String Searching

Karp-Rabin
Knuth-Morris-Pratt
Boyer-Moore


String Search

String search: given a pattern string p, find first match in text t.
Model: can't afford to preprocess the text.

\[ N = \text{# characters in text} \quad M = \text{# characters in pattern} \]
typically \( N \gg M \)
Ex: \( N = 1 \text{ million}, M = 100 \)

\[
\begin{array}{cccccccccccc}
 n & n & e & e & n & l & e & d & e & n & e & n & e & d & l & e & n & l & d \\
\end{array}
\]

\[
\begin{array}{cccccccccccc}
 n & e & e & d & l & e \\
\end{array}
\]

\[
\begin{array}{cccccccccccc}
 n & n & e & e & n & l & e & d & e & n & e & n & e & n & e & d & l & e & n & l & d \\
\end{array}
\]

Successful Search

Spam Filtering

Spam filtering: patterns indicative of spam.
- AMAZING
- GUARANTEE
- PROFITS
- herbal Viagra
- This is a one-time mailing.
- There is no catch.
- This message is sent in compliance with spam regulations.
- You're getting this message because you registered with one of our marketing partners.
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 -1;  // return -1 if not found
}
```

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 Sun Microsystems.
- `t.indexOf(p)`: index of 1st occurrence of pattern p in text t.
- Find 1st 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.readAll();
        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);
    }
}
```

Algorithmic Challenges

- Theoretical challenge: linear-time guarantee.
  - TST index costs $\sim N \lg N$.
- Practical challenge: avoid BACKUP.
  - Often no room or time to save text.

Fundamental algorithmic problem.
Karp-Rabin Fingerprint Algorithm

Idea: use hashing.

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

Example.
- Hash "table" size = 97.

Search Pattern

| 5 | 9 | 2 | 6 | 5 |

Example.
- Hash "table" size = 97.

Search Text

| 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 | 3 | 5 | 8 | 9 | 7 | 9 | 3 | 2 | 3 | 8 | 4 | 6 |

- 31415 % 97 = 84
- 14159 % 97 = 94
- 41592 % 97 = 76
- 15926 % 97 = 18
- 59265 % 97 = 95

Key idea: fast to compute hash function of adjacent substrings.
- Use previous hash to compute next hash.
- O(1) time per hash, except first one.

Example.
- Pre-compute: 10000 % 97 = 9
- Previous hash: 41592 % 97 = 76
- Next hash: 15926 % 97 = 76

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

public static int search(String p, String t) {
    int M = p.length();
    int N = t.length();
    int dM = 1, h1 = 0, h2 = 0;
    int q = 3355439; // table size
    int d = 256; // radix
    for (int j = 1; j < M; j++)         // precompute d^M % q
        dM = (d * dM) % q;
    for (int j = 0; j < M; j++) {
        h1 = (h1*d + p.charAt(j)) % q; // hash of pattern
        h2 = (h2*d + t.charAt(j)) % q; // hash of text
    }
    if (h1 == h2) return i - M; // match found
    for (int i = M; j < N; i++) {
        h2 = (h2 - t.charAt(i-M)) % q; // remove high order digit
        h2 = (h2*d + t.charAt(i)) % q; // insert low order digit
        if (h1 == h2) return i - M; // match found
    }
    return -1;                          // not found
String Search Implementation Cost Summary

Karp-Rabin fingerprint algorithm.
- Choose table size at random to be huge prime.
- Expected running time is $O(M + N)$.
- $\Theta(MN)$ worst-case, but this is (unbelievably) unlikely.

Main advantage. Extends to 2d patterns and other generalizations.

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

<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>
</tbody>
</table>

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

character comparisons

Randomized Algorithms

A randomized algorithm uses random numbers to gain efficiency.

Las Vegas algorithms.
- Expected to be fast.
- Guaranteed to be correct.
- Ex: quicksort, randomized BST, Rabin-Karp with match check.

