Substring Search

Introduction

> Find a pattern of length M in a text of length N

Application

- screen scraping
- spam detection

Brute Force

Basic

- check for pattern starting at each text position

Performance

- can be slow if text and pattern are repetitive. ~MN char compares
- Brute-force needs **backup** for every mismatch---should avoid backup in text stream

Implementation

```
1  public class Brute {
2
3      public static int search1(String pat, String txt) {
4          int m = pat.length();
5          int n = txt.length();
6
7          for (int i = 0; i <= n - m; i++) {
8              int j;
9              for (j = 0; j < m; j++) {
10                 if (txt.charAt(i+j) != pat.charAt(j))
11                     break;
12             }
13             if (j == m) return i;            // found at offset i
14         }
15         return n;                            // not found
16     }
17
18     // return offset of first match or N if no match
19     public static int search2(String pat, String txt) {
20         int m = pat.length();
21         int n = txt.length();
22         int i, j;
23         for (i = 0, j = 0; i < n && j < m; i++) {
24             if (txt.charAt(i) == pat.charAt(j)) j++;
25             else {
```

```
                i -= j;
                j = 0;
            }
        }
        if (j == m) return i - m;    // found
        else        return n;        // not found
    }

    public static int search1(char[] pattern, char[] 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[i+j] != pattern[j])
                    break;
            }
            if (j == m) return i;            // found at offset i
        }
        return n;                            // not found
    }

    // return offset of first match or n if no match
    public static int search2(char[] pattern, char[] text) {
        int m = pattern.length;
        int n = text.length;
        int i, j;
        for (i = 0, j = 0; i < n && j < m; i++) {
            if (text[i] == pattern[j]) j++;
            else {
                i -= j;
                j = 0;
            }
        }
        if (j == m) return i - m;    // found
        else        return n;        // not found
    }

    public static void main(String[] args) {
        String pat = args[0];
        String txt = args[1];
        char[] pattern = pat.toCharArray();
        char[] text    = txt.toCharArray();

        int offset1a = search1(pat, txt);
        int offset2a = search2(pat, txt);
        int offset1b = search1(pattern, text);
        int offset2b = search2(pattern, text);

        // print results
        StdOut.println("text:    " + txt);
```

```
78
79          // from brute force search method 1a
80          StdOut.print("pattern: ");
81          for (int i = 0; i < offset1a; i++)
82              StdOut.print(" ");
83          StdOut.println(pat);
84
85          // from brute force search method 2a
86          StdOut.print("pattern: ");
87          for (int i = 0; i < offset2a; i++)
88              StdOut.print(" ");
89          StdOut.println(pat);
90
91          // from brute force search method 1b
92          StdOut.print("pattern: ");
93          for (int i = 0; i < offset1b; i++)
94              StdOut.print(" ");
95          StdOut.println(pat);
96
97          // from brute force search method 2b
98          StdOut.print("pattern: ");
99          for (int i = 0; i < offset2b; i++)
100             StdOut.print(" ");
101         StdOut.println(pat);
102     }
103 }
```

