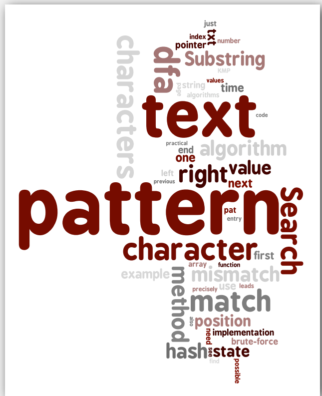
5.3 Substring Search



- brute force
- Knuth-Morris-Pratt
- Boyer-Moore
- Rabin-Karp

Substring search

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
```

Computer forensics. Search memory or disk for signatures, e.g., all URLs or RSA keys that the user has entered.

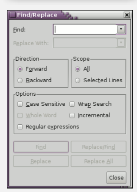
http://citp.princeton.edu/memory

Applications

- Parsers.
- Spam filters.
- Digital libraries.
- Screen scrapers.
- Word processors.
- Web search engines.
- Electronic surveillance.
- Natural language processing.
- Computational molecular biology.
- FBIs Digital Collection System 3000.
- Feature detection in digitized images.
- ...

SpamAssassin

LexisNexis™ It's how you know™

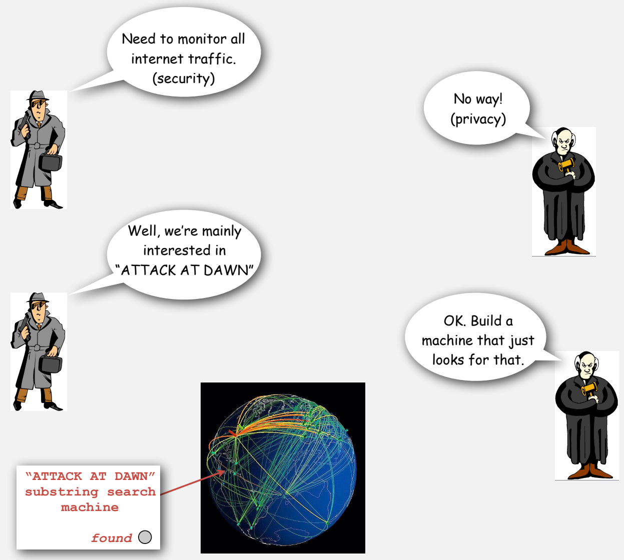


Application: Spam filtering

Identify patterns indicative of spam.

- `PROFITS`
- `L0SE WE1GHT`
- `herbal Viagra`
- `There is no catch.`
- `LOW MORTGAGE RATES`
- `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.`

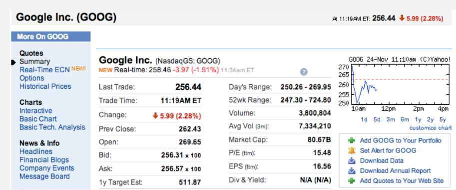
Application: Electronic surveillance



Application: Screen scraping

Goal. Extract relevant data from web page.

Ex. Find string delimited by `<b>` and `</b>` after first occurrence of pattern `Last Trade:`.



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

```
...
<tr>
<td class= "yfnc_tablehead1"
width= "48%">
Last Trade:
</td>
<td class= "yfnc_tabledata1">
<big><b>452.92</b></big>
</td></tr>
<td class= "yfnc_tablehead1"
width= "48%">
Trade Time:
</td>
<td class= "yfnc_tabledata1">
...
```


Screen scraping: Java 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.

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

```
% java StockQuote goog
256.44

% java StockQuote msft
19.68
```


- **brute force**
- Knuth-Morris-Pratt
- Boyer-Moore
- Rabin-Karp

Brute-force substring search

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	

← pat

entries in red are mismatches

entries in gray are for reference only

entries in black match the text

return i *when* j *is* M

match

Brute-force substring search

Brute-force substring search: Java implementation

Check for pattern starting at each text position.

