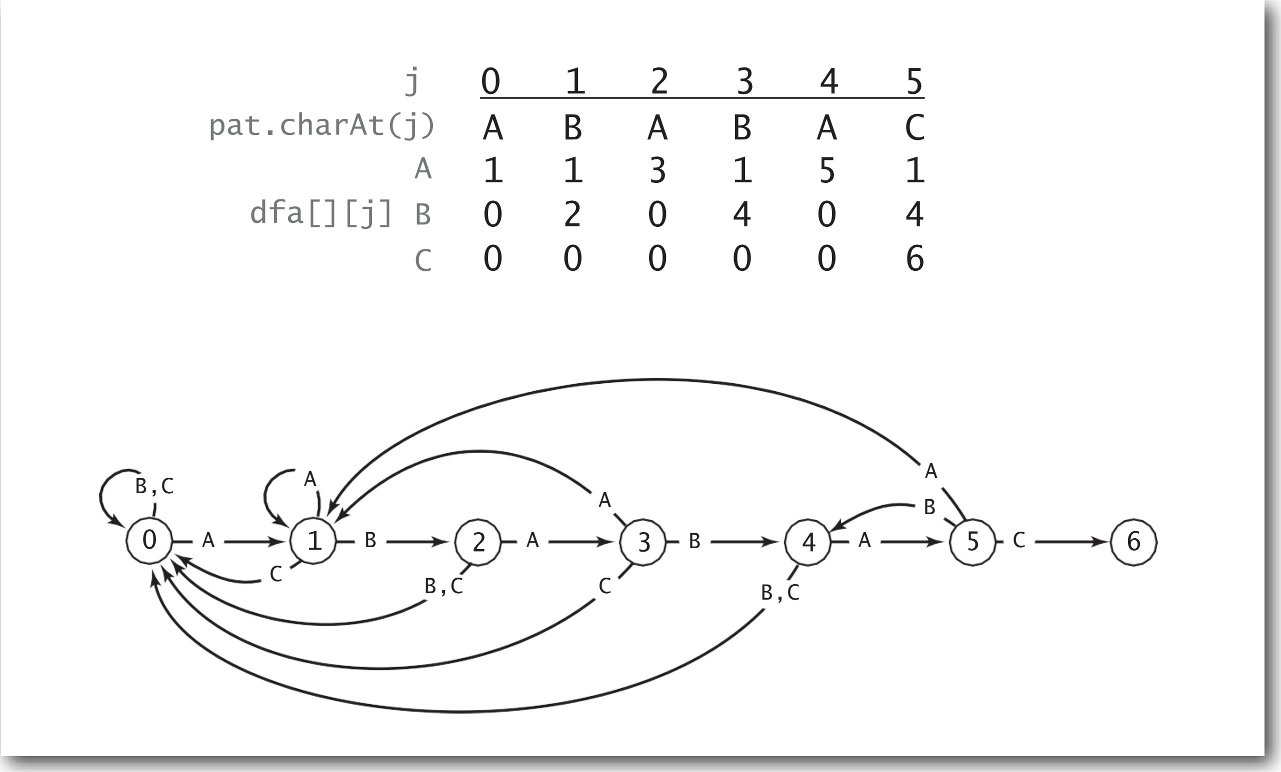
Knuth-Morris-Pratt construction

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



		0	1	2	3	4	5
j		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 construction



