String Searching

Reference: Chapter 19, Algorithms in C, 2nd Edition, Robert Sedgewick.

Robert Sedgewick and Kevin Wayne • Copyright © 2005 • http://www.Princeton.EDU/~cos226

String Search

String search. Given a pattern string, find first match in text.
Model. Can't afford to preprocess the text.

Parameters. N = length of text, M = length of pattern.

↖ typically N >> M

Pattern					
n	e	e	d	l	e

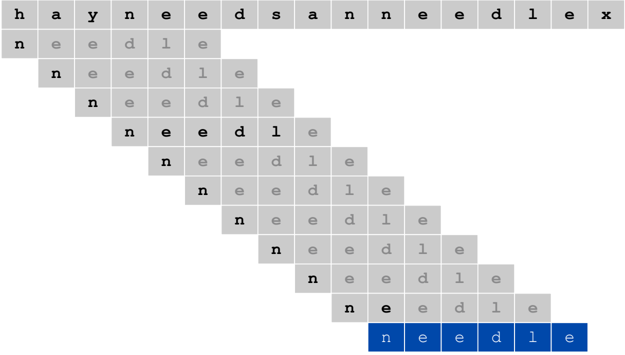
Text																				
i	n	a	h	a	y	s	t	a	c	k	a	n	e	e	d	l	e	i	n	a

M = 6, N = 21

Applications

Applications.
- Parsers.
- Lexis/Nexis.
- Spam filters.
- Virus scanning.
- Digital libraries.
- Screen scrapers.
- Word processors.
- Web search engines.
- Natural language processing.
- Carnivore surveillance system.
- Computational molecular biology.
- Feature detection in digitized images.

Brute Force: Typical Case



h	a	y	n	e	e	d	s	a	n	n	e	e	d	l	e	x
n	e	e	d	l	e											
	n	e	e	d	l	e										
		n	e	e	d	l	e									
			n	**e**	**e**	**d**	**l**	e								
				n	e	e	d	l	e							
					n	e	e	d	l	e						
						n	e	e	d	l	e					
							n	e	e	d	l	e				
								n	e	e	d	l	e			
									n	**e**	e	d	l	e		
										n	e	e	d	l	e	

Brute Force

Brute force. Check for pattern starting at every text position.

```
public static int search(String pattern, String text) {
   int M = pattern.length();
   int N = text.length();

   for (int i = 0; i < N - M; i++) {
      int j;
      for (j = 0; j < M; j++) {
         if (text.charAt(i+j) != pattern.charAt(j))
            break;
      }
      if (j == M) return i;  // return offset i of match
   }
   return -1;                // not found
}
```

Brute Force: Worst Case

```
a a a a a a a a a a a a a a a a b
a a a a a b
  a a a a a b
    a a a a a b
      a a a a a b
        a a a a a b
          a a a a a b
            a a a a a b
              a a a a a b
                a a a a a b
                  a a a a a b
                    a a a a a b
                      a a a a a b
```

Analysis of Brute Force

Analysis of brute force.

- Running time depends on pattern and text.
- Slow if M and N are large, and have lots of repetition.

	character comparisons	
Implementation	Typical	Worst
Brute	1.1 N †	M N

Search for M-character pattern in N-character text

† assumes appropriate model

Screen Scraping

Goal. Find current stock price of Google.

```
...
<tr>
<td class="yfnc_tablehead1" width="48%">
Last Trade:
</td>
<td class="yfnc_tabledata1">
<big><b>475.11</b></big>
</td></tr>
<tr>
<td class="yfnc_tablehead1" width="48%">
Trade Time:
</td>
<td class="yfnc_tabledata1">
11:13AM ET
...
```

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

NYSE symbol

Screen Scraping

Goal. Find current stock price of Google.

- `s.indexOf(t, i)`: index of first occurrence of pattern `t` in string `s`, starting at offset `i`.
- Read raw html from `http://finance.yahoo.com/q?s=goog`.
- Find first string delimited by `<b>` and `</b>` after `Last Trade`.

```
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 input = in.readAll();
      int start    = input.indexOf("Last Trade:", 0);
      int from     = input.indexOf("<b>",  start);
      int to       = input.indexOf("</b>", from);
      String price = input.substring(from + 3, to);
      System.out.println(price);
   }
}
```

```
% java StockQuote goog
475.90
```

Algorithmic Challenges

Theoretical challenge. Linear-time guarantee.

fundamental algorithmic problem

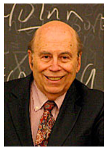
Practical challenge. Avoid backup.

