String Searching

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

String Search

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

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

typically N >> M

Search Pattern					
n	e	e	d	l	e

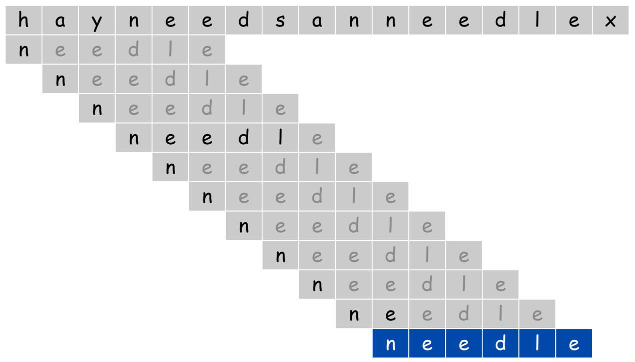
M = 6, N = 21

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

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



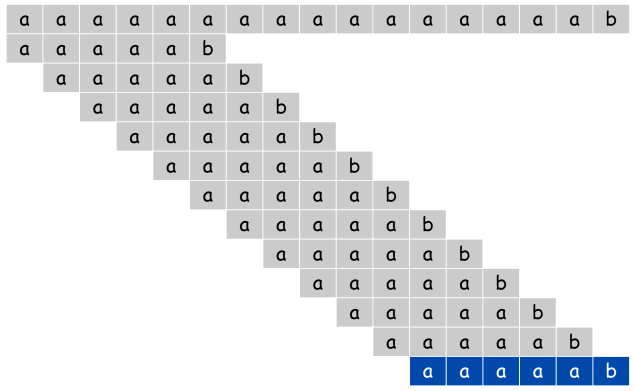
Brute Force

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 if found
   }
   return -1;                  ← return -1 if not found
}
```

Brute Force: Worst Case



Analysis of Brute Force

Analysis of brute force.

- Running time depends on pattern and text.
- Worst case: M N comparisons.
- "Average" case: 1.1 N comparisons. (!)
- Slow if M and N are large, and have lots of repetition.

Screen Scraping

Find current stock price of Google.

- `t.indexOf(p)`: index of 1st occurrence of pattern p in text t.
- Download html from: `http://finance.yahoo.com/q?s=goog`
- Find first string delimited by `<b>` and `</b>` appearing after `Last Trade`

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

```
% java StockQuote goog
357.36
```

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

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.

Search Pattern				
5	9	2	6	5

`59265 % 97 = 95`

Search 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 = ??`

Observation.
key property of mod: can mod out any time

```
▪ 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.

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

relies on prime number theorem

Karp-Rabin: Randomized Algorithms

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

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

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

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

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

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	Θ(N)	Θ(N) ‡

Search for M-character pattern in N-character text

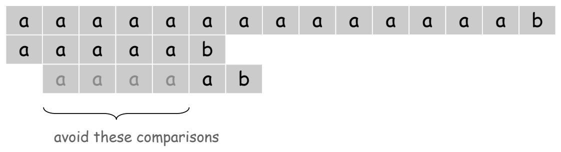
† assumes appropriate model
‡ randomized

Knuth-Morris-Pratt

How To Save Comparisons

How to avoid re-computation?

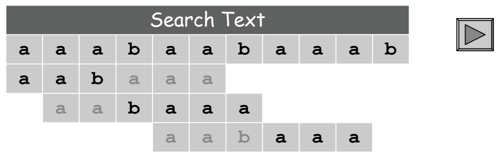
- Pre-analyze search pattern.
- Ex: suppose that first 5 characters of pattern `p` are all a's.
  - if `t[0..4]` matches `p[0..4]`, then `t[1..4]` matches `p[0..3]`
  - no need to check i = 1, j = 0, 1, 2, 3
  - saves 4 comparisons

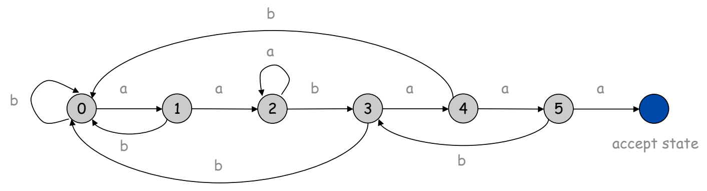


Knuth-Morris-Pratt: DFA Simulation

KMP algorithm. [over binary alphabet]

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



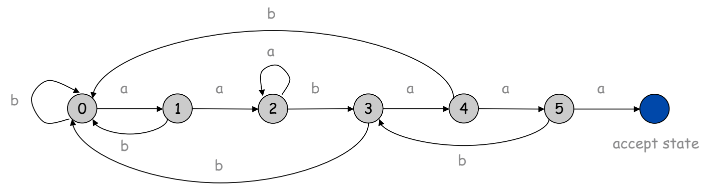


Knuth-Morris-Pratt: DFA Simulation

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`).

