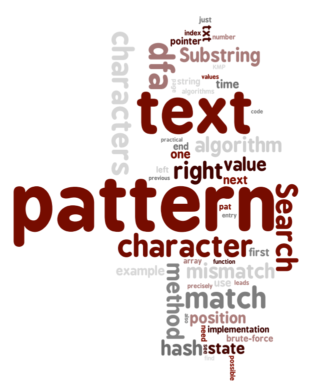
6.3 Substring Search



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

Algorithms in Java, 4th Edition · Robert Sedgewick and Kevin Wayne · Copyright © 2008 · April 7, 2009 7:35:27 PM

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

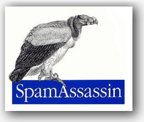
match

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

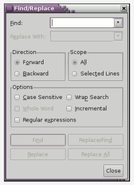
http://citp.princeton.edu/memory

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.
- ...



LexisNexis™
It's how you know™



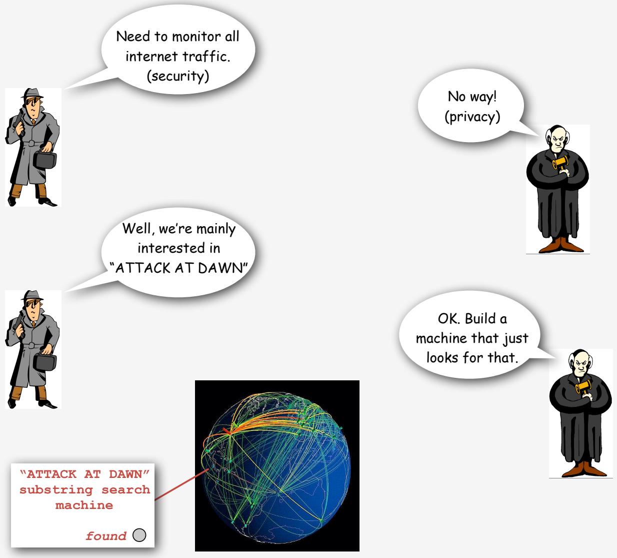
DCS3000

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.`

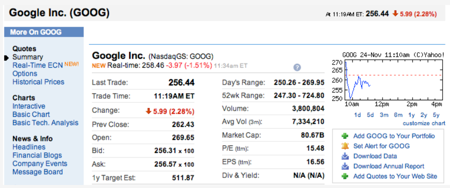
Application: Electronic surveillance



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:`.



Google Inc. (GOOG) At 11:19AM ET: 256.44 ↓ 5.99 (2.28%)

Google Inc. (NasdaqGS: GOOG)
NEW Real-time: 258.46 -3.97 (-1.51%) 11:34am ET

Last Trade:	256.44	Day's Range:	250.26 - 269.95
Trade Time:	11:19AM ET	52wk Range:	247.30 - 724.80
Change:	↓ 5.99 (2.28%)	Volume:	3,800,804
Prev Close:	262.43	Avg Vol (3m):	7,334,210
Open:	269.65	Market Cap:	80.67B
Bid:	256.31 x 100	P/E (ttm):	15.48
Ask:	256.57 x 100	EPS (ttm):	16.56
1y Target Est:	511.87	Div & Yield:	N/A (N/A)

`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, strating at a given offset.