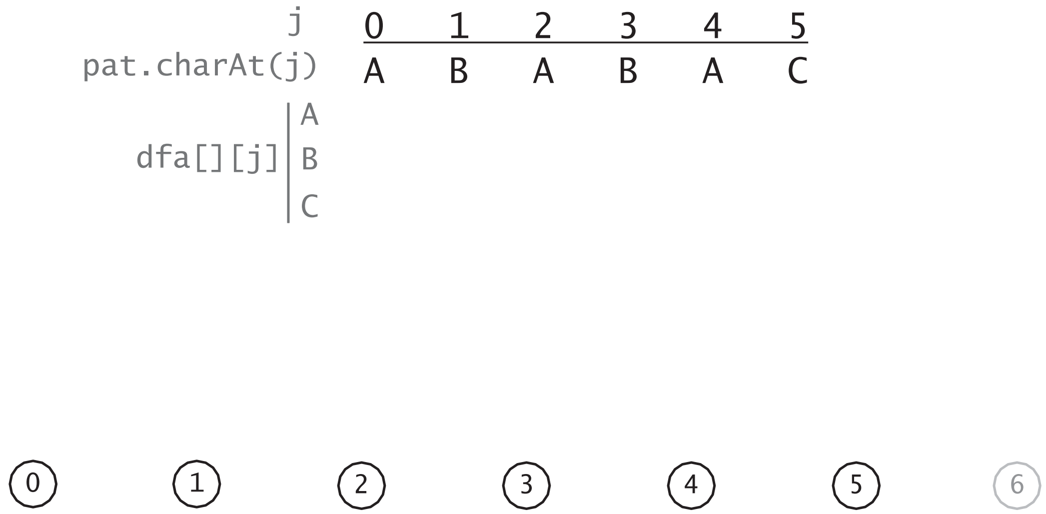
	j	0	1	2	3	4	5
	pat.charAt(j)	A	B	A	B	A	C
	A	1	1	3	1	5	1
dfa[][j]	B	0	2	0	4	0	4
	C	0	0	0	0	0	6

Exercise 2 constructing DFA:

- Construct DFA table and graphical representation for a search word “banana”
- Make up a 15-letter string in which you’re going to search for the word, assuming the alphabet contains only letters.
  - You can decide whether you want the string to contain the search word or not, but if it does, do not have it too early into the string
- Write out the trace of DFA states while searching for the word in the madeup string
- Hand up the exercise

DFA construction – Java code

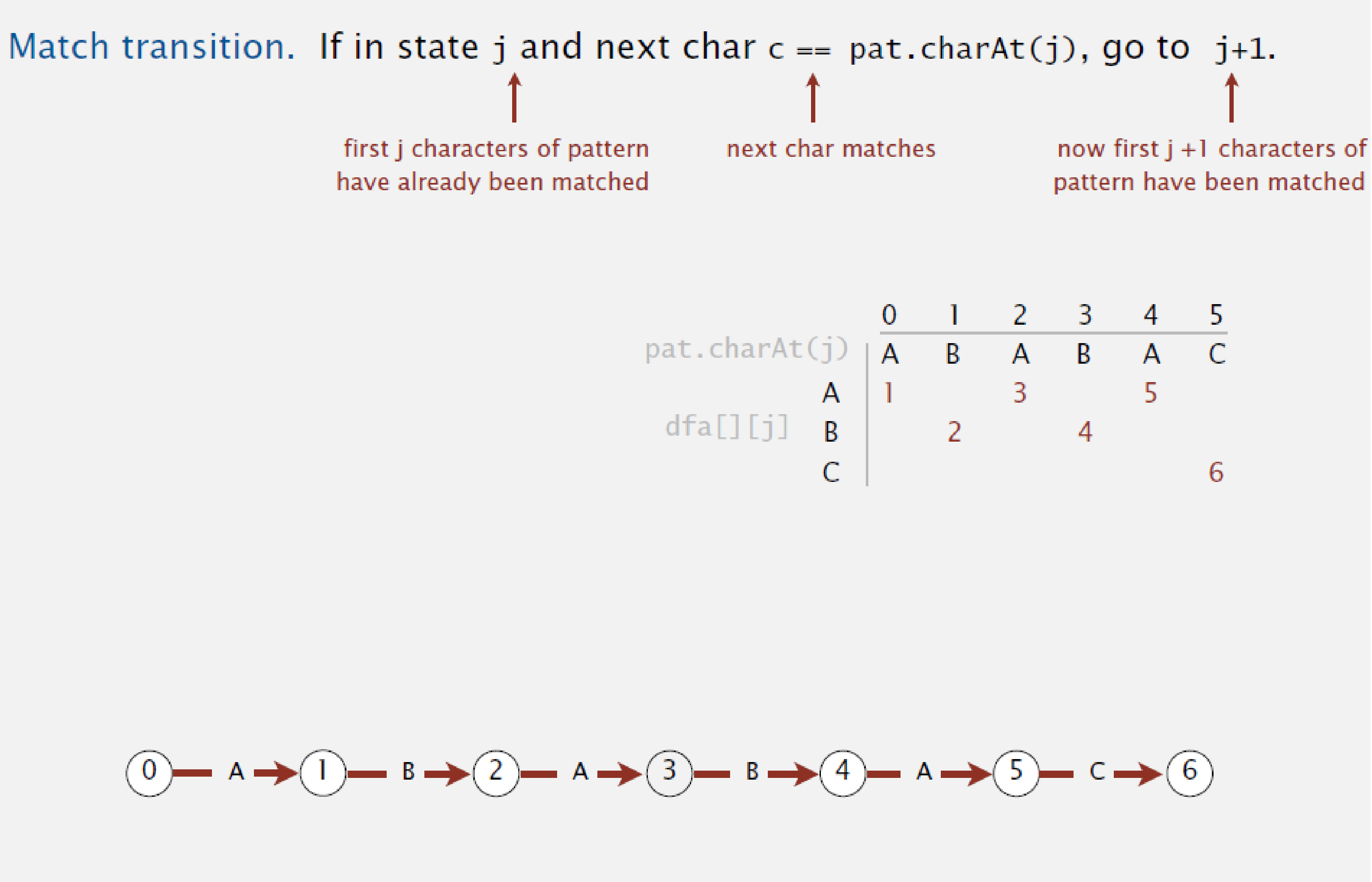
Include one state for each character in pattern (plus accept state).