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 theirparty. 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.

Karp-Rabin

Dick Karp
1985 Turing award



Michael Rabin

Karp-Rabin Randomized Fingerprint Algorithm

Idea: use hashing.

- Compute hash function for each text position.
- No explicit hash table: just compare with pattern hash!

Ex. Hash "table" size = 97.

Pattern				
5	9	2	6	5

`59265 % 97 = 95`

Text																					
3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3	2	3	8	4	6	
3	1	4	1	5																	`31415 % 97 = 84`
	1	4	1	5	9																`14159 % 97 = 94`
		4	1	5	9	2															`41592 % 97 = 76`
			1	5	9	2	6														`15926 % 97 = 18`
				5	9	2	6	5													`59265 % 97 = 95`

Computing the Hash Function

Brute force. O(M) arithmetic ops per hash.

Faster method to compute hash of adjacent substrings.

- Use previous hash to compute next hash.
- O(1) time per hash.

except first one

Ex.

- Pre-computed: `10000 % 97 =  9`
- Previous hash: `41592 % 97 = 76`
- Next hash: `15926 % 97 = ??`

key property of mod: can mod out any time

Observation.

```
15926 % 97   ≡ (41592 – (4 * 10000)) * 10   +   6
             ≡ (76    – (4 * 9    )) * 10   +   6
             ≡ 406
             ≡ 18
```

Java Implementation

```
public static int search(String p, String t) {
   int M = p.length(), N = t.length();
   int q = 8355967;                          // table size
   int d = 256;                              // radix

   int dM = 1;                               // precompute d^(M-1) % q
   for (int j = 1; j < M; j++)
      dM = (d * dM) % q;

   int h1 = 0, h2 = 0;
   for (int i = 0; i < M; j++) {
      h1 = (h1*d + p.charAt(i)) % q;         // hash of pattern
      h2 = (h2*d + t.charAt(i)) % q;         // hash of text
   }
   if (h1 == h2) return 0;                   // match found

   for (int i = M; i < N; i++) {
      h2 = (h2 + d*q - dM*t.charAt(i-M)) % q; // remove leftmost digit
      h2 = (h2*d + t.charAt(i)) % q;           // insert rightmost digit
      if (h1 == h2) return i - M + 1;          // match found
   }
   return -1;                                  // not found
}
```

Karp-Rabin: False Matches

False match. Hash of pattern collides with another substring.

- `59265 % 97 = 95`
- `59362 % 97 = 95`

How to choose modulus p.

- p too small ⇒ many false matches.
- p too large ⇒ too much arithmetic.
- Ex: p = 8355967 ⇒ avoid 32-bit integer overflow.
- Ex: p = 35888607147294757 ⇒ avoid 64-bit integer overflow.

Karp-Rabin: Randomized Algorithms

Theorem. If $MN \geq 29$ and p is a random prime between 1 and MN^2, then Pr[false match] $\leq 2.53/N$.

relies on prime number theorem

Randomized algorithm. Choose table size p at random to be huge prime.

Monte Carlo. Don't bother checking for false matches.

- Guaranteed to be fast: O(M + N).
- Expected to be correct (but false match possible).

Las Vegas. Upon hash match, do full compare; if false match, try again with new random prime.

- Guaranteed to be correct.
- Expected to be fast: O(M + N).

Q. Would either version of Rabin-Karp make a good library function?

String Search Implementation Cost Summary

Karp-Rabin summary.

- Create fingerprint of each substring and compare fingerprints.
- Expected running time is linear.
- Idea generalizes, e.g., to 2D patterns.

	character comparisons	
Implementation	Typical	Worst
Brute	$1.1\,N$ †	$M\,N$
Karp-Rabin	$\Theta(N)$	$\Theta(N)$ ‡

Search for M-character pattern in N-character text
† assumes appropriate model
‡ randomized

Knuth-Morris-Pratt

Don Knuth
1974 Turing award

Jim Morris

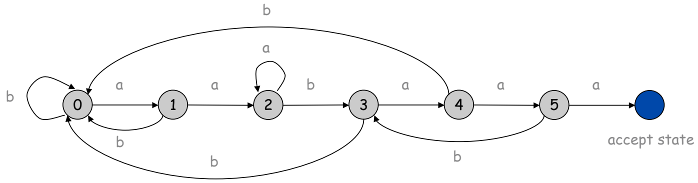
Vaughan Pratt

Knuth-Morris-Pratt: DFA Simulation

KMP algorithm. [over binary alphabet]

- Build DFA from pattern.
- Run DFA on text.