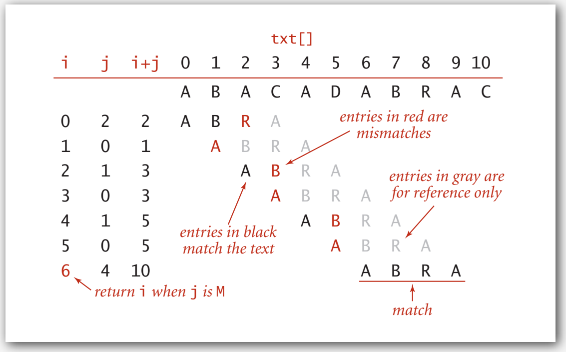
```
public class StockQuote
{
   public static void main(String[] args)
   {
      String name = "http://finance.yahoo.com/q?s=";
      In in = new In(name + args[0]);
      String text = in.readAll();
      int start    = text.indexOf("Last Trade:", 0);
      int from     = text.indexOf("<b>",   start);
      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
			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	

Brute-force substring search: Java implementation

Check for pattern starting at each text position.

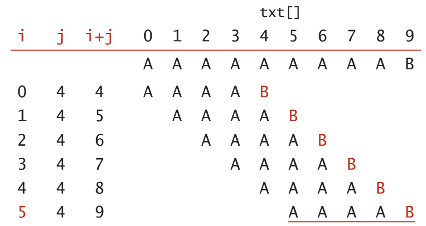
```
public static int search(char[] pat, char[] txt)
{
   int M = pat.length;
   int N = txt.length;
   for (int i = 0; i < N - M; i++)
   {
      int j;
      for (j = 0; j < M; j++)
         if (txt[i+j] != pat[j])
            break;
      if (j == M) return i;
   }
   return N;
}
```

index in text where pattern starts

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
			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

Brute-force substring search (worst case)

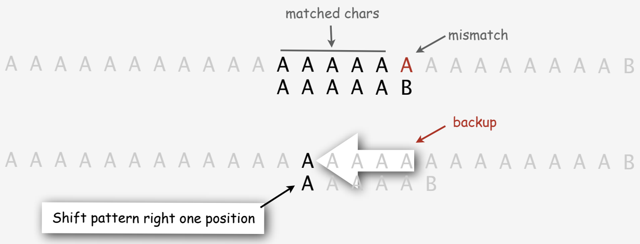
Worst case. ~ MN char compares.

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`)

Other approaches: 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(char[] pat, char[] txt)
{
   int j, M = pat.length;
   int i, N = txt.length;
   for (i = 0, j = 0; i < N && j < M; i++)
   {
      if (txt[i] == pat[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

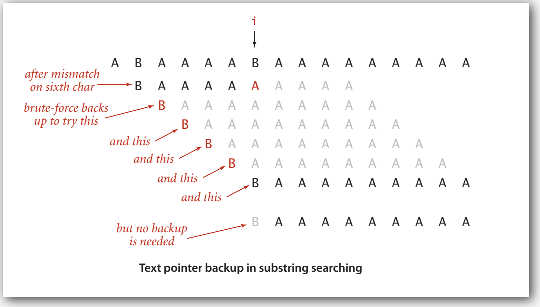
Now is the time for all people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for many good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for a 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 6th char.
- We know previous 6 chars in text are `BAAAAB`.
- Don't need to back up text pointer!



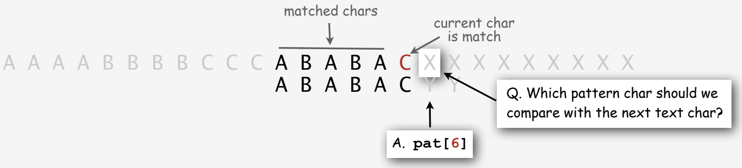
Text pointer backup in substring searching

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: the next one.



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

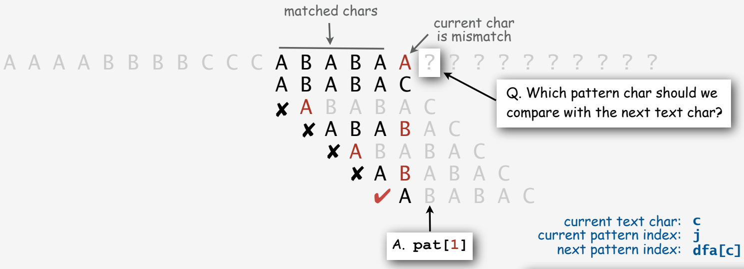
table giving pattern char to compare to the next text char

	j	0	1	2	3	4	5
	pat[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