j		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

DFA Construction – Java code



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

DFA construction – Java code

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

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

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

DFA Construction – Java code

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`.
Running time. Seems to require `j` steps.

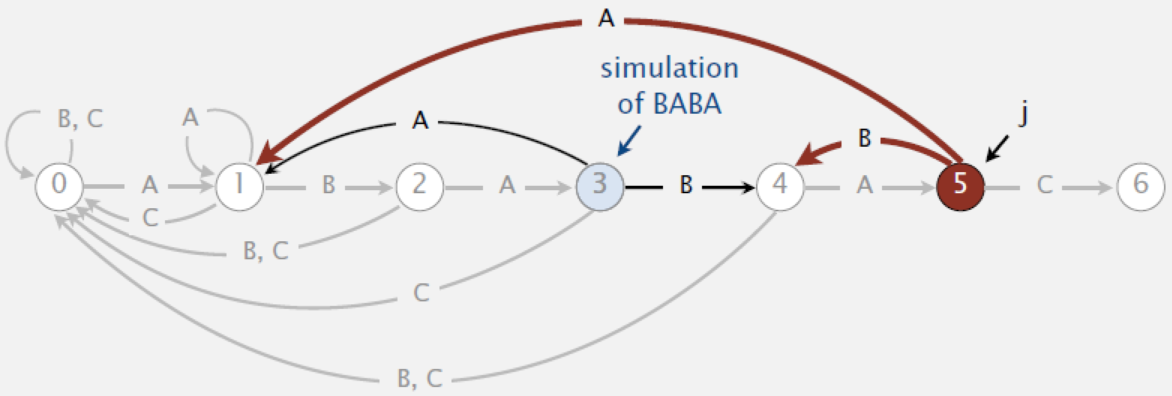
still under construction (!)

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



DFA construction

- So lets do this again, while maintaining state x

Knuth-Morris-Pratt construction

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

		0	1	2	3	4	5
j		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. 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

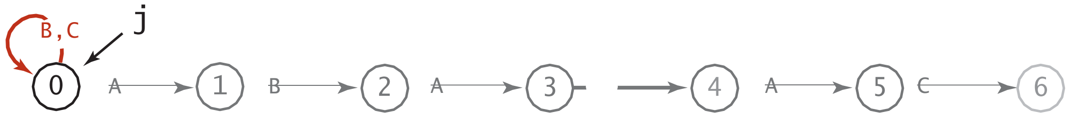
	j	0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1		3		5	
	B		2		4		
	C						6

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

Knuth-Morris-Pratt construction

Mismatch transition.