Monte Carlo algorithms.
- Guaranteed to be fast.
- Expected to be correct.
- Ex: Rabin-Karp without match check.

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

How To Save Comparisons

How to avoid re-computation?
- Pre-analyze search pattern.
- Ex: suppose that first 5 characters of pattern 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

Basic strategy: pre-compute something based on pattern.

Knuth-Morris-Pratt (over binary alphabet)

KMP algorithm.
- Use knowledge of how search pattern repeats itself.
- Build DFA from pattern.
- Run DFA on text.
### DFA Representation

DFA used in KMP has special property.

- Upon character match, go forward one state.
- Only need to keep track of where to go upon character mismatch: go to state next[j] if character mismatches in state j

### KMP Algorithm

Two key differences from brute force.

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

```java
for (int i = 0, j = 0; i < N; i++) {
    if (t.charAt(i) == p.charAt(j)) j++;
    else j = next[j];
}
```

Simulation of KMP DFA (assumes binary alphabet)

### DFA Construction for KMP

DFA construction for KMP. DFA builds itself!


- Assume you know DFA for pattern p[0..5] = aabaa.
- Assume you know state X for p[1..5] = aabaa,
  - Update X to state for p[1..6] = abaaa, X + a = 2
  - Update next[6] to state for abaaa, X + b = 3

Simulation of DFA construction for KMP
DFA Construction for KMP

DFA construction for KMP. DFA builds itself!

   - Assume you know DFA for pattern p[0..5] = aabaa.
   - Assume you know state X for p[1..5] = aabaa, \( X = 2 \)
   - Update next[6] to state for abaa. \( X + a = 2 \)
   - Update X to state for p[1..6] = aabaaab \( X + b = 3 \)

Crucial insight.

To compute transitions for state n of DFA, suffices to have:
- DFA for states 0 to n-1
- state X that DFA ends up in with input \( p[1..n-1] \)

To compute state X' that DFA ends up in with input \( p[1..n] \), it suffices to have:
- DFA for states 0 to n-1
- state X that DFA ends up in with input \( p[1..n-1] \)
DFA Construction for KMP

Build DFA for KMP.
- Takes $O(M)$ time.
- Requires $O(M)$ extra space to store $next[]$ 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)

Optimized KMP Implementation

Ultimate search program for $aabaaabb$ pattern.
- Specialized C program.
- Machine language version of C program.

```c
int kmpearch(char t[]) {
    int i = 0;
    s0: if (t[i++] != 'a') goto s0;
    s1: if (t[i++] != 'a') goto s0;
    s2: if (t[i++] != 'b') goto s2;
    s3: if (t[i++] != 'a') goto s0;
    s4: if (t[i++] != 'a') goto s0;
    s5: if (t[i++] != 'a') goto s3;
    s6: if (t[i++] != 'b') goto s2;
    s7: if (t[i++] != 'b') goto s4;
    return i - 8;
}
```

assumes pattern is in text (o/w use sentinel)

KMP Over Arbitrary Alphabet

DFA for patterns over arbitrary alphabets.
- Read new character only upon success (or failure at beginning).
- Reuse current character upon failure and follow back.
- Fact: KMP follows at most $1 + \log_{10} M$ back links in a row.
- Theorem: at most $2N$ character comparisons in total.

Ex: DFA for pattern $ababc$, 

N
a b c d e f

String Search Implementation Cost Summary

KMP analysis.
- DFA simulation takes \( \Theta(N) \) time in worst-case.
- DFA construction takes \( \Theta(M) \) time and space in worst-case.
- Extends to ASCII or UNICODE alphabets.
- Good efficiency for patterns and texts with much repetition.
- "On-line algorithm." virus scanning, internet spying

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

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Typical</th>
<th>Worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brute</td>
<td>( \Omega(N) ) †</td>
<td>( M ) N</td>
</tr>
<tr>
<td>Karp-Rabin</td>
<td>( \Theta(N) ) ‡</td>
<td>( \Theta(N) ) ‡</td>
</tr>
<tr>
<td>KMP</td>
<td>( \Omega(N) ) †</td>
<td>( 2 ) N</td>
</tr>
</tbody>
</table>

† assumes appropriate model
‡ randomized

Boyер-Mоore

- Right-to-left scanning.
  - find offset \( i \) in text by moving left to right.
  - compare pattern to text by moving right to left.