```
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;   ← index in text where pattern starts
   }
   return N;   ← not found
}
```


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					
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	4	9						A	A	A	A	B

← pat

Brute-force substring search (worst case)

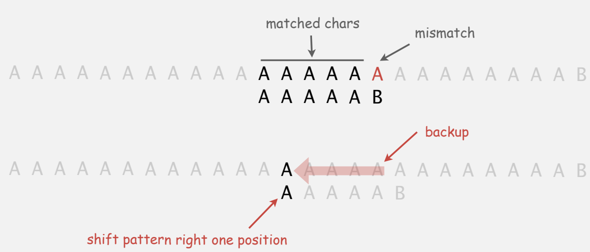
Worst case. ~ M N char compares.

Backup

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

- Treat input as stream of data.
- Abstract model: `StdIn`.

Brute-force algorithm needs backup for every mismatch



Approach 1. Maintain buffer of size `M` (build backup into `StdIn`)

Approach 2. Stay tuned.

Brute-force substring search: alternate implementation

Same sequence of char compares as previous implementation.

- `i` points to end of sequence of already-matched chars in text.
- `j` stores number of already-matched chars (end of sequence in pattern).

```
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;
}
```

backup

Algorithmic challenges in substring search

Brute-force is often not 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.

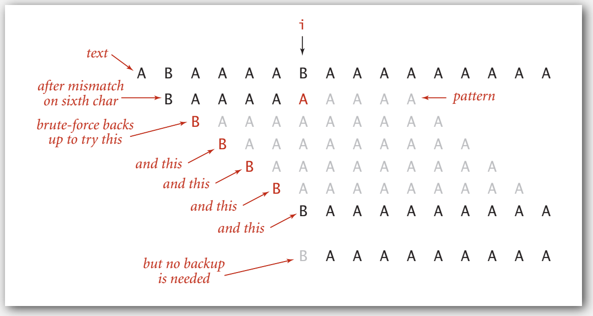
- brute force
- **Knuth-Morris-Pratt**
- Boyer-Moore
- Rabin-Karp

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 6^{th} char.
- We know previous 6 chars in text are **BAAAAB**.
- Don't need to back up text pointer!

assuming {A, B} alphabet

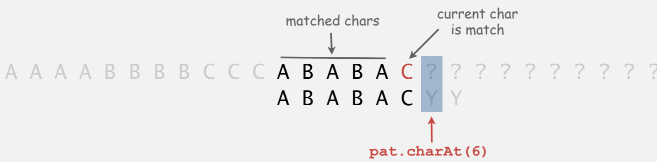


Remark. It is always possible to avoid backup (!)

KMP substring search preprocessing (concept)

Q. What pattern char do we compare to the next text char on match?

A. Easy: compare next pattern char to next text char.



	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

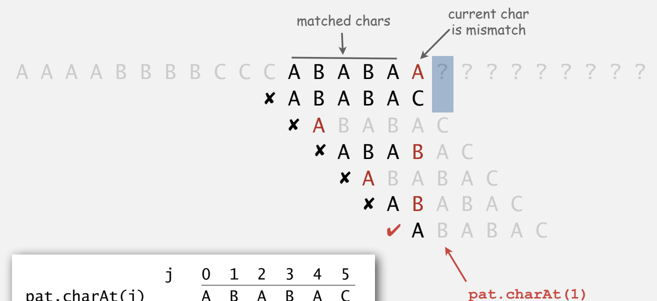