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.



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

table giving pattern char to compare to the next text char

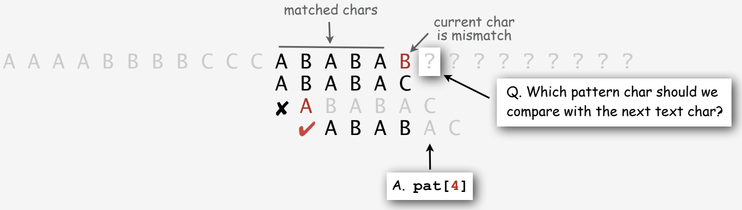
	j	0	1	2	3	4	5
	pat[j]	A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6

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.

A. Have to do it for each possible char that could mismatch



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

table giving pattern char to compare to the next text char

	j	0	1	2	3	4	5
	pat[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

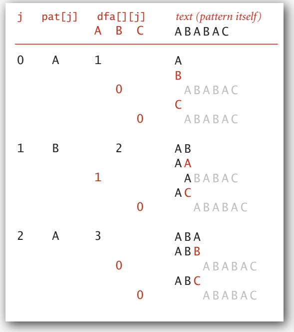
KMP substring search preprocessing (concept)

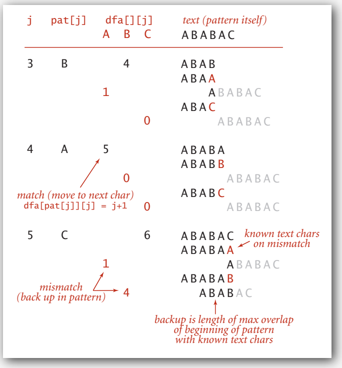
Fill in table columns by doing computation for each possible mismatch position

Ex. Builf table for `ABABAC`.

	j	0	1	2	3	4	5
	pat[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

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



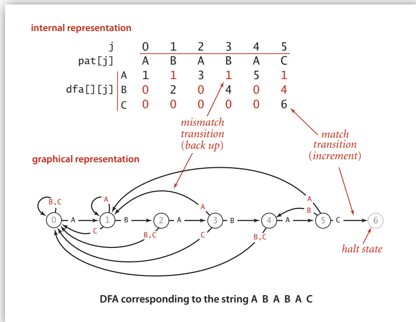


Total cost: $O(M^2R)$ char compares (stay tuned for a better method).

Deterministic finite state automaton (DFA)

DFA is abstract string-searching machine.

- Finite number of states (including start and halt).
- Exactly one transition for each input symbol.
- Accept if sequence of transitions leads to halt state.



j		0	1	2	3	4	5
pat[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

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]`

Knuth-Morris Pratt algorithm: Build machine for pattern, simulate it on text.

KMP search: Java implementation

Key differences from brute-force implementation.

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

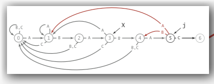
```
public int search(int[] txt)
{
   int i, j, N = txt.length;
   for (i = 0, j = 0; i < N && j < M; i++)
      j = dfa[txt[i]][j];
   if (j == M) return i - M;
   else        return N;
}
```

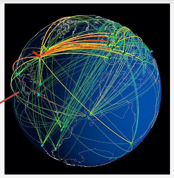
KMP search: Java implementation

Key differences from brute-force implementation.

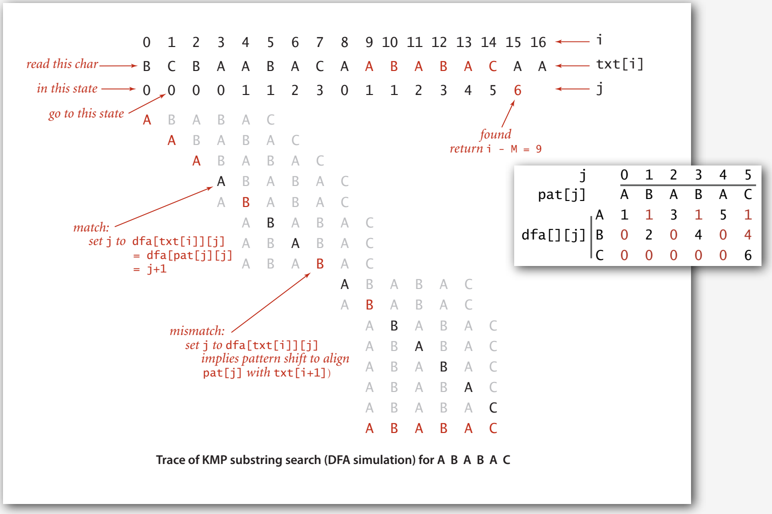
- Text pointer `i` never decrements.
- Need to precompute `dfa[][]` table from pattern.
- Could use input stream.