Text										
a	a	a	b	a	a	b	a	a	a	b
a	a	b	a	a	a					
	a	a	b	a	a	a				
				a	a	b	a	a	a	

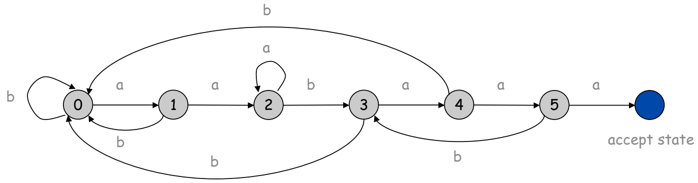


Knuth-Morris-Pratt: DFA Simulation

Interpretation of state i. Length of longest prefix of search pattern that is a suffix of input string.

Ex. End in state 4 iff text ends in `aaba`.
Ex. End in state 2 iff text ends in `aa` (but not `aabaa` or `aabaaa`).

Pattern					
a	a	b	a	a	a



DFA Representation

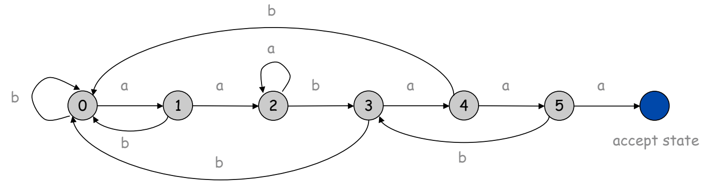
DFA used in KMP has special property.

- Upon character match in state `j`, go forward to state `j+1`.
- Upon character mismatch in state `j`, go back to state `next[j]`.

Pattern					
a	a	b	a	a	a

	0	1	2	3	4	5
a	1	2	2	4	5	6
b	0	0	3	0	0	3
next	0	0	2	0	0	3

← only need to store this row



accept state

KMP Algorithm

Two key differences from brute force.

- Text pointer `i` never "backs up."
- Need to precompute `next[]` table.

```
int j = 0;
for (int i = 0; i < N; i++) {
   if (t.charAt(i) == p.charAt(j)) j++;    // match
   else j = next[j];                       // mismatch
   if (j == M) return i - M + 1;           // found
}
return -1;                                 // not found
```

Simulation of KMP DFA (assumes binary alphabet)

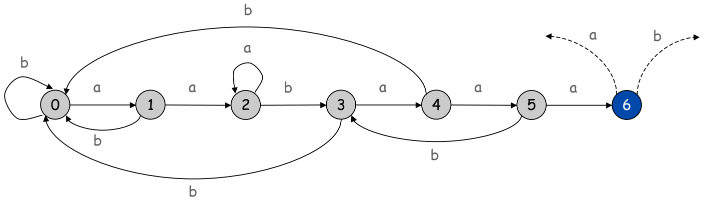
Knuth-Morris-Pratt: DFA Construction

Iterative construction. Suppose you've created DFA for pattern `aabaaa`.
How to extend to DFA for pattern `aabaaab` ?

- Easy: transition from state 6 if next char matches.
- Challenge: transition from state 6 if next char mismatches.

Wishful thinking. Simulate `aabaaaa` on DFA.

Key idea. Simulate ~~a~~`abaaaa` on DFA.



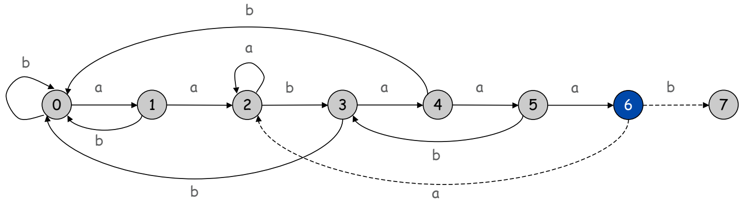
Knuth-Morris-Pratt: DFA Construction

Iterative construction. Suppose you've created DFA for pattern `aabaaa`.
How to extend to DFA for pattern `aabaaab` ?

- Easy: transition from state 6 if next char matches.
- Challenge: transition from state 6 if next char mismatches.

Wishful thinking. Simulate `aabaaaa` on DFA.

