String Searching

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

Model. Can’t afford to preprocess the text.

Parameters. \( N = \text{length of text}, M = \text{length of pattern}. \)

Typically \( N \gg M \)

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

Typical Case

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: $M \times N$ comparisons.
- "Average" case: $1.1 \times N$ comparisons. (!)
- Slow if $M$ and $N$ are large, and have lots of repetition.

Screen Scraping

Find current stock price of Google.
- `t.indexOf(p)`: index of 1st occurrence of pattern `p` in text `t`.
- Download html from: `http://finance.yahoo.com/q?s=goog`
- Find first 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.readString();
        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 goog
357.36
```
Algorithmic Challenges

Theoretical challenge. Linear-time guarantee.

Practical challenge. Avoid backup.

Karp-Rabin

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>Search Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 9 2 6 5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Search Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>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</td>
</tr>
</tbody>
</table>

59265 % 97 = 95

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, except first one.

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

Robert Sedgewick and Kevin Wayne · Copyright © 2005 · http://www.Princeton.EDU/~cs226
Karp-Rabin: Randomized Algorithms

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

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

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

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

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: $p = 8355967$ $\Rightarrow$ avoid 32-bit integer overflow.
- Ex: $p = 35888607147294757$ $\Rightarrow$ avoid 64-bit integer overflow.

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

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>Character comparisons</th>
<th>Implementation</th>
<th>Typical</th>
<th>Worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brute</td>
<td>$\Omega(N)$</td>
<td>$\Omega(N)$</td>
<td></td>
</tr>
<tr>
<td>Karp-Rabin</td>
<td>$\Omega(N)$</td>
<td>$\Omega(N)$</td>
<td></td>
</tr>
</tbody>
</table>

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

\[ \backslash \text{relies on prime number theorem} \]
Knuth-Morris-Pratt

How To Save Comparisons

How to avoid re-computation?

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

Knuth-Morris-Pratt: DFA Simulation

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

Knuth-Morris-Pratt: DFA Simulation

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 aaab, Ex. End in state 2 iff text ends in aa (but not aabaa or aabaaa).
**DFA Representation**

**KMP algorithm.**  [over binary alphabet]

- 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 suffix of aabab.
- next[5]: longest prefix of aabaaa that is a suffix of aabaaa.

**DFA Construction for KMP**

**KMP Algorithm**

Two key differences from brute force.

- Text pointer i never "backs up."
- Need to precompute next[] table.

```java
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)**

**DFA Construction**

**Knuth-Morris-Pratt: DFA Construction**

**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 \( \text{next}[j] \).

**Search Pattern**

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
\hline
a & 1 & 2 & 4 & 5 & 6 \\
b & 0 & 0 & 3 & 0 & 0 \\
\end{array}
\]

**Simulation**

**DFA Construction for KMP.**  DFA builds itself!

**Ex.**  Compute `next[6]` for pattern \( p[0..6] = \text{aabaaa} \).

- Assume you know DFA for pattern \( p[0..5] = \text{aabaa} \).
- Assume you know state \( X \) for \( p[1..5] = \text{abaa} \).
- Update \( X \) to state for \( p[1..6] = \text{aabaaa} \).
DFA Construction for KMP

DFA construction for KMP. DFA builds itself!

Ex. Compute \( \text{next}[7] \) for pattern \( p[0..7] = \text{aabaaab} \).

- Assume you know DFA for pattern \( p[0..6] = \text{aabaaab} \).
- Assume you know state \( X \) for \( p[1..6] = \text{abaaab} \), \( X = 3 \)
- Update \( \text{next}[7] \) to state for \( \text{abaaabb} \), \( X + a = 4 \)
- Update \( X \) to state for \( p[1..7] = \text{abaaabb} \), \( X + b = 0 \)

DFA Construction for KMP (assumes binary alphabet)

Build DFA for KMP.
- Takes \( O(M) \) time.
- Requires \( O(M) \) extra space to store \text{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)
KMP Over Arbitrary Alphabet

DFA for patterns over arbitrary alphabet $\Sigma$.
- For each character in alphabet, determine next state.
- Lookup table requires $O(M |\Sigma|)$ space.
- Can be expensive if $\Sigma = \text{Unicode alphabet}$

Ex. DFA for pattern $ababch$.

---

NFA for patterns over arbitrary alphabet $\Sigma$.
- Read new character only upon success (or failure at beginning).
- Reuse current character upon failure and follow back.

Ex. NFA for pattern $ababch$.

---

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.
- Good efficiency for patterns and texts with much repetition.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Typical</th>
<th>Worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brute</td>
<td>$1.1N^\dagger$</td>
<td>$MN$</td>
</tr>
<tr>
<td>Karp-Rabin</td>
<td>$\Theta(N)$</td>
<td>$\Theta(N)^\ddagger$</td>
</tr>
<tr>
<td>KMP</td>
<td>$1.1N^\dagger$</td>
<td>$2N$</td>
</tr>
</tbody>
</table>

$^\dagger$ assumes appropriate model
$^\ddagger$ randomized

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

---

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

**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.

```
right
 c 3
 k 4
 1 1
 o 2
* -1
```

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
  c lo c k
clock
clock
clock
clock
```

**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.
Bad Character Rule: Java Implementation

```java
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[text.charAt(i + j)]);
            }
        }
        if (skip == 0) return i; // found
        i = i + skip;
    }
    return -1;
}
```

Bad Character Rule: Analysis

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

Bad character rule analysis.

- Strong good suffix rule. [a 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"
c = 'b'
```

Bad Character Rule: Analysis

- Boyer-Moore.
  - Right-to-left scanning.
  - Bad character rule.
  - Strong good suffix rule.

```text
Boyero-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.
```

Boyero-Moore in the wild. Unix grep, emacs.
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>

† assumes appropriate model
‡ randomized

Search for M-character pattern in N-character text

Boyer-Moore and Alphabet Size

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

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, aaac, ...

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.

Multiple String Search

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

search pattern: aabaaa

accept state

search pattern: aaaa or abb or baba
Java String Library

Java string library has built-in string searching.

- indexOf(p): index of 1st occurrence of pattern p in text t.
- Caveat: it’s brute force, and can take $\Omega(MN)$ time.

```java
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));
}
```

Q. Why does Java string library use brute force?

Tip of the Iceberg

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

Approximate string matching: allow up to $k$ mismatching chars.
- Ex: fix transmission errors in signal processing.
- Ex: recover from typing or spelling errors in information retrieval.

Edit-distance: allow up to $k$ edits.
- Recover from measurement errors in computational biology.

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.
- Extends to more general settings (e.g., multiple string search).

Boyer-Moore.
- Simple idea leads to dramatic speedup for long patterns.
- Running time depends on alphabet size.
- Need to tweak for small or large alphabets.