table giving pattern char to compare to the next text char

current text char: `c`
current pattern index: `j`
next pattern index: `dfa[c][j]`

KMP substring search preprocessing (concept)

Q. What pattern char do we compare to the next text char on mismatch?

A. Check each position, working from left to right.



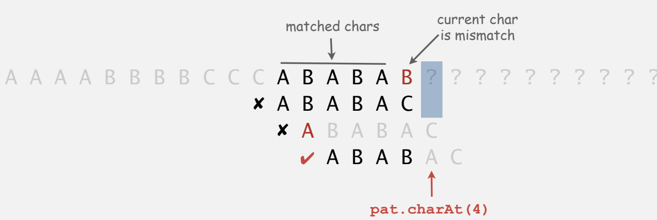
	j	0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6

table giving pattern char to compare to the next text char

KMP substring search preprocessing (concept)

Q. What pattern char do we compare to the next text char on mismatch?

A. Check each position, working from left to right.

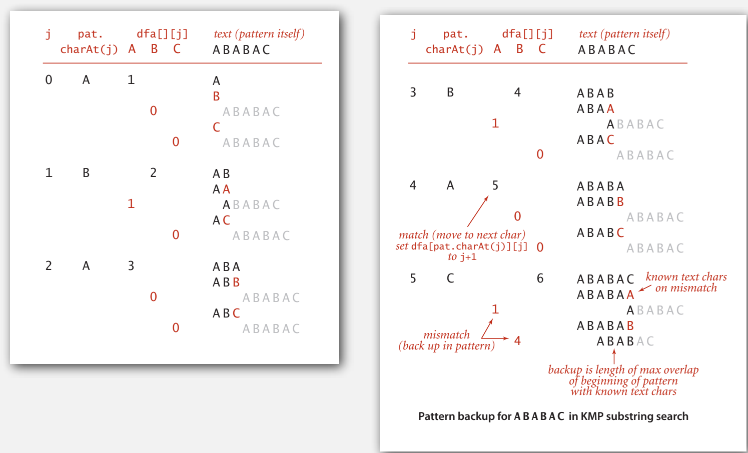


	j	0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6

table giving pattern char to compare to the next text char

KMP substring search preprocessing (concept)

Fill in table columns by doing computation for each possible mismatch position.



Pattern backup for A B A B A C in KMP substring search

Deterministic finite state automaton (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.

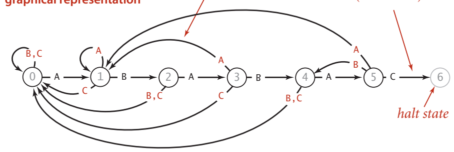
internal representation

	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

mismatch transition (back up)

match transition (increment)

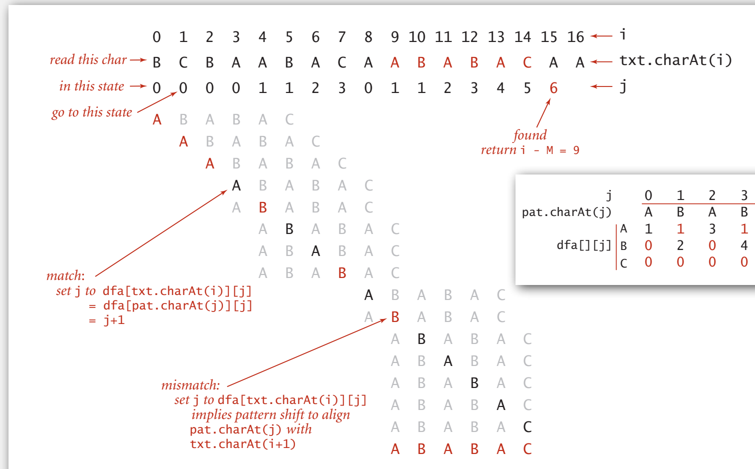
graphical representation



DFA corresponding to the string A B A B A C

If in state `j` reading char `c`:
    halt if `j` is `6`
    else move to state `dfa[c][j]`

KMP substring search: trace



	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

KMP search: Java implementation

KMP implementation. Build machine for pattern, simulate it on text.

Key differences from brute-force implementation.

- Text pointer `i` never decrements.
- Need to precompute `dfa[][]` table from pattern.

