String Searching

Karp-Rabin

Knuth-Morris-Pratt

Boyer-Moore

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.

N = # characters in text
M = # characters in pattern

typically N >> M
Ex: N = 1 million, M = 100

Search Text																				
n	n	e	e	n	l	e	d	e	n	e	e	n	e	e	d	l	e	n	l	d

Search Pattern					
n	e	e	d	l	e

M = 21, N = 6

Successful Search																				
n	n	e	e	n	l	e	d	e	n	e	e	**n**	**e**	**e**	**d**	**l**	**e**	n	l	d

String Search

String: Sequence of characters over some alphabet.
Ex alphabets: binary, decimal, ASCII, UNICODE, DNA.

Some applications.

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

Spam Filtering

Spam filtering: patterns indicative of spam.

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

Brute Force

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

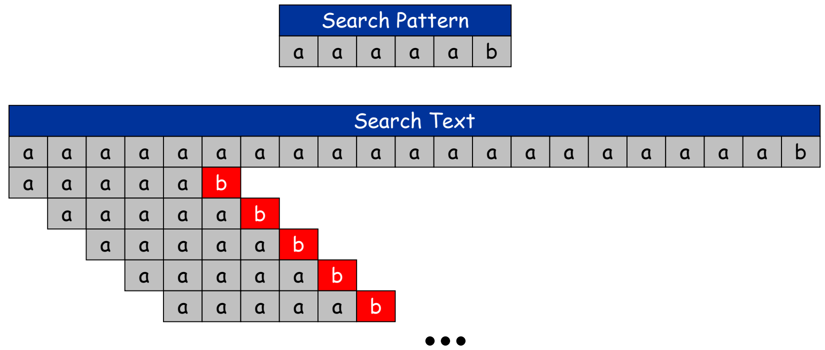
```java
public static int search(String pattern, String text) {
   int M = pattern.length();
   int N = text.length();

   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
}
```


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 Sun Microsystems.

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

```java
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 sunw
5.20

% java StockQuote ibm
96.84
```


Algorithmic Challenges

Theoretical challenge: linear-time guarantee.

- TST index costs ~ N lgN.

Practical challenge: avoid BACKUP.

- Often no room or time to save text.

Fundamental algorithmic problem.

```
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 manygood
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 manygood
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 Fingerprint Algorithm

Idea: use hashing.

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

Example.

- 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																
	1	4	1	5	9															
		4	1	5	9	2														
			1	5	9	2	6													
				5	9	2	6	5												

```
31415 % 97 = 84
14159 % 97 = 94
41592 % 97 = 76
15926 % 97 = 18
59265 % 97 = 95
```


Karp-Rabin Fingerprint Algorithm

Idea: use hashing.

- Compute hash function for each text position.

Guaranteeing correctness.

- Need full compare on hash match to guard against collisions.
  - `59265 % 97 = 95`
  - `59362 % 97 = 95`

Running time.

- Hash function depends on M characters.
- Running time is $\Theta(MN)$ for search miss.

how can we fix this?

Karp-Rabin Fingerprint Algorithm

Key idea: fast to compute hash function of adjacent substrings.

- Use previous hash to compute next hash.
- O(1) time per hash, except first one.

Example.

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

Observation.

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


Karp-Rabin Fingerprint Algorithm : Java Implementation

```java
public static int search(String p, String t) {
   int M = p.length();
   int N = t.length();
   int dM = 1, h1 = 0, h2 = 0;
   int q = 3355439;                     // table size
   int d = 256;                         // radix

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

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

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

String Search Implementation Cost Summary

Karp-Rabin fingerprint algorithm.

- Choose table size at random to be huge prime.
- Expected running time is O(M + N).
- Θ(MN) worst-case, but this is (unbelievably) unlikely.

Main advantage. Extends to 2d patterns and other generalizations.

Search for an M-character pattern in an N-character text.