```
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;
}
```





KMP substring search: trace



j		0	1	2	3	4	5
pat[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

Efficiently constructing the DFA for KMP substring search

Q1. 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 A A A B B B B C C C A B A B A ? ? ? ? ? ? ? ? ? ?
A B A B A C

j		0	1	2	3	4
pat[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

A1. Use the (partially constructed) DFA to find X!

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

Q2. Why is that relevant?

A2. We want the same transitions for the next state on mismatch

`copy dfa[][X] to dfa[][j]`

j		0	1	2	3	4	5
pat[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	0

Efficiently constructing the DFA for KMP substring search

Q1. 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 A A A B B B B C C C A B A B A ? ? ? ? ? ? ? ? ? ?
A B A B A C

j		0	1	2	3	4
pat[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

A1. Use the (partially constructed) DFA to find X!

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

Q2. Why is that relevant?

A2. We want the same transitions for the next state on mismatch

`copy dfa[][X] to dfa[][j]`

A2 (continued). and a different transition (to j+1) on match

`dfa[pat[j]][j] = j+1`

j		0	1	2	3	4	5
pat[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 Q1 for each pattern position.

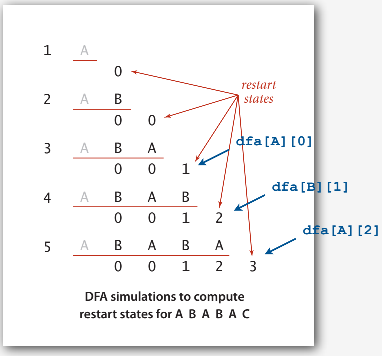
Q1. What state X would the DFA be in if it were restarted to correspond to shifting the pattern one position to the right?

j		0	1	2	3	4	5
pat[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

Important note:

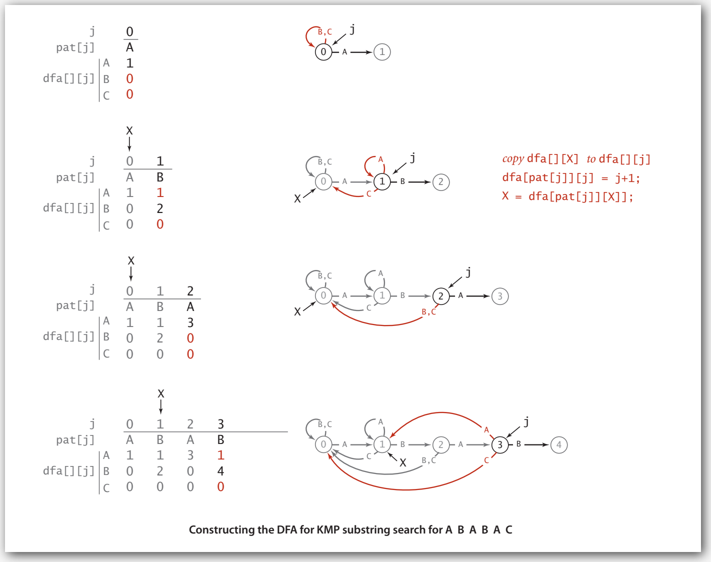
- no need to restart DFA
- remember last restart state in X
- use DFA to update X

`X = dfa[pat[j]][X]`



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

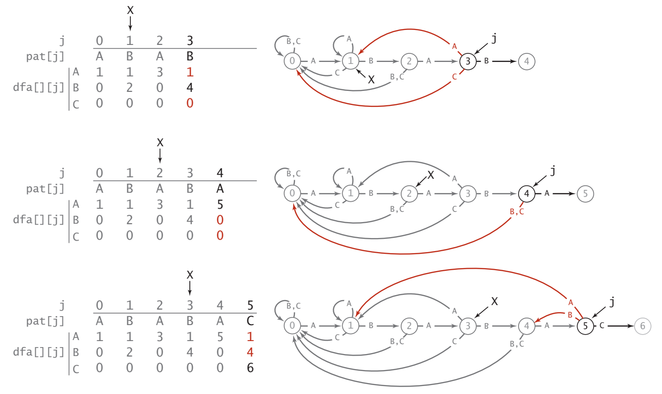
Constructing the DFA for KMP substring search: example



copy dfa[][X] to dfa[][j]
dfa[pat[j]][j] = j+1;
X = dfa[pat[j]][X];

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

Constructing the DFA for KMP substring search: example



j	0	1	2	3
pat[j]	A	B	A	B
dfa[][j] A	1	1	3	1
dfa[][j] B	0	2	0	4
dfa[][j] C	0	0	0	0

j	0	1	2	3	4
pat[j]	A	B	A	B	A
dfa[][j] A	1	1	3	1	5
dfa[][j] B	0	2	0	4	0
dfa[][j] C	0	0	0	0	0

j	0	1	2	3	4	5
pat[j]	A	B	A	B	A	C
dfa[][j] A	1	1	3	1	5	1
dfa[][j] B	0	2	0	4	0	4
dfa[][j] 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[j]][j]` to `j+1` for match case.
- Update X.

