String Searching

String search. Given a pattern string, find first match in text.

Model: Can’t afford to preprocess the text.

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

Applications

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

Brute Force: Typical Case

```
hayneedle

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

M = 6, N = 21
Brute Force

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

```
public static int search(String pattern, String text) {
    int M = pattern.length();
    int N = text.length();
    for (int i = 0; i < N - M; i++) {
        int j;
        for (j = 0; j < M; j++) {
            if (text.charAt(i+j) != pattern.charAt(j))
                break;
        } if (j == M) return i; // return offset i of match
    } return -1; // not found
}
```

Analysis of Brute Force

Analysis of brute force.
- Running time depends on pattern and text.
- Slow if M and N are large, and have lots of repetition.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Typical</th>
<th>Worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brute</td>
<td>11N</td>
<td>MN</td>
</tr>
</tbody>
</table>

Search for M-character pattern in N-character text

↑ assumes appropriate model

Screen Scraping

Goal. Find current stock price of Google.

http://finance.yahoo.com/q?s=goog NYSE symbol
Screen Scrapeing

**Goal.** Find current stock price of Google.
- `s.indexOf(t, i)`; index of first occurrence of pattern `t` in string `s`, starting at offset `i`.
- Find first string delimited by `<b>` and `</b>` after `Last Trade`.

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

### Algorithmic Challenges

**Theoretical challenge.** Linear-time guarantee.

**Practical challenge.** Avoid backup.

Often no room or time to save text.

Karp-Rabin Randomized Fingerprint Algorithm

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

**Ex.** Hash "table" size = 97.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Text</th>
<th>Product</th>
</tr>
</thead>
</table>
| 5 9 2 6 5 | 59265 % 97 = 95

---

Dick Karp  
1985 Turing award

Michael Rabin
Computing the Hash Function

**Brute force.** $O(M)$ arithmetic ops per hash.

**Faster method** to compute hash of adjacent substrings.
- Use previous hash to compute next hash.
- $O(1)$ time per hash.  

**Ex.**
- Pre-computed: $10000 \% 97 = 9$
- Previous hash: $41592 \% 97 = 76$
- Next hash: $15926 \% 97 = ??$

**Observation.**
- $15926 \% 97 = (41592 - (4 \times 10000)) \times 10 + 6$
- $76 - (4 \times 9 ) \times 10 + 6$
- 406
- 18

Karp-Rabin: False Matches

**False match.** Hash of pattern collides with another substring.
- $59265 \% 97 = 95$
- $59362 \% 97 = 95$

How to choose modulus $p$.
- $p$ too small $\Rightarrow$ many false matches.
- $p$ too large $\Rightarrow$ too much arithmetic.
- **Ex:** $p = 8359567$ $\Rightarrow$ avoid 32-bit integer overflow.
- **Ex:** $p = 35888607147294757$ $\Rightarrow$ avoid 64-bit integer overflow.

Java Implementation

```java
public static int search(String p, String t) {
    int M = p.length(), N = t.length();
    int q = 8355967;  // table size
    int d = 256;      // radix
    int dm = 1;       // precompute $d^{(M-1)} \% q$
    for (int j = 1; j < M; j++)
        dm = (d * dm) \% q;

    int h1 = 0, h2 = 0;
    for (int i = 0; i < M; i++) {
        h1 = (h1*d + p.charAt(i)) \% q;  // hash of pattern
        h2 = (h2*d + t.charAt(i)) \% q;  // hash of text
        if (h1 == h2) return i;  // found
    }
    return -1;  // not found
}
```

Karp-Rabin: Randomized Algorithms

**Theorem.** If $MN \geq 29$ and $p$ is a random prime between $1$ and $MN^2$, then
\[
Pr[\text{false match}] \leq \frac{2.53}{N}.
\]

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

**Monte Carlo.** Don’t bother checking for false matches.
- Guaranteed to be fast: $O(M + N)$.
- Expected to be correct (but false match possible).

**Las Vegas.** Upon hash match, do full compare; if false match, try again with new random prime.
- Guaranteed to be correct.
- Expected to be fast: $O(M + N)$.

**Q.** Would either version of Rabin-Karp make a good library function?
String Search Implementation Cost Summary

Karp-Rabin summary.
- Create fingerprint of each substring and compare fingerprints.
- Expected running time is linear.
- Idea generalizes, e.g., to 2D patterns.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Typical</th>
<th>Worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brute</td>
<td>$1.1 N^t$</td>
<td>$MN$</td>
</tr>
<tr>
<td>Karp-Rabin</td>
<td>$\Theta(N)$</td>
<td>$\Theta(N)^t$</td>
</tr>
</tbody>
</table>

Search for $M$-character pattern in $N$-character text

Knuth-Morris-Pratt: DFA Simulation

KMP algorithm. [over binary alphabet]
- Build DFA from pattern.
- Run DFA on text.

Interpretation of state $i$. Length of longest prefix of search pattern that is a suffix of input string.

Ex. End in state 4 iff text ends in aaba.
Ex. End in state 2 iff text ends in aa (but not aabaa or aabaaa).
DFA Representation

DFA used in KMP has special property.
- Upon character match in state \( j \), go forward to state \( j+1 \).
- Upon character mismatch in state \( j \), go back to state \( \text{next}[j] \).

<table>
<thead>
<tr>
<th>Pattern</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>next</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

DFA used in KMP has special property.
Upon character match in state \( j \), go forward to state \( j+1 \).
Upon character mismatch in state \( j \), go back to state \( \text{next}[j] \).

Key idea. Simulate aabaaaa on DFA.

Two key differences from brute force.
- Text pointer \( i \) never "backs up."
- Need to precompute \( \text{next}[\cdot] \) table.