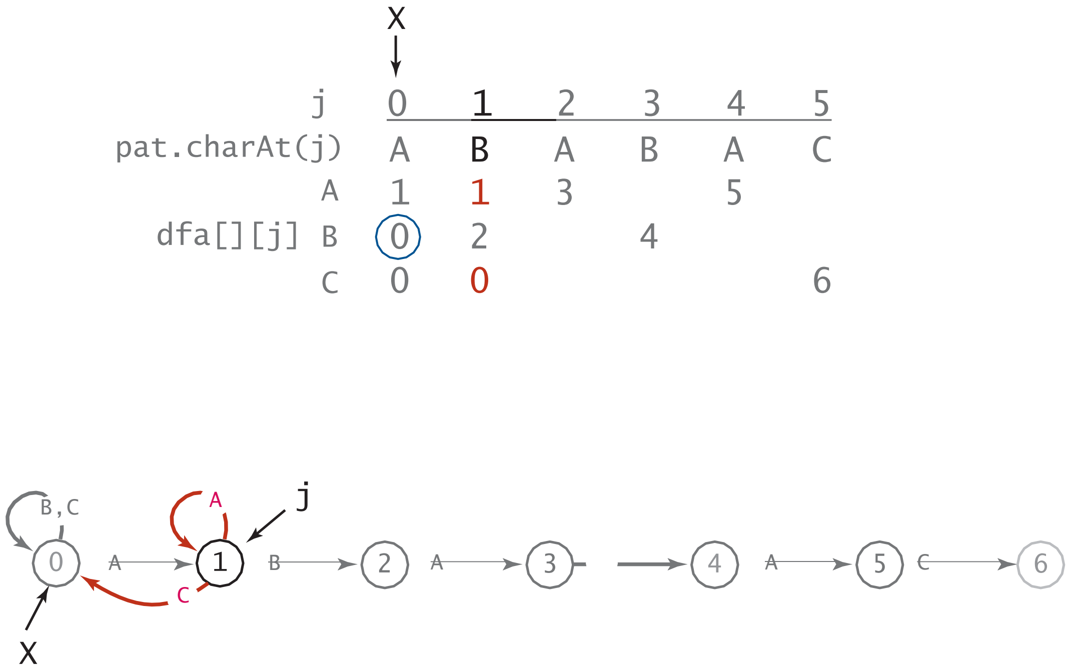
	j	0	1	2	3	4	5
	pat.charAt(j)	A	B	A	B	A	C
	A	1		3		5	
dfa[][j]	B	0	2		4		
	C	0					6



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

Knuth-Morris-Pratt construction

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

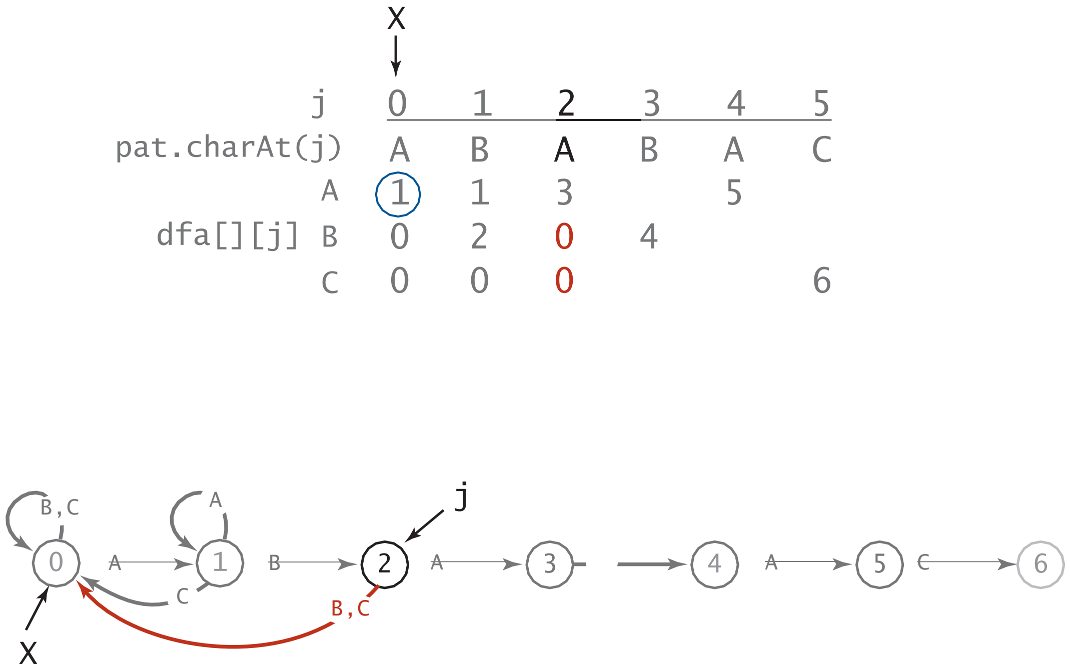


	j	0	1	2	3	4	5
	pat.charAt(j)	A	B	A	B	A	C
	A	1	1	3		5	
dfa[][j]	B	0	2		4		
	C	0	0				6