Key idea. Simulate ~~a~~`abaaaa` on DFA.
Efficient version. Pre-compute simulation of ~~a~~`abaaa`.



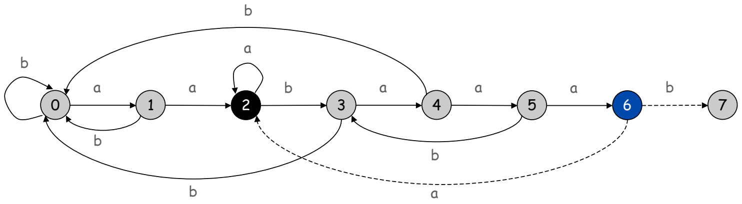
Knuth-Morris-Pratt: DFA Construction

DFA construction for KMP. DFA builds itself!

State 6. Given DFA for `aabaaa` and state X of simulating `~~a~~abaaa`, compute DFA for `~~a~~abaaab` and state X of simulating `~~a~~abaaab`.

X = 2

- `next[6]` = X → `a` = 2.
- Update X = X → `b` = 3.



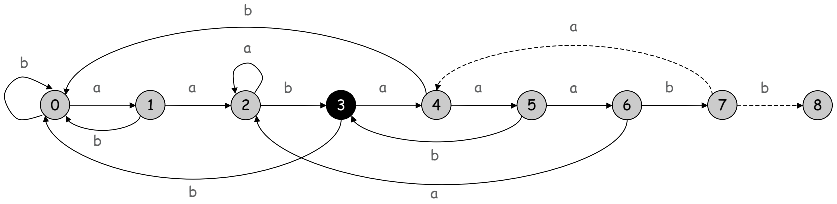
DFA Construction for KMP

DFA construction for KMP. DFA builds itself!

State 7. Given DFA for `aabaaab` and state X of simulating `~~a~~abaaab`, compute DFA for `~~a~~abaaabb` and state X of simulating `~~a~~abaaabb`.

X = 3

- `next[7]` = X → `a` = 4.
- Update X = X → `b` = 0.



DFA Construction for KMP: Java Implementation

Build DFA for KMP.

- Takes O(M) time.
- Requires O(M) extra space to store `next[]` table.

```
int X = 0;
int[] next = new int[M];
for (int j = 1; j < M; j++) {
   if (p.charAt(X) == p.charAt(j)) {  // char match
      next[j] = next[X];
      X = X + 1;
   }
   else {                             // char mismatch
      next[j] = X + 1;
      X = next[X];
   }
}
```

DFA Construction for KMP (assumes binary alphabet)

Optimized KMP Implementation

Ultimate search program for `aabaaabb` pattern.

- Specialized C program.
- Machine language version of C program.

```
int kmpearch(char t[]) {
   int i = 0;
   s0: if (t[i++] != 'a') goto s0;
   s1: if (t[i++] != 'a') goto s0;
   s2: if (t[i++] != 'b') goto s2;
   s3: if (t[i++] != 'a') goto s0;
   s4: if (t[i++] != 'a') goto s0;
   s5: if (t[i++] != 'a') goto s3;
   s6: if (t[i++] != 'b') goto s2;
   s7: if (t[i++] != 'b') goto s4;
   return i - 8;            ↑            ↑
}                       pattern[]     next[]
```

assumes pattern is in text (o/w use sentinel)

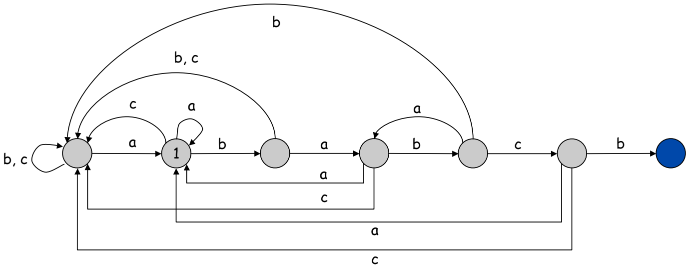
KMP Over Arbitrary Alphabet

DFA for patterns over arbitrary alphabet Σ.

- For each character in alphabet, determine next state.
- Lookup table requires O(M |Σ|) space.

can be expensive if Σ = Unicode alphabet

Ex. DFA for pattern `ababcb`.