```c
int j = 0;
for (int i = 0; i < N; i++) {
    if (t.charAt(i) == p.charAt(j)) j++;     // match
    else j = next[j];                       // mismatch
    if (j == M) return i - M + 1;          // found
}
return -1;                               // not found
```

Simulation of KMP DFA (assumes binary alphabet)

Knuth-Morris-Pratt: DFA Construction

Iterative construction. Suppose you’ve created DFA for pattern aabaaa.
How to extend to DFA for pattern aabaaa?
- Easy: transition from state 6 if next char matches.
- Challenge: transition from state 6 if next char mismatches.

Wishful thinking. Simulate aabaaa on DFA.

Key idea. Simulate xabaaa on DFA.

Knuth-Morris-Pratt: DFA Construction

Iterative construction. Suppose you’ve created DFA for pattern aabaaa.
How to extend to DFA for pattern aabaaa?
- Easy: transition from state 6 if next char matches.
- Challenge: transition from state 6 if next char mismatches.

Wishful thinking. Simulate aabaaa on DFA.

Key idea. Simulate xabaaa on DFA.

Efficient version. Pre-compute simulation of xabaaa.
Knuth-Morris-Pratt: DFA Construction

DFA construction for KMP. DFA builds itself!

State 6. Given DFA for aabaa and state X of simulating xabaab, compute DFA for xabaab and state X of simulating xabaab.

- \( \text{next}[6] = X \rightarrow a = 2 \)
- Update \( X = X \rightarrow b = 3 \)

DFA Construction for KMP: Java Implementation

Build DFA for KMP.

- Takes \( O(M) \) time.
- Requires \( O(M) \) extra space to store \( \text{next}[j] \) table.

```java
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)
KMP Over Arbitrary Alphabet

DFA for patterns over arbitrary alphabet Σ.
- For each character in alphabet, determine next state.
- Lookup table requires $O(M \cdot |\Sigma|)$ space.

<table>
<thead>
<tr>
<th>character comparisons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementation</td>
</tr>
<tr>
<td>Brute</td>
</tr>
<tr>
<td>Karp-Rabin</td>
</tr>
<tr>
<td>KMP</td>
</tr>
</tbody>
</table>

String Search Implementation Cost Summary

KMP analysis.
- NFA simulation requires at most $2N$ comparisons.
  - advances $\leq N$
  - retreats $= \text{advances}$
- NFA construction takes $\Theta(M)$ time and space.

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 a text editor.

Resolved theoretical and practical problems.
- Surprise when it was discovered.
- In hindsight, seems like right algorithm.
Bad Character Rule

- Use right-to-left scanning.
- Upon mismatch of text character $c$, increase offset so that character $c$ in pattern lines up with text character $c$.
- Precompute $\text{right}[c] =$ rightmost occurrence of $c$ in pattern.

<table>
<thead>
<tr>
<th>right[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>c 3</td>
</tr>
<tr>
<td>k 4</td>
</tr>
<tr>
<td>1 1</td>
</tr>
<tr>
<td>o 2</td>
</tr>
<tr>
<td>* -1</td>
</tr>
</tbody>
</table>

Boyer-Moore

Right-to-Left Scanning

Right-to-left scanning.
- Find offset $i$ in text by moving left to right.
- Compare pattern to text by moving $j$ right to left.

hickory, dickory, dock, clock.