```
public KMP(int R, char[] pat)
{
   this.pat = pat;
   M = pat.length;
   dfa = new int[R][M];
   dfa[pat[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];
      dfa[pat[j]][j] = j+1;
      X = dfa[pat[j]][X];
   }
}
```

copy mismatch cases

set match case

update restart state

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.

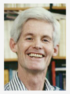
Pf. We access each pattern char once when construction DFA, and we access each text char once (in the worst case) when simulating the DFA on given text.

Remark. Takes time and space proportional to R M to build `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

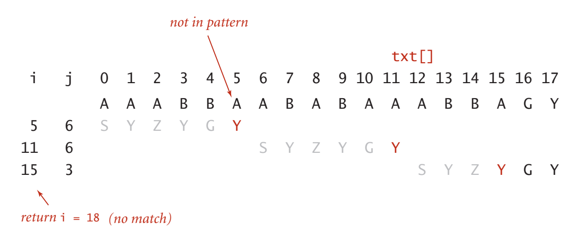
Robert Boyer

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.



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
```

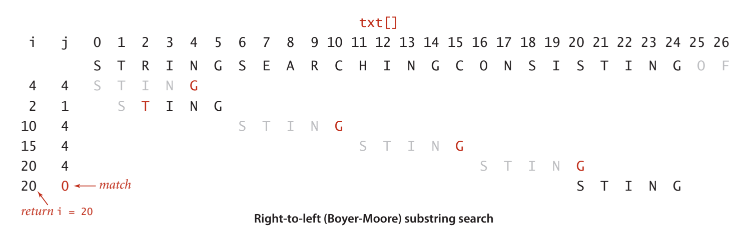
return i = 18 (no match)

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[j]] = j;
```



txt[]

```
  i   j   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
          S  T  R  I  N  G  S  E  A  R  C  H  I  N  G  C  O  N  S  I  S  T  I  N  G  O  F
  4   4   S  T  I  N  G
  2   1      S  T  I  N  G
 10   4                     S  T  I  N  G
 15   4                                    S  T  I  N  G
 20   4                                                   S  T  I  N  G
 20   0                                                               S  T  I  N  G
```

← match

return i = 20

Right-to-left (Boyer-Moore) substring search

Boyer-Moore: Java implementation

```
public int search(char[] 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[j] != txt[i+j])
         {
            skip = Math.max(1, j - right[txt[i+j]]);
            break;
         }
      if (skip == 0) return i;
   }
   return N;
}
```

← compute skip value

Boyer-Moore: analysis

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

sublinear

Worst-case. Can be as bad as MN.

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

- Used in Unix, emacs.

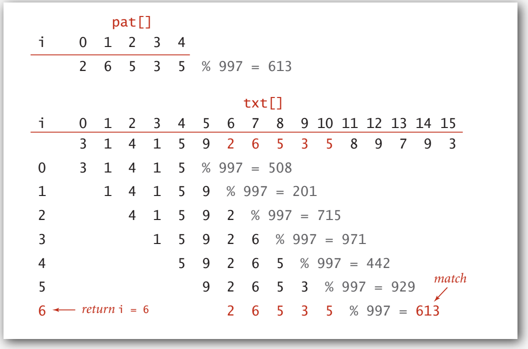
- brute force
- Knuth-Morris-Pratt
- Boyer-Moore
- Rabin-Karp

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

Rabin-Karp fingerprint search

Basic idea.

- Compute a hash of `pat[0..M)`.
- Compute a hash of `txt[i..M+i)` for each `i`.
- If pattern hash = text substring hash, check for a match.