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

Knuth-Morris-Pratt construction

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

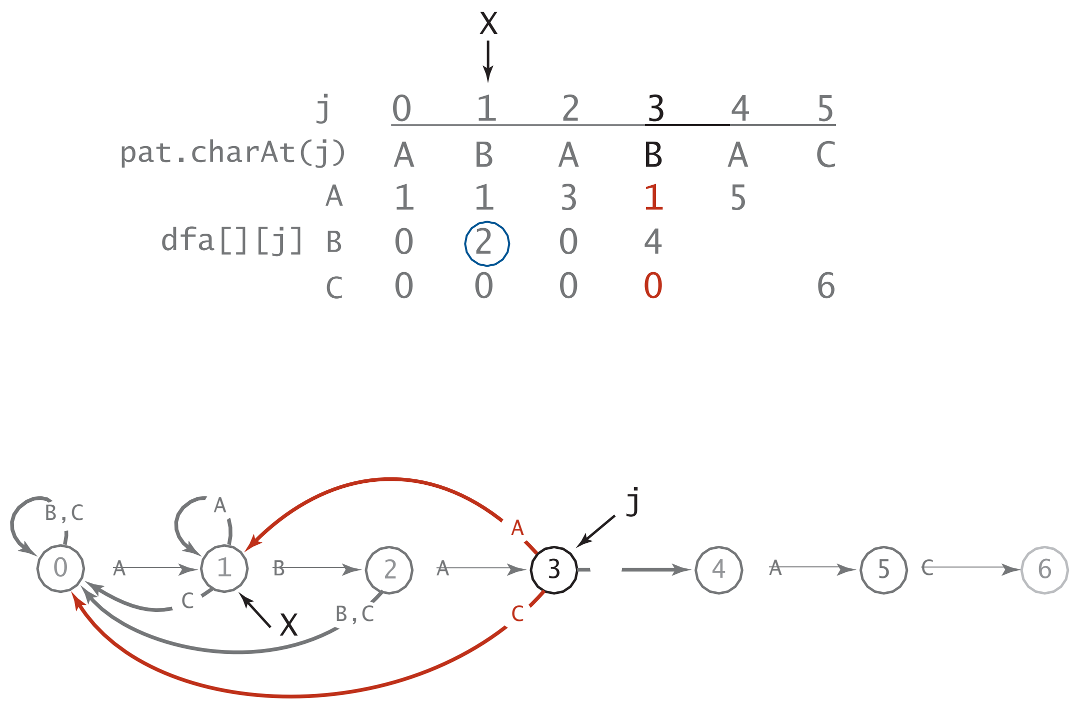


		X					
	j	0	1	2	3	4	5
	pat.charAt(j)	A	B	A	B	A	C
	A	1	1	3		5	
dfa[][j]	B	0	2	0	4		
	C	0	0	0			6