Implementation	Typical	Worst
Brute	1.1 N †	M N
Karp-Rabin	Θ(N)	Θ(N) ‡

† assumes appropriate model
‡ randomized

character comparisons

Randomized Algorithms

A randomized algorithm uses random numbers to gain efficiency.

Las Vegas algorithms.

- Expected to be fast.
- Guaranteed to be correct.
- Ex: quicksort, randomized BST, Rabin-Karp with match check.

Monte Carlo algorithms.

- Guaranteed to be fast.
- Expected to be correct.
- Ex: Rabin-Karp without match check.

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

How To Save Comparisons

How to avoid re-computation?

- Pre-analyze search pattern.
- Ex: suppose that first 5 characters of pattern 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

Basic strategy: pre-compute something based on pattern.

Search Pattern					
a	a	a	a	a	b

Search Text																				
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															
	a	a	a	a	a	b														

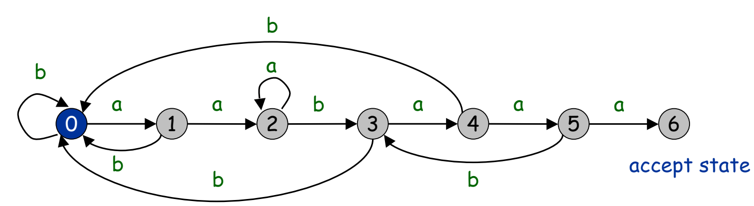
Knuth-Morris-Pratt (over binary alphabet)

KMP algorithm.

- Use knowledge of how search pattern repeats itself.
- Build DFA from pattern.
- Run DFA on text.

Search Pattern					
a	a	b	a	a	a

Search Text										
a	a	a	b	a	a	b	a	a	a	b



DFA Representation

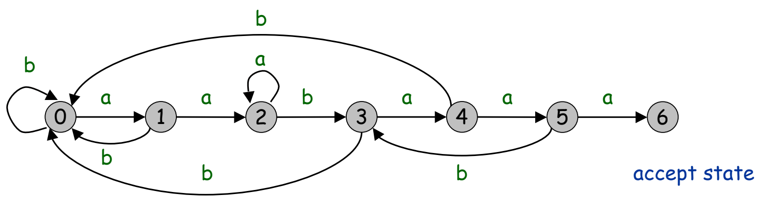
DFA used in KMP has special property.