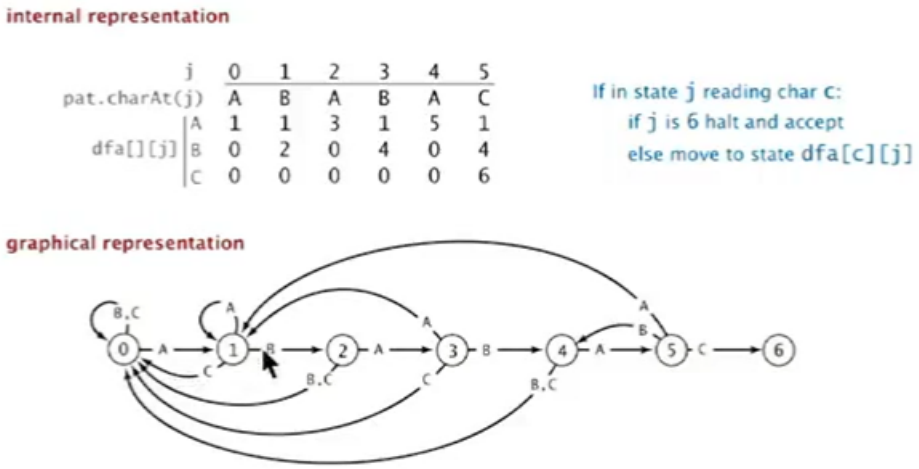
Knuth-Morris-Pratt

> clever mehod to always avoid backup

DFA

> Deterministic finite state automaton

- 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



- Mismatch transition

performance

- KMP substring search accesses no more than $M + N$ chars to search for a pattern of length M in a text of length N
- IKMP constructs dfa[][] and space proportional to RM

Implementation

```java
public class KMP {
    private final int R;       // the radix
    private final int m;       // length of pattern
    private int[][] dfa;       // the KMP automaton

    public KMP(String pat) {
        this.R = 256;
        this.m = pat.length();

        // build DFA from pattern
        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.
        }
    }

    public KMP(char[] pattern, int R) {
        this.R = R;
        this.m = pattern.length;

        // build DFA from pattern
        int m = pattern.length;
        dfa = new int[R][m];
        dfa[pattern[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[pattern[j]][j] = j+1;      // Set match case.
            x = dfa[pattern[j]][x];        // Update restart state.
        }
    }

    public int search(String txt) {

        // simulate operation of DFA on text
        int n = txt.length();
        int i, j;
        for (i = 0, j = 0; i < n && j < m; i++) {
            j = dfa[txt.charAt(i)][j];
        }
        if (j == m) return i - m;    // found
        return n;                    // not found
    }

    public int search(char[] text) {

        // simulate operation of DFA on text
        int n = text.length;
        int i, j;
        for (i = 0, j = 0; i < n && j < m; i++) {
            j = dfa[text[i]][j];
        }
        if (j == m) return i - m;    // found
        return n;                    // not found
    }

    public static void main(String[] args) {
        String pat = args[0];
        String txt = args[1];
        char[] pattern = pat.toCharArray();
        char[] text    = txt.toCharArray();

        KMP kmp1 = new KMP(pat);
        int offset1 = kmp1.search(txt);

        KMP kmp2 = new KMP(pattern, 256);
        int offset2 = kmp2.search(text);
    }
}
```

Boyer-Moore

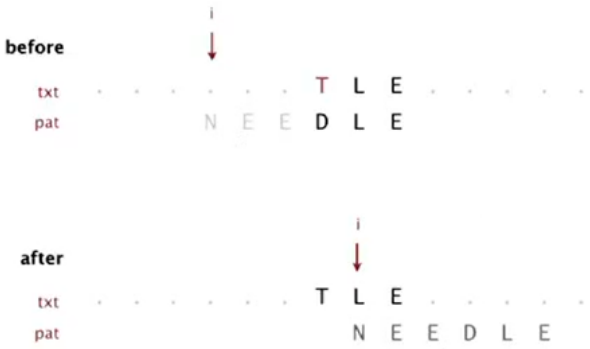
Basic

> Mismatch character heuristic

- scan characters in pattern from righ o left
- can skip as many as M text chars when finding one not in the pattern