```
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;
}
```

Running time.

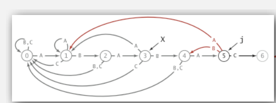
- Simulate DFA: at most N character accesses.
- Build DFA: at most $M^2 R$ character accesses (stay tuned for better method).

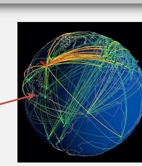
KMP search: Java implementation

Key differences from brute-force implementation.

- Text pointer `i` never decrements.
- Need to precompute `dfa[][]` table from pattern.
- Could use input stream.

```
public int search(In in)
{
   int i, j;
   for (i = 0, j = 0; !in.isEmpty() && j < M; i++)
      j = dfa[in.readChar()][j];
   if (j == M) return i - M;
   else        return i;
}
```





Efficiently constructing the DFA for KMP substring search

Q. What state X would the DFA be in if it were restarted to correspond to shifting the pattern one position to the right?

matched chars   next char

A B B B B C C C A B A B A ? ? ? ? ? ? ?
A B A B A C

A. Use the (partially constructed) DFA to find X!

A B B B B C C C A B A B A ? ? ? ? ? ? ?
0 0 1 2 3

X

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

Consequence.

- We want the same transitions as X for the next state on mismatch.
  copy `dfa[][X]` to `dfa[][j]`
- But a different transition (to j+1) on match.
  set `dfa[pat.charAt(j)][j]` to `j+1`

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

Efficiently constructing the DFA for KMP substring search

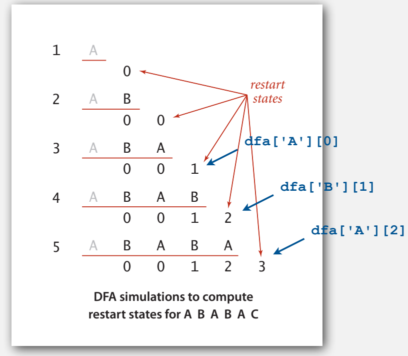
Build table by finding answer to Q for each pattern position.

Q. What state X would the DFA be in if it were restarted to correspond to shifting the pattern one position to the right?

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

Observation. No need to restart DFA.

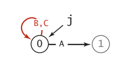
- Remember last restart state in X.
- Use DFA to update X.
- `X = dfa[pat.charAt(j)][X]`



DFA simulations to compute restart states for A B A B A C

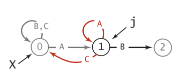
Constructing the DFA for KMP substring search: example

j		0
pat.charAt(j)		A
dfa[][j]	A	1
	B	0
	C	0



X

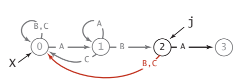
j		0	1
pat.charAt(j)		A	B
dfa[][j]	A	1	1
	B	0	2
	C	0	0



copy `dfa[][X]` *to* `dfa[][j]`
`dfa[pat.charAt(j)][j] = j+1;`
`X = dfa[pat.charAt(j)][X];`

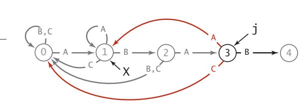
X

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



X

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

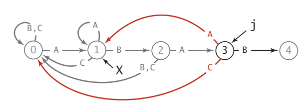


Constructing the DFA for KMP substring search for A B A B A C

Constructing the DFA for KMP substring search: example

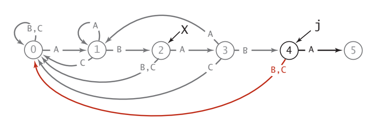
X

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



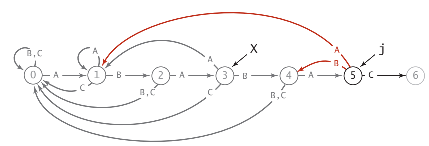
X

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



X

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



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

Constructing the DFA for KMP substring search: Java implementation

For each `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];          // copy mismatch cases
      dfa[pat.charAt(j)][j] = j+1;       // set match case
      X = dfa[pat.charAt(j)][X];         // update restart state
   }
}
```

Running time. M character accesses.

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.

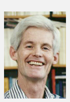
Pf. We access each pattern char once when constructing the DFA, and each text char once (in the worst case) when simulating the DFA.

Remark. Takes time and space proportional to R M to construct `dfa[][]`, but with cleverness, can reduce time and space to M.

Knuth-Morris-Pratt: brief history

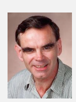
Brief history.

- Inspired by esoteric theorem of Cook.
- 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
- Theory meets practice.

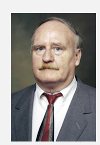

Stephen Cook

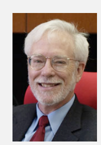

Don Knuth


Jim Morris


Vaughan Pratt

- brute force
- Knuth-Morris-Pratt
- **Boyer-Moore**
- Rabin-Karp


Robert Boyer


J. Strother Moore

Boyer-Moore: mismatched character heuristic

Intuition.

- Scan characters in pattern from right to left.
- Can skip M text chars when finding one not in the pattern.

i	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
text →		H	A	Y	S	T	A	C	K	N	E	E	D	L	E	I	N	A
0	5	N	E	E	D	L	E ← *pattern*											
6	5							N	E	E	D	L	E					
8	5									N	E	E	D	L	E			
8	0																	

return i = 8

Boyer-Moore: mismatched character heuristic

Intuition.