- Upon character match, go forward one state.
- Only need to keep track of where to go upon character mismatch: go to state `next[j]` if character mismatches in state `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.

```
for (int i = 0, j = 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

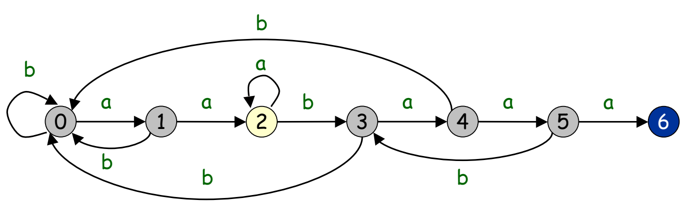
KMP algorithm. (over binary alphabet, for simplicity)

- Use knowledge of how search pattern repeats itself.
- ➡ 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 proper suffix of `aaba`**b**.
  `next[5]`: longest prefix of `aabaaa` that is a proper suffix of `aabaa`**b**.

compute by simulating `abaab` on DFA

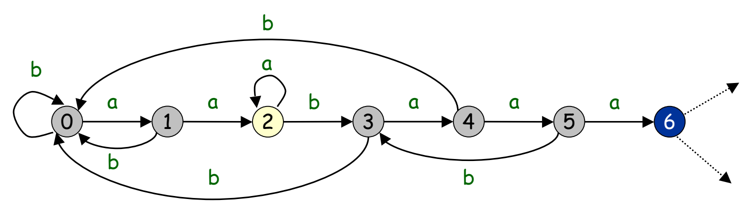


DFA Construction for KMP

DFA construction for KMP. DFA builds itself!

Ex: compute `next[6]` for pattern `p[0..6] = aabaaab`.

- 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

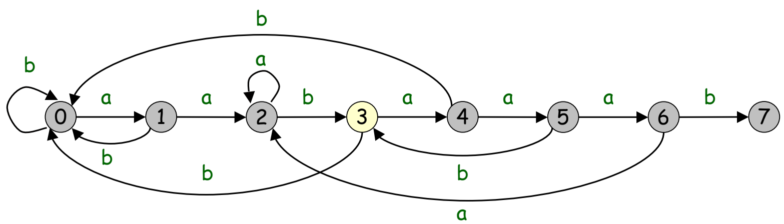


DFA Construction for KMP

DFA construction for KMP. DFA builds itself!

Ex: compute `next[6]` for pattern `p[0..6] = aabaaab`.

- 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

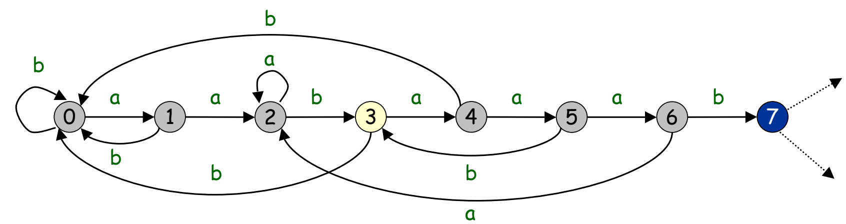


DFA Construction for KMP

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 `abaaab`**a**. X + **a** = 4
- Update X to state for `p[1..7] = abaaab`**b** X + **b** = 0

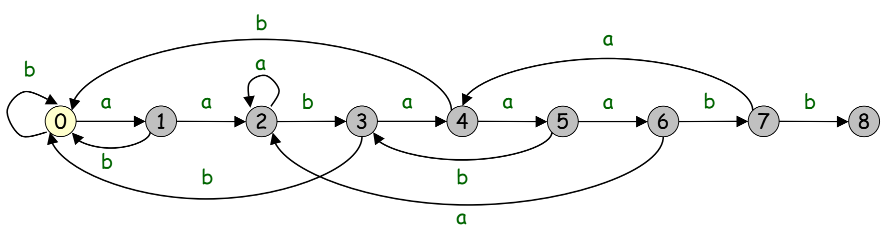


DFA Construction for KMP

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 `abaaab`**a**. X + **a** = 4
- Update X to state for `p[1..7] = abaaab`**b** X + **b** = 0



DFA Construction for KMP

DFA construction for KMP. DFA builds itself!

Crucial insight.

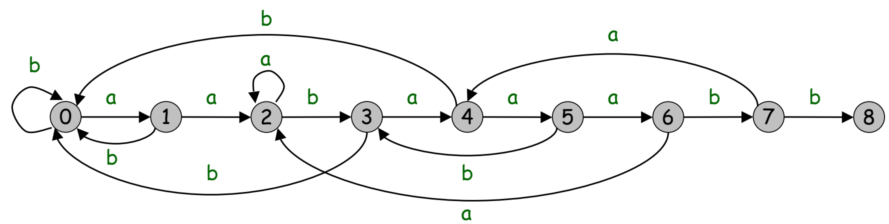
- To compute transitions for state n of DFA, suffices to have:
  - DFA for states 0 to n-1
  - state X that DFA ends up in with input `p[1..n-1]`
- To compute state X' that DFA ends up in with input `p[1..n]`, it suffices to have:
  - DFA for states 0 to n-1
  - state X that DFA ends up in with input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

DFA Construction for KMP

Search Pattern							
a	a	b	a	a	a	b	b

	0	1	2	3	4	5	6	7
a	1	2	2	4	5	6	2	4
b	0	0	3	0	0	3	7	8

j	pattern[1..j]							X	next
0								0	0
1	a							1	0
2	a	b						0	2
3	a	b	a					1	0
4	a	b	a	a				2	0
5	a	b	a	a	a			2	3
6	a	b	a	a	a	b		3	2
➡ 7	a	b	a	a	a	b	b	0	4



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;
   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;
}
```

next[]