Index

<table>
<thead>
<tr>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
</tr>
<tr>
<td>i</td>
</tr>
<tr>
<td>n</td>
</tr>
<tr>
<td>s</td>
</tr>
<tr>
<td>t</td>
</tr>
<tr>
<td>*</td>
</tr>
</tbody>
</table>

History of KMP

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.

Boyер-Mоore

- Right-to-left scanning.
  - Heuristic 1: advance offset \( i \) using "bad character rule."
  - upon mismatch of text character \( c \), look up \( \text{index}[c] \)
  - increase offset \( i \) so that \( j \)th character of pattern lines up with text character \( c \)

Mismatch
Match
No comparison
Boyer-Moore

- Right-to-left scanning.
- Heuristic 1: advance offset \( i \) using "bad character rule."
  - upon mismatch of text character \( c \), look up \( \text{index}[c] \)
  - increase offset \( i \) so that \( j \)th character of pattern lines up with text character \( c \)

```
private static void badCharSkip(String pattern, int[] skip) {
    int M = pattern.length();
    for (int j = 0; j < 256; j++)
        skip[j] = M;
    for (int j = 0; j < M-1; j++)
        skip[pattern.charAt(j)] = M-j-1;
}
```

construction of bad character skip table

---

String Search Implementation Cost Summary

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!
- \( O(N) \) worst-case with Galil variant.

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

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Typical</th>
<th>Worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brute</td>
<td>( 1.1 N )†</td>
<td>( MN )</td>
</tr>
<tr>
<td>Karp-Rabin</td>
<td>( O(N) )</td>
<td>( O(N) )‡</td>
</tr>
<tr>
<td>KMP</td>
<td>( 1.1 N )†</td>
<td>( 2N )</td>
</tr>
<tr>
<td>Boyer-Moore</td>
<td>( N / M )†</td>
<td>( 4N )</td>
</tr>
</tbody>
</table>

† assumes appropriate model
‡ randomized

---

Right-to-left scanning.
- Heuristic 1: advance offset \( i \) using "bad character rule."
- Heuristic 2: use KMP-like suffix rule.
  - effective with small alphabets
  - different rules lead to different worst-case behavior

strong good suffix

can skip over this since we know dab doesn’t match
Boyer-Moore and Alphabet Size

Boyer-Moore space requirement. $O(M + A)$

Big alphabets.
- Direct implementation may be impractical, e.g., UNICODE.
- May explain why Java's `indexOf` doesn't use it.
- Solution 1: search one byte at a time.
- Solution 2: hash UNICODE characters to smaller range.

Small alphabets.
- Loses effectiveness when $A$ is too small, e.g., DNA.
- Solution: group characters together (aaaa, aaac, ...).

Tip of the Iceberg

Multiple string search. Search for any of $k$ different strings.
- Naive: $O(M + kN)$.
- Aho-Corasick: $O(M + N)$.
- Screen out dirty words from a text stream.

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

Approximate string matching: allow up to $k$ mismatches.
- Recovering from typing or spelling errors in information retrieval.
- Fixing transmission errors in signal processing.

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

Java String Library

Java String library has built-in string searching.
- `t.indexOf(p)`: index of 1st occurrence of pattern $p$ in text $t$.
- Caveat: it's brute force, and can take $O(MN)$ 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));
}
```

Why do you think library uses brute force?

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.
- Independent of alphabet size.
- Extends to multiple string search, wild-cards, regular expressions.

Boyer-Moore.
- Simple idea leads to dramatic speedup for long patterns.
- Need to tweak for small or large alphabets.