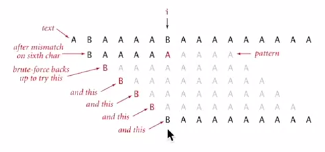
Knuth-Morris-Pratt

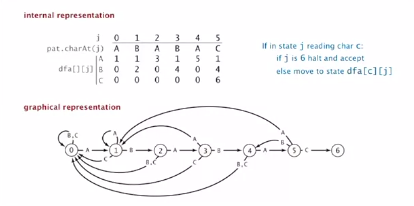
Intuition: Suppose we are searching for pattern BAAAAAAAAA

* Suppose we match 5 chars in pattern, but mismatch on 6th char
* We know previous 6 characters in text are BAAAAB (assuming { A, B } alphabet)
* No need to back up text pointer!

Knuth-Morris-Pratt: clever method to *always* avoid backup.

Deterministic finite state automation (DFA) is abstract string-searching machine

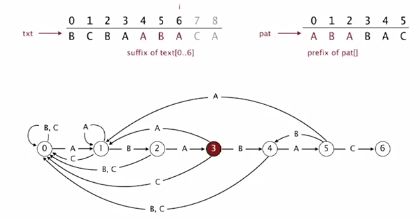
* Finite number of states (including start and halt)
* Exactly one transition for each char in alphabet
* Accept if sequence of transitions leads to halt state



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

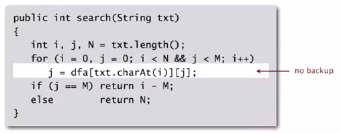
A: State = number of characters in pattern that have been matched.  
 *length of longest prefix of pat[] that is a suffix of txt[0…i]*

Example: DFA is in state 3 after reading in txt[0..6]:



Key difference from brute-force implementation:

* Must precompute DFA from pattern
* Text pointer I never decrements



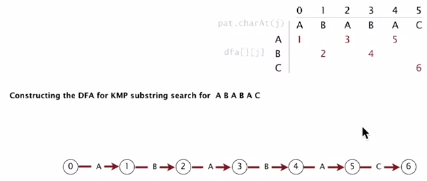
Running time:

* Simulate DFA on text: at most N character accesses… LINEAR!!
* Build DFA: how to do so efficiently?

Constructing the DFA

**Match transition** first

If in state j (first j characters of pattern have already been matched) and next char c == (next char matches) pat.charAt(j), go to j+1 (now first j + 1 characters of pattern have been matched)



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

**Mismatch transition next**

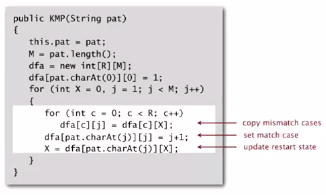
If in state j and next char c != pat.charAt(j), then the last j-1 characters of input are pat[1…j-1], followed by c

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

Running time: takes only constant time if we maintain state X.

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]

Constructing the DFA: 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

Running time: M character accesses (but space/time proportional to R M)

Performance 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.

Proof: 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.