- Scan characters in pattern from right to left.
- Can skip M text chars when finding one not in the pattern.

not in pattern

txt[]

i	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
		A	A	A	B	B	A	A	B	A	B	A	A	A	B	B	A	G	Y
5	6	S	Y	Z	Y	G	Y												
11	6							S	Y	Z	Y	G	Y						
15	3													S	Y	Z	Y	G	Y

return i = 18 *(no match)*

Boyer-Moore: mismatched character heuristic

Q. How much to skip?

A. Compute `right[c]` = rightmost occurrence of character `c` in `pat[]`.

```
right = new int[R];
for (int c = 0; c < R; c++)
   right[c] = -1;
for (int j = 0; j < M; j++)
   right[pat.charAt(j)] = j;
```

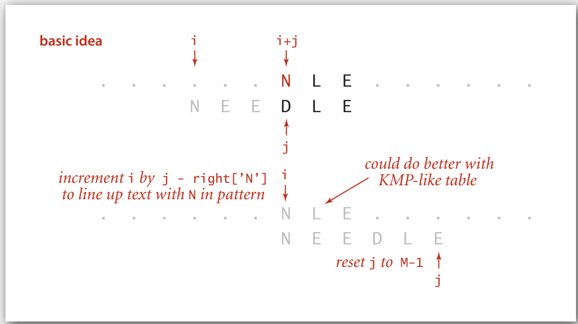
		N	E	E	D	L	E	
c		0	1	2	3	4	5	right[c]
A	-1	-1	-1	-1	-1	-1	-1	-1
B	-1	-1	-1	-1	-1	-1	-1	-1
C	-1	-1	-1	-1	-1	-1	-1	-1
D	-1	-1	-1	-1	3	3	3	3
E	-1	-1	1	2	2	2	5	5
...								-1
L	-1	-1	-1	-1	-1	4	4	4
M	-1	-1	-1	-1	-1	-1	-1	-1
N	-1	0	0	0	0	0	0	0
...								-1

Boyer-Moore skip table computation

Boyer-Moore: mismatched character heuristic

Q. How much to skip?

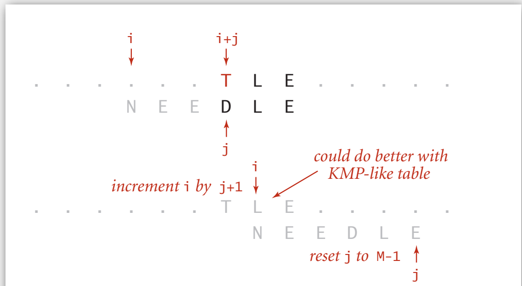
A. Compute `right[c]` = rightmost occurrence of character `c` in `pat[]`.



Boyer-Moore: mismatched character heuristic

Q. How much to skip?

A. Compute `right[c]` = rightmost occurrence of character `c` in `pat[]`.



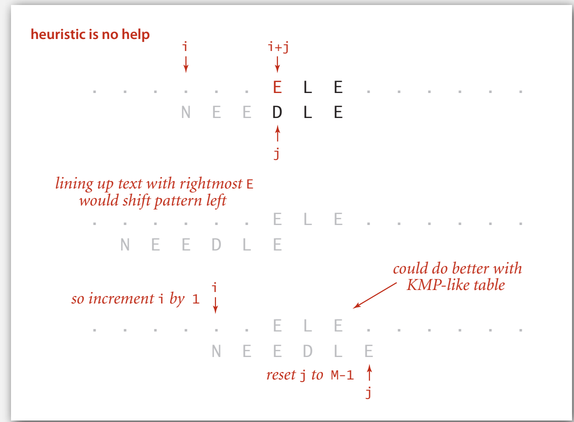
Mismatched character heuristic (mismatch not in pattern)

Easy fix. Set `right[c]` to -1 for characters not in pattern.