Search Pattern
a a b a a a



DFA Representation

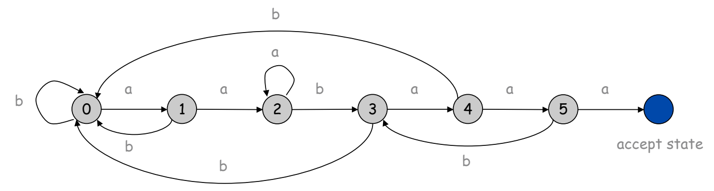
DFA used in KMP has special property.

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

Search 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 store this row



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

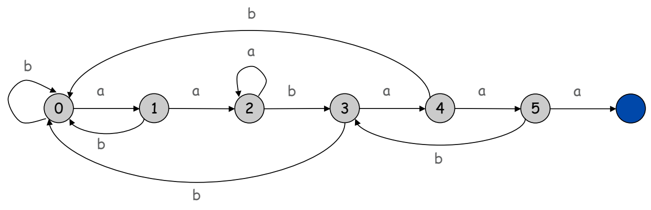
KMP algorithm. [over binary alphabet]

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

Rule for creating `next[]` table for pattern `aabaaa`.

- `next[4]`: longest prefix of `aabaa` that is a suffix of `aaba`**`b`**.
- `next[5]`: longest prefix of `aabaaa` that is a suffix of `aabaa`**`b`**.

↑
compute by simulating `abaab` on DFA

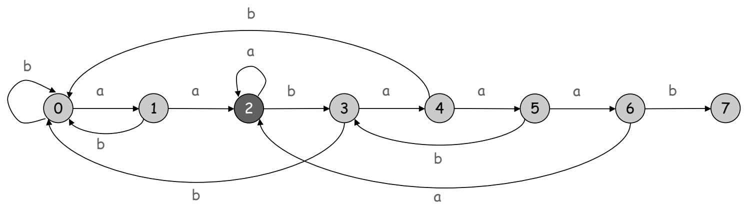


DFA Construction for KMP

DFA construction for KMP. DFA builds itself!

Ex. Compute `next[6]` for pattern `p[0..6] = aabaaa`**`b`**.

- Assume you know DFA for pattern `p[0..5] = aabaaa`.
- Assume you know state X for `p[1..5] = abaaa`. X = 2
- Update `next[6]` to state for `abaaa`**`a`**. X + **a** = 2
- Update X to state for `p[1..6] = abaaa`**`b`** X + **b** = 3

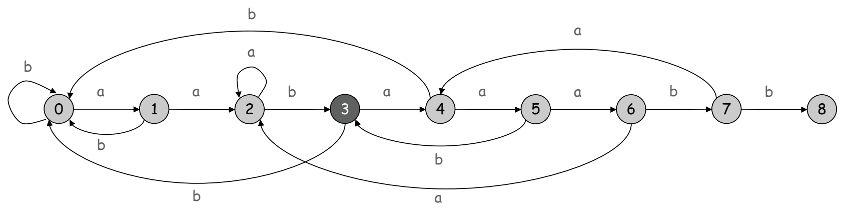


DFA Construction for KMP

DFA construction for KMP. DFA builds itself!

Ex. Compute `next[7]` for pattern `p[0..7] = aabaaabb`.

- Assume you know DFA for pattern `p[0..6] = aabaaab`.
- Assume you know state X for `p[1..6] = abaaab`. X = 3
- Update `next[7]` to state for `abaaaba`. X + **a** = 4
- Update X to state for `p[1..7] = abaaabb` X + **b** = 0



DFA Construction for KMP

DFA construction for KMP. DFA builds itself!

Crucial insight.

- To compute transitions for state n of DFA, suffices to have:
  - DFA for states 0 to n-1
  - state X that DFA ends up in with input `p[1..n-1]`
- To compute state X' that DFA ends up in with input `p[1..n]`, it suffices to have:
  - DFA for states 0 to n-1
  - state X that DFA ends up in with input `p[1..n-1]`

DFA Construction for KMP: 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;  ⎬ next[]
   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;
}
```

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

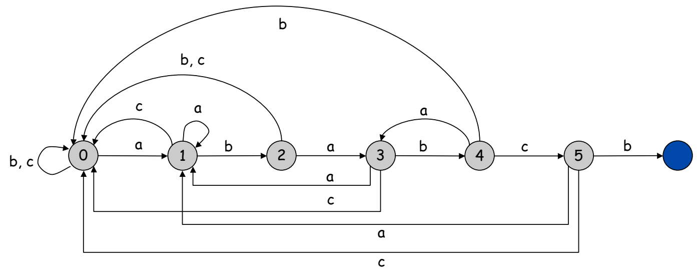
KMP Over Arbitrary Alphabet

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`.



29

KMP Over Arbitrary Alphabet

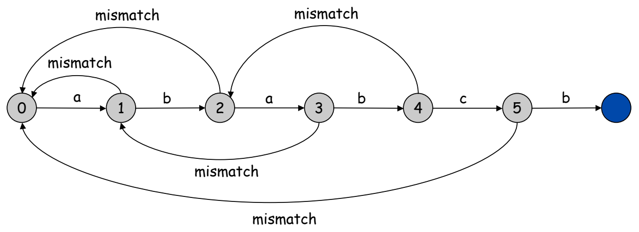
NFA for patterns over arbitrary alphabet Σ.

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

Ex. NFA for pattern `ababcb`.

$$0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{a} 3 \xrightarrow{b} 4 \xrightarrow{a} 2 \rightarrow 3 \xrightarrow{a} 1 \rightarrow 0 \rightarrow 1 \xrightarrow{b} 2 \xrightarrow{a} 3$$

text = `abababa`



30

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.
- Good efficiency for patterns and texts with much repetition.

	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

31

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 an editor and avoid annoying buffer for string search

Resolved theoretical and practical problems.

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

32

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;
}
```