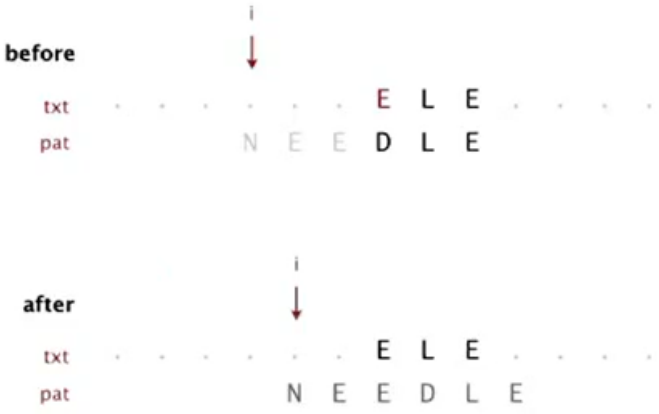
Mismatch cases

- mismatch character not in pattern



- mismatch character in pattern

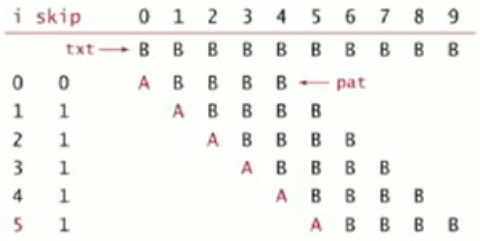
Case 2b. Mismatch character in pattern (but heuristic no help).



Performance

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 ~ $M N$.



Implementation

```java
public class BoyerMoore {
    private final int R;     // the radix
    private int[] right;     // the bad-character skip array

    private char[] pattern;  // store the pattern as a character array
    private String pat;      // or as a string

    public BoyerMoore(String pat) {
        this.R = 256;
        this.pat = pat;

        // position of rightmost occurrence of c in the pattern
        right = new int[R];
        for (int c = 0; c < R; c++)
            right[c] = -1;
        for (int j = 0; j < pat.length(); j++)
            right[pat.charAt(j)] = j;
    }

    public BoyerMoore(char[] pattern, int R) {
        this.R = R;
        this.pattern = new char[pattern.length];
        for (int j = 0; j < pattern.length; j++)
            this.pattern[j] = pattern[j];

        // position of rightmost occurrence of c in the pattern
        right = new int[R];
        for (int c = 0; c < R; c++)
            right[c] = -1;
        for (int j = 0; j < pattern.length; j++)
            right[pattern[j]] = j;
    }

```

```
    public int search(String txt) {
        int m = pat.length();
        int n = txt.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;    // found
        }
        return n;                       // not found
    }

    public int search(char[] text) {
        int m = pattern.length;
        int n = text.length;
        int skip;
        for (int i = 0; i <= n - m; i += skip) {
            skip = 0;
            for (int j = m-1; j >= 0; j--) {
                if (pattern[j] != text[i+j]) {
                    skip = Math.max(1, j - right[text[i+j]]);
                    break;
                }
            }
            if (skip == 0) return i;    // found
        }
        return n;                       // not found
    }

    public static void main(String[] args) {
        String pat = args[0];
        String txt = args[1];
        char[] pattern = pat.toCharArray();
        char[] text    = txt.toCharArray();

        BoyerMoore boyermoore1 = new BoyerMoore(pat);
        BoyerMoore boyermoore2 = new BoyerMoore(pattern, 256);
        int offset1 = boyermoore1.search(txt);
        int offset2 = boyermoore2.search(text);

        // print results
        StdOut.println("text:    " + txt);

        StdOut.print("pattern: ");
        for (int i = 0; i < offset1; i++)
            StdOut.print(" ");
        StdOut.println(pat);
```

```
86
87          StdOut.print("pattern: ");
88          for (int i = 0; i < offset2; i++)
89              StdOut.print(" ");
90          StdOut.println(pat);
91      }
92  }
```

Rabin-Karp

Basic

> modular hashing

- 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

Two Versions

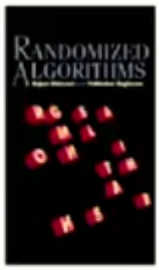
- Monte Carlo version---return match if hash mathes
- Las Vegas version---check for substring match if hash matches; continue search if false collision

Theory. If Q is a sufficiently large random prime (about $M\,N^2$), then the probability of a false collision is about $1 / N$.