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

Knuth-Morris-Pratt construction

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

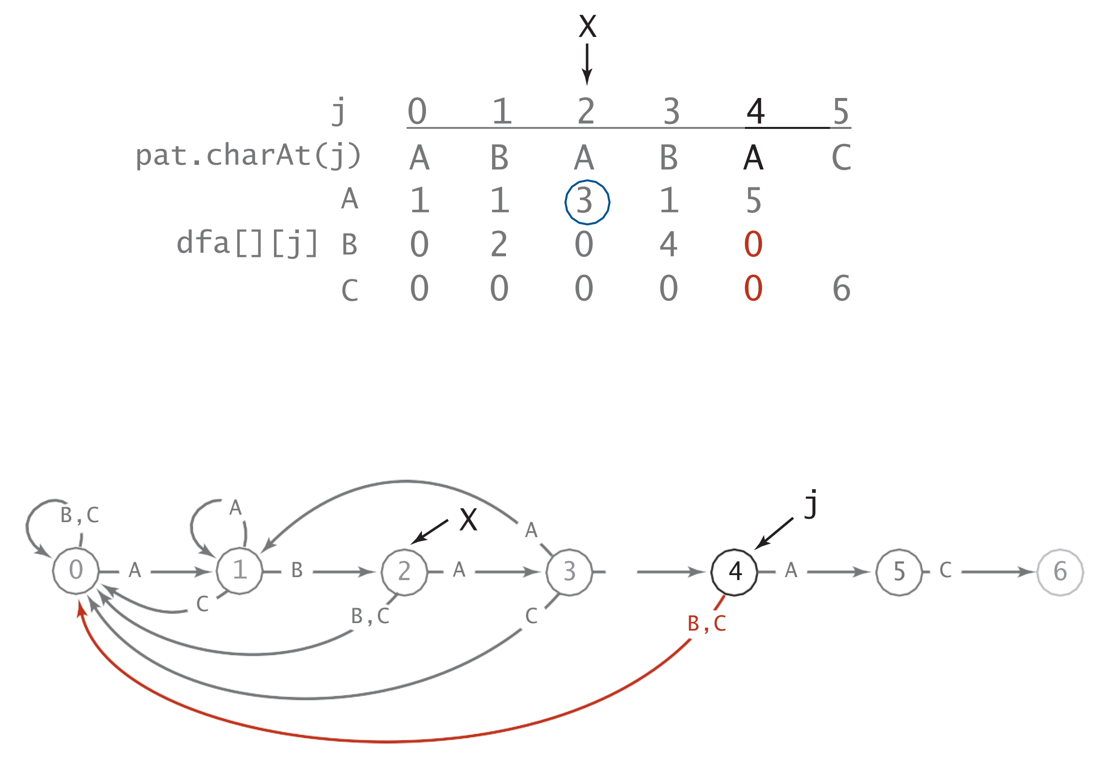


	j	0	1	2	3	4	5
	pat.charAt(j)	A	B	A	B	A	C
	A	1	1	3	1	5	
dfa[][j]	B	0	2	0	4		
	C	0	0	0	0		6

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

Knuth-Morris-Pratt construction

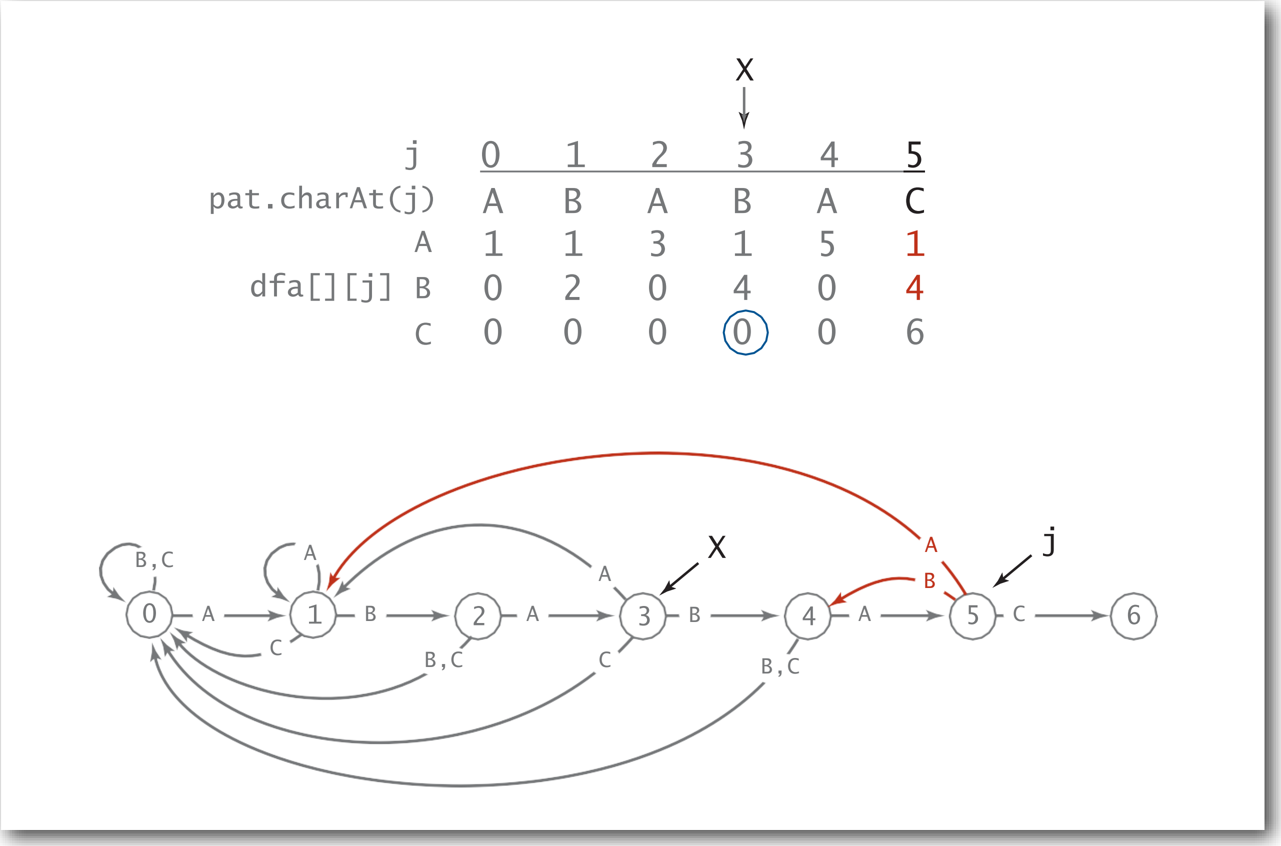
Mismatch transition. For each state `j` and char `c != pat.charAt(j)`, `dfa[c][j] = dfa[c][X]`; then update `X = dfa[pat.charAt(j)][X]`.