```
          pat[]
i     0  1  2  3  4
      2  6  5  3  5  % 997 = 613

                        txt[]
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
```

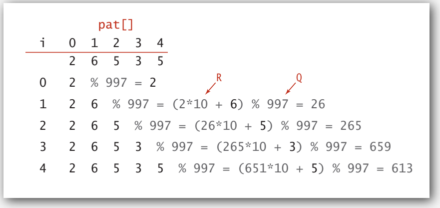
Efficiently computing the hash function

Modular hash function. Using the notation t_i for `txt[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[]
i   0  1  2  3  4
    2  6  5  3  5
                               R          Q
0   2  % 997 = 2
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
```

```
// Compute hash for key[0..M-1]
private int hash(char[] key, int M)
{
   int h = 0;
   for (int j = 0; j < M; j++)
      h = (R * h + key[j]) % Q;
   return h;
}
```

Efficiently computing the hash function

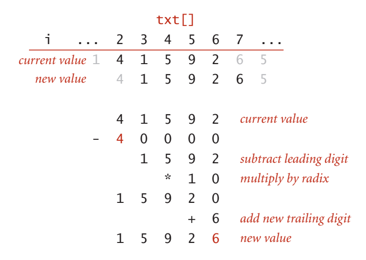
Challenge. How to efficiently compute x_{i+1} given that we know x_i.

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

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

Key observation. Can do it in constant time!

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



```
                    txt[]
   i      ...  2  3  4  5  6  7  ...
current value 1  4  1  5  9  2  6  5
    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 char[] 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(int R, char[] pat) {
        this.R = R;
        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(char[] key)
    {  /* as before */  }

    public int search(char[] txt)
    {  /* see next slide */  }
}
```

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

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

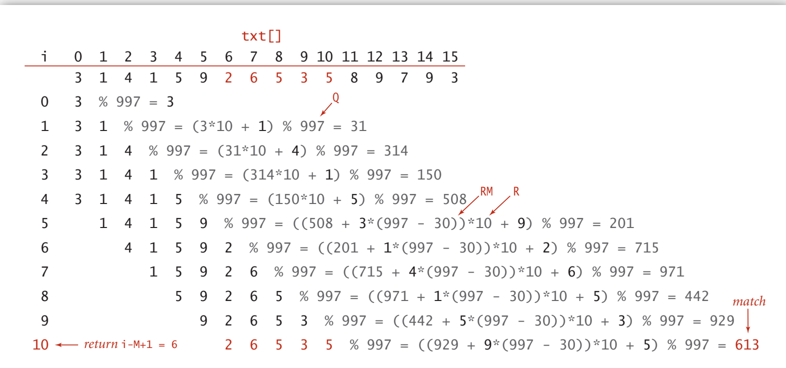
Rabin-Karp: Java implementation (continued)

```
public int search(char[] 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[i] != txt[offset + i])
           return N;
    return offset;
}

private int hashSearch(char[] 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[i-M] % Q) % Q;
        txtHash = (txtHash*R + txt[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



```
                                txt[]
 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 analysis

Property 4. 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.

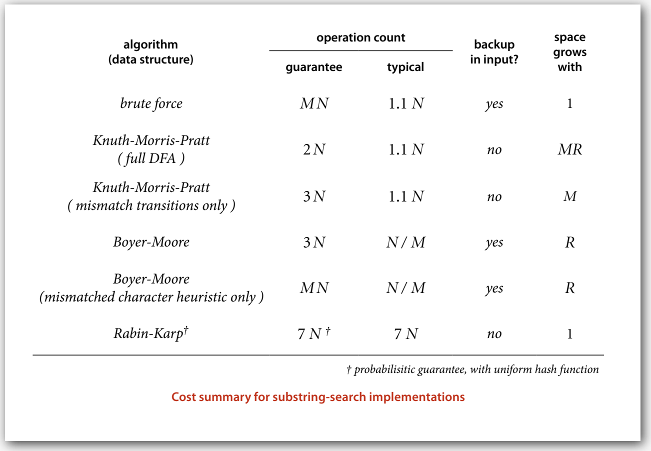
Disadvantages.

- Arithmetic ops slower than char compares.
- No worst-case guarantee.

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†	$7\ N$ †	$7\ N$	*no*	1

† probabilisitic guarantee, with uniform hash function

Cost summary for substring-search implementations