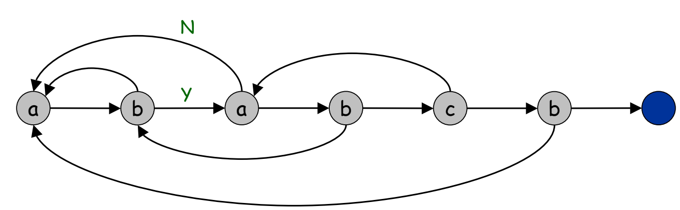
assumes pattern is in text (o/w use sentinel)

KMP Over Arbitrary Alphabet

DFA for patterns over arbitrary alphabets.

- Read new character only upon success (or failure at beginning).
- Reuse current character upon failure and follow back.
- Fact: KMP follows at most 1 + $\log_\phi$ M back links in a row.
- Theorem: at most 2N character comparisons in total.

Ex: DFA for pattern `ababcb`.



String Search Implementation Cost Summary

KMP analysis.

- DFA simulation takes $\Theta(N)$ time in worst-case.
- DFA construction takes $\Theta(M)$ time and space in worst-case.
- Extends to ASCII or UNICODE alphabets.
- Good efficiency for patterns and texts with much repetition.
- "On-line algorithm." virus scanning, internet spying

Search for an M-character pattern in an N-character text.

Implementation	Typical	Worst
Brute	1.1 N †	M N
Karp-Rabin	$\Theta(N)$	$\Theta(N)$ ‡
KMP	1.1 N †	2 N

character comparisons

† 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 an editor and avoid annoying buffer for string search

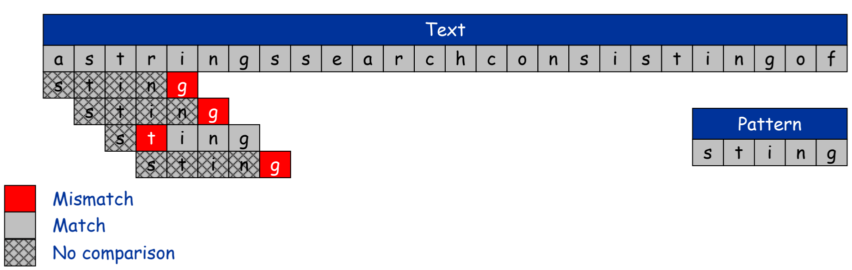
Resolved theoretical and practical problems.

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

Boyer-Moore

Boyer-Moore algorithm (1974).

- Right-to-left scanning.
  - find offset `i` in text by moving left to right.
  - compare pattern to text by moving right to left.

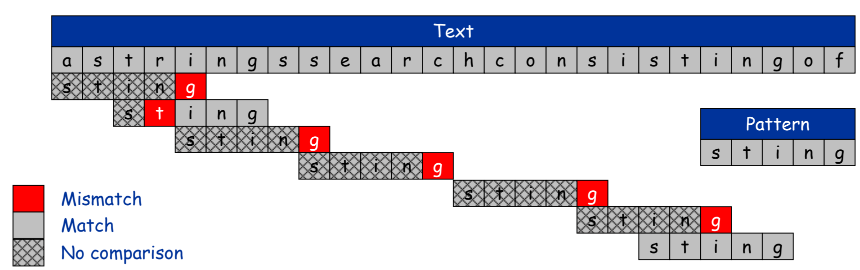


Boyer-Moore

Boyer-Moore algorithm (1974).

- Right-to-left scanning.
- Heuristic 1: advance offset i using "bad character rule."
  - upon mismatch of text character `c`, look up `index[c]`
  - increase offset `i` so that `j`th character of pattern lines up with text character `c`

Index	
g	5
i	2
n	1
s	4
t	3
*	5



Boyer-Moore

Boyer-Moore algorithm (1974).