Boyer-Moore: mismatched character heuristic

Q. How much to skip?

A. Compute `right[c]` = rightmost occurrence of character `c` in `pat[]`.



Boyer-Moore: Java implementation

```
public int search(String txt)
{
   int N = txt.length();
   int M = pat.length();
   int skip;
   for (int i = 0; i <= N-M; i += skip)
   {
      skip = 0;
      for (int j = M-1; j >= 0; j--)
         if (pat.charAt(j) != txt.charAt(i+j))
         {
            skip = Math.max(1, j - right[txt.charAt(i+j)]);
            break;
         }
      if (skip == 0) return i;
   }
   return N;
}
```

compute skip value

match

Boyer-Moore: analysis

Property. Substring search with the Boyer-Moore mismatched character heuristic takes about ~ N/M character compares to search for a pattern of length M in a text of length N. sublinear

Worst-case. Can be as bad as ~ M N.

i	skip		0	1	2	3	4	5	6	7	8	9	
		txt →	B	B	B	B	B	B	B	B	B	B	
0	0		A	B	B	B	B						← pat
1	1			A	B	B	B	B					
2	1				A	B	B	B	B				
3	1					A	B	B	B	B			
4	1						A	B	B	B	B		
5	1							A	B	B	B	B	

Boyer-Moore variant. Can improve worst case to ~ 3 N by adding a KMP-like rule to guard against repetitive patterns.

- brute force
- Knuth-Morris-Pratt
- Boyer-Moore
- **Rabin-Karp**

Michael Rabin, Turing Award '76
and Dick Karp, Turing Award '85

Rabin-Karp fingerprint search

Basic idea.

- Compute a hash of pattern characters 0 to M-1.
- For each i, compute a hash of text characters i to M+i-1.
- If pattern hash = text substring hash, check for a match.

```
      pat.charAt(i)
 i    0  1  2  3  4
      2  6  5  3  5  % 997 = 613

                      txt.charAt(i)
 i    0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
      3  1  4  1  5  9  2  6  5  3  5  8  9  7  9  3
 0    3  1  4  1  5  % 997 = 508
 1       1  4  1  5  9  % 997 = 201
 2          4  1  5  9  2  % 997 = 715
 3             1  5  9  2  6  % 997 = 971
 4                5  9  2  6  5  % 997 = 442
 5                   9  2  6  5  3  % 997 = 929   match
 6  ← return i = 6      2  6  5  3  5  % 997 = 613
```

Basis for Rabin-Karp substring search

Efficiently computing the hash function

Modular hash function. Using the notation t_i for `txt.charAt(i)`, we wish to compute

$$x_i = t_i\, R^{M-1} + t_{i+1}\, R^{M-2} + \ldots + t_{i+M-1}\, R^{0} \pmod{Q}$$

Intuition. M-digit, base-R integer, modulo Q.

Horner's method. Linear-time method to evaluate degree-M polynomial.

```
      pat.charAt(i)
 i    0  1  2  3  4
      2  6  5  3  5
 0    2  % 997 = 2           R            Q
 1    2  6  % 997 = (2*10 + 6) % 997 = 26
 2    2  6  5  % 997 = (26*10 + 5) % 997 = 265
 3    2  6  5  3  % 997 = (265*10 + 3) % 997 = 659
 4    2  6  5  3  5  % 997 = (651*10 + 5) % 997 = 613
```

Computing the hash value for the pattern with Horner's method

```
// Compute hash for M-digit key
private int hash(String key)
{
   int h = 0;
   for (int i = 0; i < M; i++)
      h = (R * h + key.charAt(j)) % Q;
   return h;
}
```


Efficiently computing the hash function

Challenge. How to efficiently compute x_{i+1} given that we know x_i.

$$x_i = t_i\, R^{M-1} + t_{i+1}\, R^{M-2} + \ldots + t_{i+M-1}\, R^{0}$$

$$x_{i+1} = t_{i+1}\, R^{M-1} + t_{i+2}\, R^{M-2} + \ldots + t_{i+M}\, R^{0}$$

Key property. Can do it in constant time!

$$x_{i+1} = (x_i - t_i\, R^{M-1})\, R + t_{i+M}$$

```
 i              ...  2  3  4  5  6  7  ...
 current value   1   4  1  5  9  2  6  5   text
     new value       4  1  5  9  2  6  5

                     4  1  5  9  2     current value
                  -  4  0  0  0  0
                        1  5  9  2     subtract leading digit
                           *  1  0     multiply by radix
                     1  5  9  2  0
                              +  6     add new trailing digit
                     1  5  9  2  6     new value
```