<table>
<thead>
<tr>
<th>clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>clock</td>
</tr>
<tr>
<td>clock</td>
</tr>
<tr>
<td>clock</td>
</tr>
<tr>
<td>clock</td>
</tr>
<tr>
<td>clock</td>
</tr>
</tbody>
</table>

Bob Boyer
J. Strother Moore
Strong Good Suffix Rule

[Strong suffix rule. [KMP-like suffix rule]]
Right-to-left scanning.
Suppose text matches suffix $t$ of pattern but mismatches in previous character $c$.
Find rightmost copy of $t$ in pattern whose preceding letter is not $c$, and shift; if no such copy, shift $M$ positions.

$t = "ab"
$M = 0; c = 'b'$

```
public static int search(String pattern, String text) {
    int M = pattern.length(), N = text.length();
    int[] right = new int[256];
    for (int c = 0; c < 256; c++) right[c] = -1;
    for (int j = 0; j < M; j++) right[Pattern.charAt(j)] = j;

    int i = 0; // offset
    while (i < N - M) {
        int skip = 0;
        for (int j = M - 1; j >= 0; j--) {
            if (Pattern.charAt(j) != text.charAt(i + j)) {
                skip = Math.max(skip, j - right[Pattern.charAt(i + j)]);
            }
        }
        if (skip == 0) return i; // found
        i = i + skip;
    }
    return -1;
}
```

Bad Character Rule: Java Implementation

```
public static int search(String pattern, String text) {
    int M = pattern.length(), N = text.length();
    int[] right = new int[256];
    for (int c = 0; c < 256; c++) right[c] = -1;
    for (int j = 0; j < M; j++) right[Pattern.charAt(j)] = j;

    int i = 0; // offset
    while (i < N - M) {
        int skip = 0;
        for (int j = M - 1; j >= 0; j--) {
            if (Pattern.charAt(j) != text.charAt(i + j)) {
                skip = Math.max(skip, j - right[Pattern.charAt(i + j)]);
            }
        }
        if (skip == 0) return i; // found
        i = i + skip;
    }
    return -1;
}
```

Strong good suffix rule. [KMP-like suffix rule]
- Right-to-left scanning.
- Suppose text matches suffix $t$ of pattern but mismatches in previous character $c$.
- Find rightmost copy of $t$ in pattern whose preceding letter is not $c$, and shift; if no such copy, shift $M$ positions.

```
x x x x x x b a b ? ? ? ? ? x x x x x x x x
x c a b d a b d a b
x c a b d a b d a b
```

Boyero-Moore

```
```

Bad character rule analysis.
- Highly effective in practice, particularly for English text: $O(N/M)$.
- Takes $\Omega(MN)$ time in worst case.

```
```

```
```

Boyer-Moore analysis.
- $O(N/M)$ average case if given letter usually doesn't occur in string.
  - time decreases as pattern length increases
  - sublinear in input size!
- At most $3N$ comparisons to find a match.

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```
String Search Implementation Cost Summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Typical</th>
<th>Worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brute</td>
<td>(1.1N)</td>
<td>(MN)</td>
</tr>
<tr>
<td>Karp-Rabin</td>
<td>(\Theta(N))</td>
<td>(\Theta(N))</td>
</tr>
<tr>
<td>KMP</td>
<td>(1.1N)</td>
<td>(2N)</td>
</tr>
<tr>
<td>Boyer-Moore</td>
<td>(N/M)</td>
<td>(3N)</td>
</tr>
</tbody>
</table>

Search for \(M\)-character pattern in \(N\)-character text

† assumes appropriate model
‡ randomized

Finding All Matches

**Karp-Rabin.** Can find all matches in \(O(M + N)\) expected time using Muthukrishnan variant.

**Knuth-Morris-Pratt.** Can find all matches in \(O(M + N)\) time via simple modification.

**Boyer-Moore.** Can find all matches in \(O(M + N)\) time using Galil variant.

Boyé-Moore and Alphabet Size

**Boyer-Moore space requirement.** \(O(M + |\Sigma|)\)

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

**Small alphabets.**
- Loses effectiveness when \(\Sigma\) is too small, e.g., DNA.
- Fix: group characters together, e.g., \(aaaa, aace, \ldots\).

Multiple String Search

**Multiple string search.** Search for any of \(k\) different patterns.
- Naïve KMP: \(O(kN + M_1 + \ldots + M_k)\).
- Aho-Corasick: \(O(N + M_1 + \ldots + M_k)\).
- Ex: screen out dirty words from a text stream.

DFA for (aaa or abb or baba)