- Right-to-left scanning.
- Heuristic 1: advance offset i using "bad character rule."
  - upon mismatch of text character `c`, look up `index[c]`
  - increase offset `i` so that `j`th character of pattern lines up with text character `c`

Index	
g	5
i	2
n	1
s	4
t	3
*	5

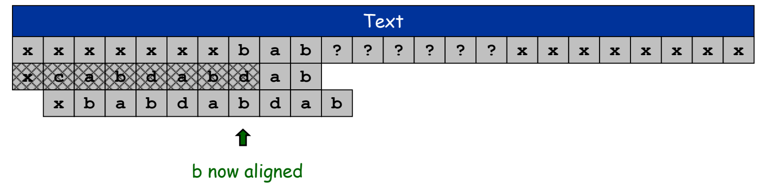
```java
private static void badCharSkip(String pattern, int[] skip) {
   int M = pattern.length();
   for (int j = 0; j < 256; j++)
      skip[j] = M;
   for (int j = 0; j < M-1; j++)
      skip[pattern.charAt(j)] = M-j-1;
}
```

construction of bad character skip table

Boyer-Moore

Boyer-Moore algorithm (1974).

- Right-to-left scanning.
- Heuristic 1: advance offset i using "bad character rule."
- Heuristic 2: use KMP-like suffix rule.
  - effective with small alphabets
  - different rules lead to different worst-case behavior

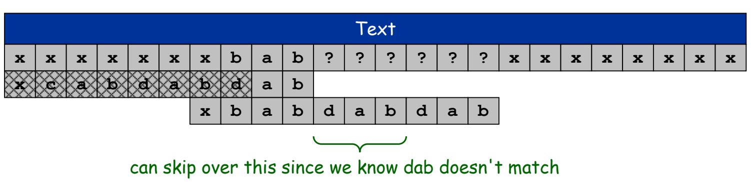


bad character rule

Boyer-Moore

Boyer-Moore algorithm (1974).

- Right-to-left scanning.
- Heuristic 1: advance offset i using "bad character rule."
- Heuristic 2: use KMP-like suffix rule.
  - effective with small alphabets
  - different rules lead to different worst-case behavior



strong good suffix

String Search Implementation Cost Summary

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!
- O(N) worst-case with Galil variant.

Search for an M-character pattern in an N-character text.

Implementation	Typical	Worst
Brute	1.1 N †	M N
Karp-Rabin	$\Theta(N)$	$\Theta(N)$ ‡
KMP	1.1 N †	2N
Boyer-Moore	N / M †	4N

character comparisons

† assumes appropriate model
‡ randomized

Boyer-Moore and Alphabet Size

Boyer-Moore space requirement. $\Theta(M + A)$

Big alphabets.
- Direct implementation may be impractical, e.g., UNICODE.
- May explain why Java's `indexOf` doesn't use it.
- Solution 1: search one byte at a time.
- Solution 2: hash UNICODE characters to smaller range.

Small alphabets.
- Loses effectiveness when A is too small, e.g., DNA.
- Solution: group characters together (aaaa, aaac, . . .).

Tip of the Iceberg

Multiple string search. Search for any of k different strings.
- Naïve: $O(M + kN)$.
- Aho-Corasick: $O(M + N)$.
- Screen out dirty words from a text stream.

Wildcards / character classes.
- Ex: PROSITE patterns for computational biology.
- $O(M + N)$ time using $O(M + A)$ extra space.
- Multiple matches

Approximate string matching: allow up to k mismatches.
- Recovering from typing or spelling errors in information retrieval.
- Fixing transmission errors in signal processing.

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 $\Theta(MN)$ 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));
}
```

s = $\underbrace{\text{aaa ... a}}_{2^n}$

pattern = $\overbrace{\text{aaa ... a}}^{2^n}\text{b}$

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

Why do you think library uses 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.
- Independent of alphabet size.
- Extends to multiple string search, wild-cards, regular expressions.

Boyer-Moore.
- Simple idea leads to dramatic speedup for long patterns.
- Need to tweak for small or large alphabets.