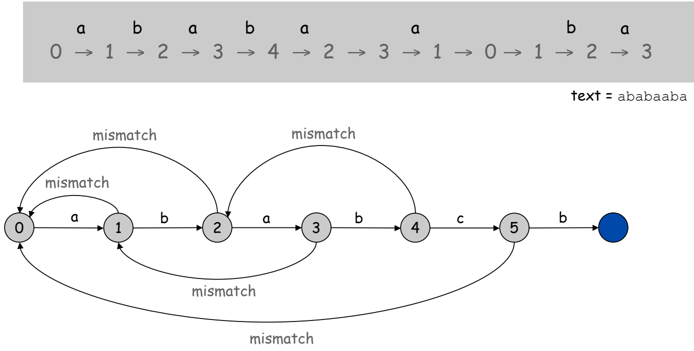


KMP Over Arbitrary Alphabet

NFA for patterns over arbitrary alphabet Σ.

- Read new character only upon success (or failure at beginning).
- Reuse current character upon failure and follow back.

Ex. NFA for pattern `ababcb`.



String Search Implementation Cost Summary

KMP analysis.

- NFA simulation requires at most 2N comparisons.
  - advances ≤ N
  - retreats ≤ advances
- NFA construction takes Θ(M) time and space.

	character comparisons	
Implementation	Typical	Worst
Brute	1.1 N †	M N
Karp-Rabin	Θ(N)	Θ(N) ‡
KMP	1.1 N †	2 N

Search for M-character pattern in N-character text
† assumes appropriate model
‡ randomized

History of KMP

History of KMP.

- Inspired by esoteric theorem of Cook that says linear time algorithm should be possible for 2-way pushdown automata.
- Discovered in 1976 independently by two theoreticians and a hacker.
  - Knuth: discovered linear time algorithm
  - Pratt: made running time independent of alphabet
  - Morris: trying to build a text editor.

Resolved theoretical and practical problems.

- Surprise when it was discovered.
- In hindsight, seems like right algorithm.

Boyer-Moore

Bob Boyer

J. Strother Moore

Right-to-Left Scanning

Right-to-left scanning.

- Find offset `i` in text by moving left to right.
- Compare pattern to text by moving `j` right to left.

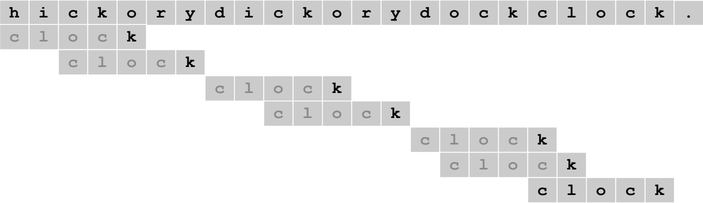
```
h i c k o r y , d i c k o r y , d o c k , c l o c k .
c l o c k
  c l o c k
    c l o c k
      c l o c k
```

Bad Character Rule

Bad character rule.

- Use right-to-left scanning.
- Upon mismatch of text character `c`, increase offset so that character `c` in pattern lines up with text character `c`.
- Precompute `right[c]` = rightmost occurrence of `c` in pattern.

right[]	
c	3
k	4
l	1
o	2
*	-1



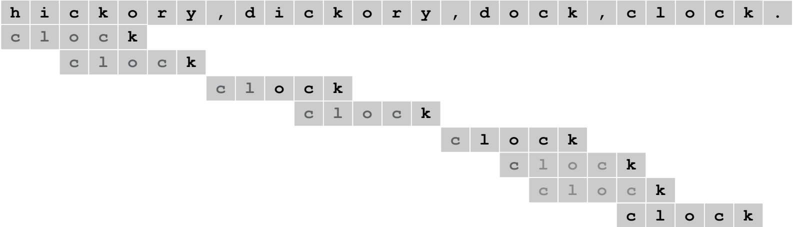
```
h i c k o r y d i c k o r y d o c k c l o c k .
c l o c k
    c l o c k
              c l o c k
                  c l o c k
                            c l o c k
                              c l o c k
                                    c l o c k
```

Bad Character Rule

Bad character rule.