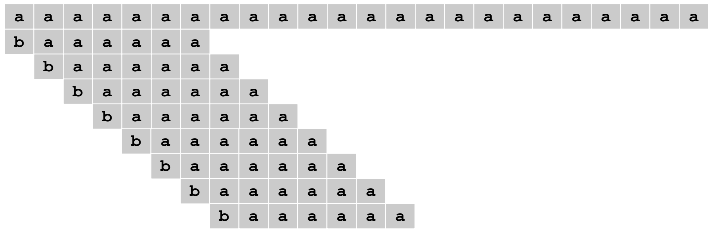
rightmost occurrence of c in pattern

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.

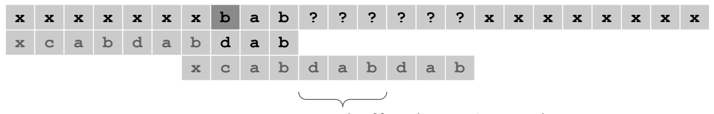


Strong Good Suffix Rule

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'
```



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. $\Theta(M + |\Sigma|)$

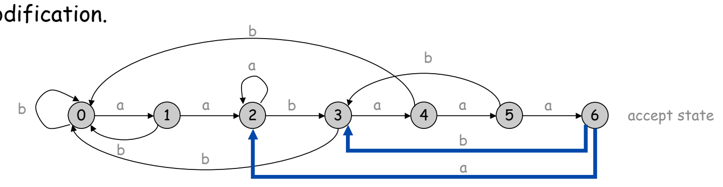
Big alphabets.
- Direct implementation may be impractical, e.g., UNICODE.
- Fix: search one byte at a time.

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.

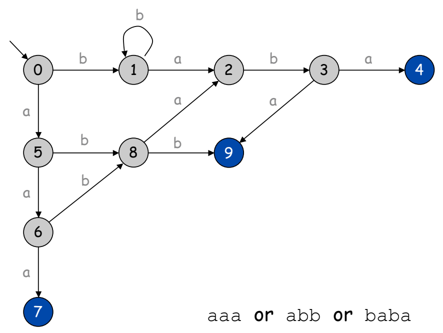


search pattern: `aabaaa`

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: $O(kN + M_1 + \ldots + M_k)$.
- Aho-Corasick: $O(N + M_1 + \ldots + M_k)$.
- Ex: screen out dirty words from a text stream.



`aaa` or `abb` or `baba`

Spam Filtering

Spam filtering. Identify patterns indicative of spam.

- PROFITS
- AMAZING
- GUARANTEE
- herbal Viagra
- There is no catch.
- This is a one-time mailing.
- This message is sent in compliance with spam regulations.
- You're getting this message because you registered with one of our marketing partners.

Tip of the Iceberg

Wildcards / character classes.

- O(M + N) time using O(M + |Σ|) extra space.
- Ex: PROSITE patterns for computational biology.

Approximate string matching: allow up to k mismatching chars.

- Ex: fix transmission errors in signal processing.
- Ex: recover from typing or spelling errors in information retrieval.

Edit-distance: allow up to k edits.

- Recover from measurement errors in computational biology.

Java String Library

Java String library has built-in string searching.

- `t.indexOf(p)`: index of 1st occurrence of pattern `p` in text `t`.
- Caveat: it's brute force, and can take Ω(MN) time.

```
public static void main(String[] args) {
   int n = Integer.parseInt(args[0]);

   String s = "a";
   for (int i = 0; i < n; i++)
      s = s + s;

   String pattern = s + "b";
   String text    = s + s;

   System.out.println(text.indexOf(pattern));
}
```

$\underbrace{\texttt{aaa ... a}}_{2^n}$

$\overbrace{\texttt{aaa ... ab}}$

$\underbrace{\texttt{aaa ... aaaa ... a}}_{2^{n+1}}$

Q. Why does Java string library use brute force?

String Search Summary

Ingenious algorithms for a fundamental problem.

Rabin-Karp.

- Easy to implement, but usually worse than brute-force.
- Extends to more general settings (e.g., 2D search).

Knuth-Morris-Pratt.

- Quintessential solution to theoretical problem.
- Extends to more general settings (e.g., multiple string search).

Boyer-Moore.

- Simple idea leads to dramatic speedup for long patterns.
- Running time depends on alphabet size.
- Need to tweak for small or large alphabets.