Rabin-Karp: Java implementation

```
public class RabinKarp {
    private String pat;      // the pattern
    private int patHash;     // pattern hash value
    private int M;           // pattern length
    private int Q = 8355967; // modulus
    private int R;           // radix
    private int RM;          // R^(M-1) % Q

    public RabinKarp(String pat) {
        this.R = 256;
        this.pat = pat;
        this.M = pat.length;

        RM = 1;
        for (int i = 1; i <= M-1; i++)
           RM = (R * RM) % Q;
        patHash = hash(pat);
     }

    private int hash(String key)
    {  /* as before */  }

    public int search(String txt)
    {  /* see next slide */  }
}
```

a large prime, but small enough to avoid 32-bit integer overflow

precompute R^{M-1} (mod Q)

Rabin-Karp: Java implementation (continued)

```
public int search(String txt)
{
   int N = txt.length();
   if (N < M) return N;
   int offset = hashSearch(txt);
   if (offset == N) return N;

   for (int i = 0; i < M; i++)
       if (pat.charAt(i) != txt.charAt(offset + i))
          return N;
    return offset;
}

private int hashSearch(String txt)
{
    int N = txt.length();
    int txtHash = hash(txt);
    if (patHash == txtHash) return 0;
    for (int i = M; i < N; i++)
    {
        txtHash = (txtHash + Q - RM*txt.charAt(i-M) % Q) % Q;
        txtHash = (txtHash*R + txt.charAt(i)) % Q;
        if (patHash == txtHash) return i - M + 1;
    }
    return N;
}
```

check if hash collision corresponds to a match

check for hash collision using rolling hash function

Rabin-Karp substring search example

```
 i    0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
      3  1  4  1  5  9  2  6  5  3  5  8  9  7  9  3
 0    3  % 997 = 3                      Q
 1    3  1  % 997 = (3*10 + 1) % 997 = 31
 2    3  1  4  % 997 = (31*10 + 4) % 997 = 314
 3    3  1  4  1  % 997 = (314*10 + 1) % 997 = 150
 4    3  1  4  1  5  % 997 = (150*10 + 5) % 997 = 508   RM  R
 5       1  4  1  5  9  % 997 = ((508 + 3*(997 - 30))*10 + 9) % 997 = 201
 6          4  1  5  9  2  % 997 = ((201 + 1*(997 - 30))*10 + 2) % 997 = 715
 7             1  5  9  2  6  % 997 = ((715 + 4*(997 - 30))*10 + 6) % 997 = 971
 8                5  9  2  6  5  % 997 = ((971 + 1*(997 - 30))*10 + 5) % 997 = 442   match
 9                   9  2  6  5  3  % 997 = ((442 + 5*(997 - 30))*10 + 3) % 997 = 929
10 ← return i-M+1 = 6   2  6  5  3  5  % 997 = ((929 + 9*(997 - 30))*10 + 5) % 997 = 613
```

Rabin-Karp substring search example

Rabin-Karp analysis

Proposition. Rabin-Karp substring search is extremely likely to be linear-time.

Worst-case. Takes time proportional to MN.

- In worst case, all substrings hash to same value.
- Then, need to check for match at each text position.

Theory. If Q is a sufficiently large random prime (about MN^2), then probability of a false collision is about 1/N ⇒ expected running time is linear.

Practice. Choose Q to avoid integer overflow. Under reasonable assumptions, probability of a collision is about 1/Q ⇒ linear in practice.

Rabin-Karp fingerprint search

Advantages.

- Extends to 2D patterns.
- Extends to finding multiple patterns.

Disadvantages.

- Arithmetic ops slower than char compares.
- Poor worst-case guarantee.
- Requires backup.

Q. How would you extend Rabin-Karp to efficiently search for any one of P possible patterns in a text of length N?

Substring search cost summary

Cost of searching for an M-character pattern in an N-character text.

algorithm (data structure)	operation count		backup in input?	space grows with
	guarantee	typical		
brute force	MN	$1.1\,N$	*yes*	1
Knuth-Morris-Pratt (full DFA)	$2N$	$1.1\,N$	*no*	MR
Knuth-Morris-Pratt (mismatch transitions only)	$3N$	$1.1\,N$	*no*	M
Boyer-Moore	$3N$	N/M	*yes*	R
Boyer-Moore (mismatched character heuristic only)	MN	N/M	*yes*	R
Rabin-Karp†	$7N$ †	$7N$	*no*	1

† probabilisitic guarantee, with uniform hash function

Cost summary for substring-search implementations