- Use right-to-left scanning.
- Upon mismatch of text character `c`, increase offset so that character `c` in pattern lines up with text character `c`.
- Precompute `right[c]` = rightmost occurrence of `c` in pattern.

right[]	
c	3
k	4
l	1
o	2
*	-1



```
h i c k o r y , d i c k o r y , d o c k , c l o c k .
c l o c k
    c l o c k
              c l o c k
                    c l o c k
                              c l o c k
                                  c l o c k
                                    c l o c k
                                          c l o c k
```

Bad Character Rule: Java Implementation

```
public static int search(String pattern, String text) {
   int M = pattern.length(), N = text.length();
   int[] right = new int[256];
   for (int c = 0; c < 256; c++) right[c] = -1;
   for (int j = 0; j < M;   j++) right[pattern.charAt(j)] = j;

   int i = 0; // offset
   while (i < N - M) {
      int skip = 0;
      for (int j = M-1; j >= 0; j--) {
         if (pattern.charAt(j) != text.charAt(i + j)) {
            skip = Math.max(1, j - right[text.charAt(i + j)]);
            break;
         }
      }
      if (skip == 0) return i;  // found
      i = i + skip;
   }
   return -1;
}
```

rightmost occurrence of c in pattern

bad character rule

Bad Character Rule: Analysis

Bad character rule analysis.

- Highly effective in practice, particularly for English text: O(N / M).
- Takes Ω(MN) time in worst case.

```
a a a 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 a
  b a a a a a a
    b a a a a a a
      b a a a a a a
        b a a a a a a
          b a a a a a a
            b a a a a a a
              b a a a a a a
```

Strong Good Suffix Rule

Strong good suffix rule. [a KMP-like suffix rule]

- Right-to-left scanning.
- Suppose text matches suffix t of pattern but mismatches in previous character `c`.
- Find rightmost copy of t in pattern whose preceding letter is not `c`, and shift; if no such copy, shift M positions.

```
t = "ab"
c = 'b'
```

```
x x x x x x x b a b ? ? ? ? ? ? x x x x x x x x
x c a b d a b d a b
            x c a b d a b d a b
```

string good suffix rule: can skip over this since we already know dab doesn't match

bad character rule: skip only 1 position

Boyer-Moore

Boyer-Moore.

- Right-to-left scanning.
- Bad character rule.
- Strong good suffix rule.

always take best of two shifts

Boyer-Moore analysis.

- O(N / M) average case if given letter usually doesn't occur in string.
  - time decreases as pattern length increases
  - sublinear in input size!
- At most 3N comparisons to find a match.

Boyer-Moore in the wild. Unix grep, emacs.

String Search Implementation Cost Summary

Implementation	Typical	Worst
Brute	1.1 N †	M N
Karp-Rabin	Θ(N)	Θ(N) ‡
KMP	1.1 N †	2N
Boyer-Moore	N / M †	3N

Search for M-character pattern in N-character text

† assumes appropriate model
‡ randomized

Boyer-Moore and Alphabet Size

Boyer-Moore space requirement. Θ(M + |Σ|)

Big alphabets.

- Direct implementation may be impractical, e.g., Unicode.
- Fix: search one byte at a time.

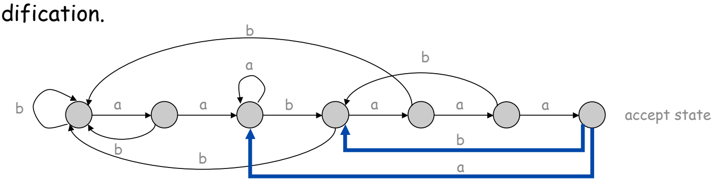
Small alphabets.

- Loses effectiveness when Σ is too small, e.g., DNA.
- Fix: group characters together, e.g., `aaaa`, `aaac`, ….

Finding All Matches

Karp-Rabin. Can find all matches in O(M + N) expected time using Muthukrishnan variant.

Knuth-Morris-Pratt. Can find all matches in O(M + N) time via simple modification.



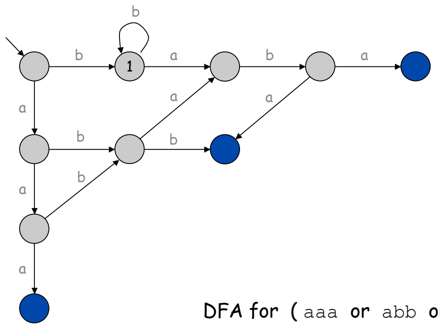
search pattern: `aabaaa`

Boyer-Moore. Can find all matches in O(M + N) time using Galil variant.

Multiple String Search

Multiple string search. Search for any of k different patterns.

- Naïve KMP: O(kN + M_1 + … + M_k).
- Aho-Corasick: O(N + M_1 + … + M_k).
- Ex: screen out dirty words from a text stream.



DFA for (`aaa` or `abb` or `baba`)