Practice. Choose Q to be a large prime (but not so large to cause overflow). Under reasonable assumptions, probability of a collision is about $1 / Q$.

Monte Carlo version.

- Always runs in linear time.
- Extremely likely to return correct answer (but not always!).



Las Vegas version.

- Always returns correct answer.
- Extremely likely to run in linear time (but worst case is $M\,N$).

Modular 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.

Implementation

```
import java.math.BigInteger;
import java.util.Random;

public class RabinKarp {
    private String pat;      // the pattern  // needed only for Las Vegas
    private long patHash;    // pattern hash value
    private int m;           // pattern length
    private long q;          // a large prime, small enough to avoid long
overflow
    private int R;           // radix
    private long RM;         // R^(M-1) % Q

    public RabinKarp(char[] pattern, int R) {
        this.pat = String.valueOf(pattern);
        this.R = R;
        throw new UnsupportedOperationException("Operation not supported yet");
    }

    public RabinKarp(String pat) {
        this.pat = pat;      // save pattern (needed only for Las Vegas)
        R = 256;
        m = pat.length();
        q = longRandomPrime();

        // precompute R^(m-1) % q for use in removing leading digit
        RM = 1;
        for (int i = 1; i <= m-1; i++)
            RM = (R * RM) % q;
        patHash = hash(pat, m);
    }

    // Compute hash for key[0..m-1].
    private long hash(String key, int m) {
        long h = 0;
        for (int j = 0; j < m; j++)
            h = (R * h + key.charAt(j)) % q;
        return h;
    }

    // Las Vegas version: does pat[] match txt[i..i-m+1] ?
```

```
    private boolean check(String txt, int i) {
        for (int j = 0; j < m; j++)
            if (pat.charAt(j) != txt.charAt(i + j))
                return false;
        return true;
    }

    public int search(String txt) {
        int n = txt.length();
        if (n < m) return n;
        long txtHash = hash(txt, m);

        // check for match at offset 0
        if ((patHash == txtHash) && check(txt, 0))
            return 0;

        // check for hash match; if hash match, check for exact match
        for (int i = m; i < n; i++) {
            // Remove leading digit, add trailing digit, check for match.
            txtHash = (txtHash + q - RM*txt.charAt(i-m) % q) % q;
            txtHash = (txtHash*R + txt.charAt(i)) % q;

            // match
            int offset = i - m + 1;
            if ((patHash == txtHash) && check(txt, offset))
                return offset;
        }
        // no match
        return n;
    }


    // a random 31-bit prime
    private static long longRandomPrime() {
        BigInteger prime = BigInteger.probablePrime(31, new Random());
        return prime.longValue();
    }

    public static void main(String[] args) {
        String pat = args[0];
        String txt = args[1];

        RabinKarp searcher = new RabinKarp(pat);
        int offset = searcher.search(txt);

        // print results
        StdOut.println("text:    " + txt);

        // from brute force search method 1
        StdOut.print("pattern: ");
        for (int i = 0; i < offset; i++)
            StdOut.print(" ");
```

```
92            StdOut.println(pat);
93        }
94    }
```

Comparison

algorithm	version	operation count		backup in input?	correct?	extra space
		guarantee	typical			
brute force	—	MN	$1.1\,N$	*yes*	*yes*	1
Knuth-Morris-Pratt	*full DFA (Algorithm 5.6)*	$2\,N$	$1.1\,N$	*no*	*yes*	MR
	mismatch transitions only	$3\,N$	$1.1\,N$	*no*	*yes*	M
Boyer-Moore	*full algorithm*	$3\,N$	N / M	*yes*	*yes*	R
	mismatched char heuristic only (Algorithm 5.7)	MN	N / M	*yes*	*yes*	R
Rabin-Karp†	*Monte Carlo (Algorithm 5.8)*	$7\,N$	$7\,N$	*no*	*yes*†	1
	Las Vegas	$7\,N$†	$7\,N$	*yes*	*yes*	1