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

Knuth-Morris-Pratt construction

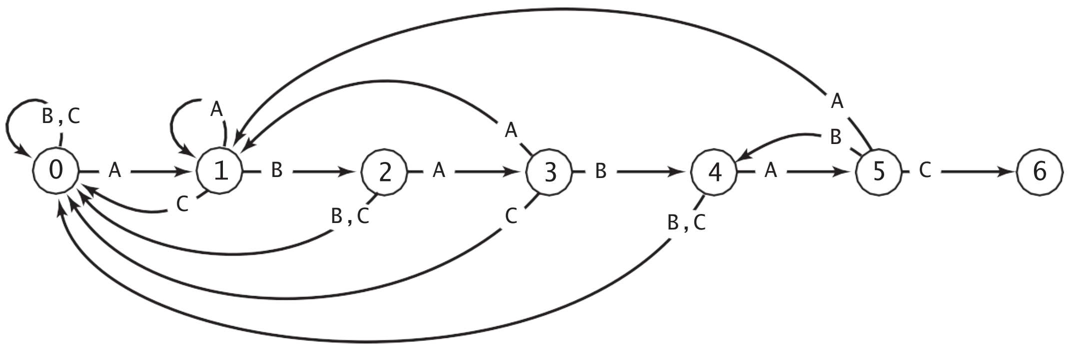
Mismatch transition. For each state `j` and char `c != pat.charAt(j)`, `dfa[c][j] = dfa[c][X]`; then update `X = dfa[pat.charAt(j)][X]`.



j		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 construction

	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



DFA Construction – Java code

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];
   }
}
```

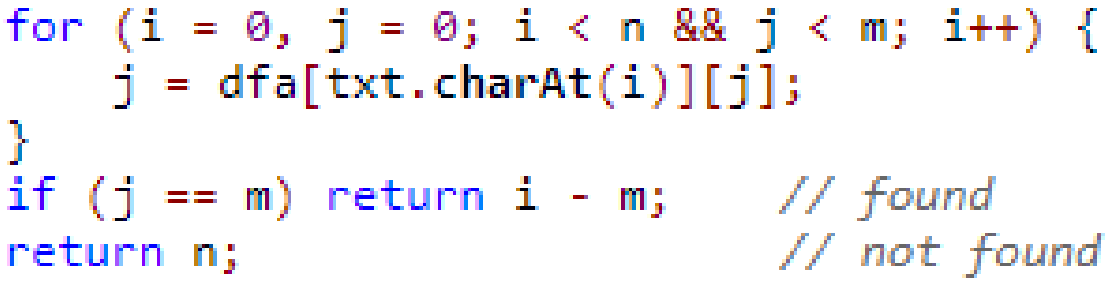
copy mismatch cases

set match case

update restart state

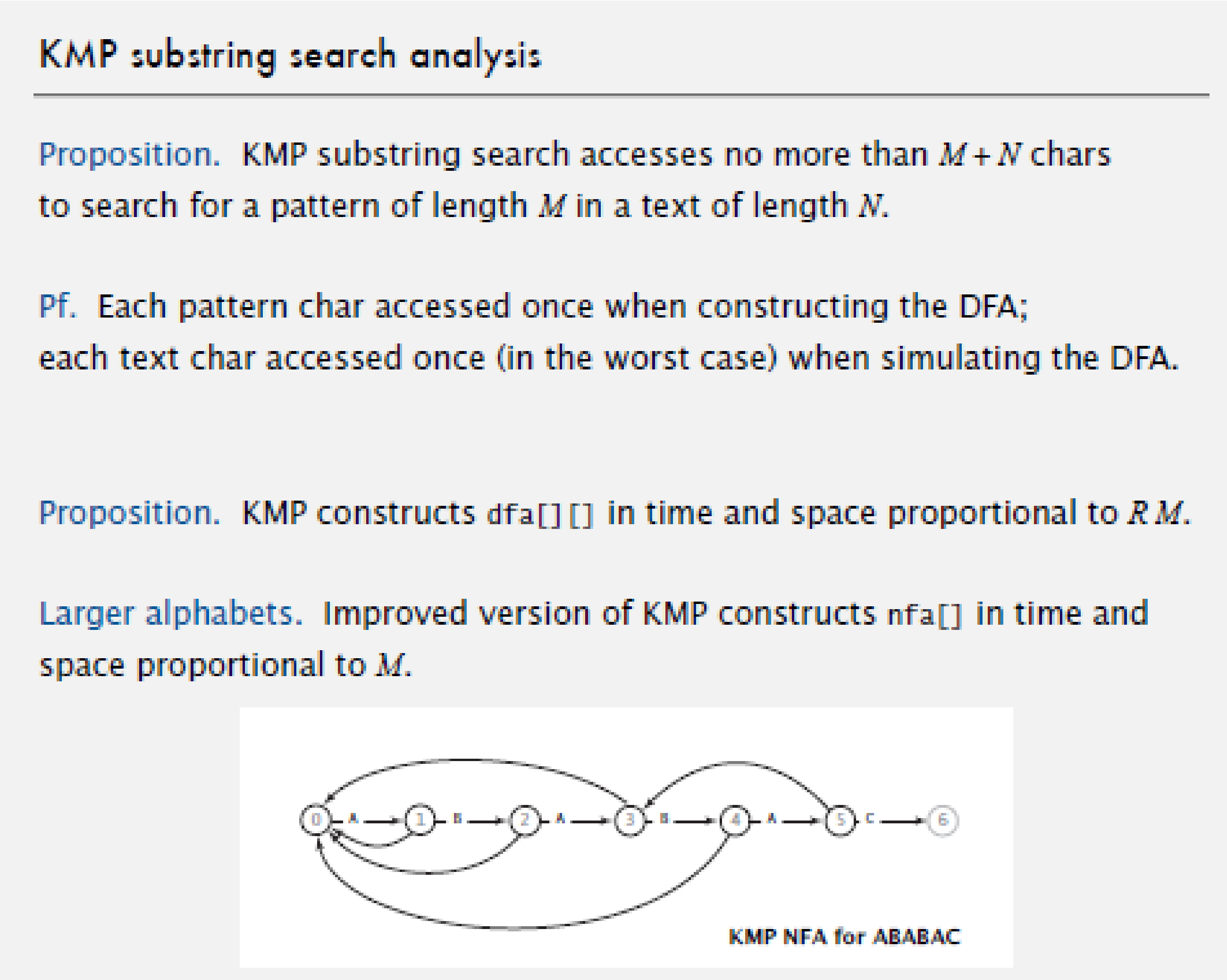
Running time. M character accesses (but space/time proportional to $R\,M$).

KMP search – Java code



```java
for (i = 0, j = 0; i < n && j < m; i++) {
    j = dfa[txt.charAt(i)][j];
}
if (j == m) return i - m;    // found
return n;                    // not found
```

KMP search performance



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