Strings

String.
- Sequence of characters over some alphabet.
  - binary \{0, 1\}
  - ASCII, UNICODE

Some applications.
- Word processors.
- Virus scanning.
- Text information retrieval systems. (Lexis, Nexis)
- Digital libraries.
- Natural language processing.
- Specialized databases.
- Computational molecular biology.
- Web search engines.

---

Strings

String.
- Sequence of characters over some alphabet.
  - binary \{0, 1\}
  - ASCII, UNICODE

Some applications.
- Word processors.
- Virus scanning.
- Text information retrieval systems. (Lexis, Nexis)
- Digital libraries.
- Natural language processing.
- Specialized databases.
- Computational molecular biology.
- Web search engines.

---

String Searching

Strings

String.
- Sequence of characters over some alphabet.
  - binary \{0, 1\}
  - ASCII, UNICODE

Some applications.
- Word processors.
- Virus scanning.
- Text information retrieval systems. (Lexis, Nexis)
- Digital libraries.
- Natural language processing.
- Specialized databases.
- Computational molecular biology.
- Web search engines.

---

String Searching

String.
- Sequence of characters over some alphabet.
  - binary \{0, 1\}
  - ASCII, UNICODE

Some applications.
- Word processors.
- Virus scanning.
- Text information retrieval systems. (Lexis, Nexis)
- Digital libraries.
- Natural language processing.
- Specialized databases.
- Computational molecular biology.
- Web search engines.

---

String Searching

String.
- Sequence of characters over some alphabet.
  - binary \{0, 1\}
  - ASCII, UNICODE

Some applications.
- Word processors.
- Virus scanning.
- Text information retrieval systems. (Lexis, Nexis)
- Digital libraries.
- Natural language processing.
- Specialized databases.
- Computational molecular biology.
- Web search engines.
Analysis of Brute Force

- Running time depends on pattern and text.
  - can be slow when strings repeat themselves
- Worst case: MN comparisons.
  - too slow when M and N are large

How To Save Comparisons

- How to avoid recomputation?
  - 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
  - Need better ideas in general.

Knuth-Morris-Pratt

- Use knowledge of how search pattern repeats itself.
- Build FSA from pattern.
- Run FSA on text.
- O(M + N) worst-case running time.
Knuth-Morris-Pratt

KMP algorithm.
- Use knowledge of how search pattern repeats itself.
- Build FSA from pattern.
- Run FSA on text.
- \( O(M + N) \) worst-case running time.
Knuth-Morris-Pratt

KMP algorithm.
- Use knowledge of how search pattern repeats itself.
- Build FSA from pattern.
- Run FSA on text.
- O(M + N) worst-case running time.

Search Pattern: a a b a a a
Search Text: a a a b a a b a a a b

Diagram:

Accept state

Search Pattern: a a b a a a
Search Text: a a a b a a b a a a b

Diagram:

Accept state

Knuth-Morris-Pratt

KMP algorithm.
- Use knowledge of how search pattern repeats itself.
- Build FSA from pattern.
- Run FSA on text.
- O(M + N) worst-case running time.

Search Pattern: a a b a a a
Search Text: a a a b a a b a a a b

Diagram:

Accept state

Search Pattern: a a b a a a
Search Text: a a a b a a b a a a b

Diagram:

Accept state
Knuth-Morris-Pratt

KMP algorithm.
- Use knowledge of how search pattern repeats itself.
- Build FSA from pattern.
- Run FSA on text.
- O(M + N) worst-case running time.

Search Pattern: a a b a a a
Search Text: a a a a b a b a a a a a b

Knuth-Morris-Pratt

KMP algorithm.
- Use knowledge of how search pattern repeats itself.
- Build FSA from pattern.
- Run FSA on text.
- O(M + N) worst-case running time.

Search Pattern: a a b a a a
Search Text: a a a a b a b a a a a a b

Knuth-Morris-Pratt

FSA Representation

FSA 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

Search Pattern: a a b a a a
next: 0 0 2 0 0 3

Knuth-Morris-Pratt

KMP algorithm.
- Use knowledge of how search pattern repeats itself.
- Build FSA from pattern.
- Run FSA on text.
- O(M + N) worst-case running time.

Search Pattern: a a b a a a
Search Text: a a a a b a b a a a a a b
KMP Algorithm

Given the FSA, string search is easy.
- The array next[] contains next FSA state if character mismatches.

### KMP String Search

```c
int kmpsearch(char p[], char t[], int next[]) {
    int i, j = 0;
    int M = strlen(p);  // pattern length
    int N = strlen(t);  // text length

    for (i = 0; i < N; i++) {
        if (t[i] == p[j]) j++;  // char match
        else j = next[j];       // char mismatch

        if (j == M) return i - M + 1;  // found
    }

    return -1;  // not found
}
```

FSA Construction for KMP

FSA construction for KMP.
- FSA builds itself!

Example. Building FSA for aabaaabb.
- State 6. p[0..5] = aabaa
  - assume you know state for p[1..5] = abaaa X = 2
  - if next char is b (match): go forward 6 + 1 = 7
  - if next char is a (mismatch): go to state for abaaa X + 'a' = 2
  - update X to state for p[1..6] = abaaab X + 'b' = 3

Example. Building FSA for aabaaabb.
- State 7. p[0..6] = aabaaab
  - assume you know state for p[1..6] = abaaab X = 3
  - if next char is b (match): go forward 7 + 1 = 8
  - next char is a (mismatch): go to state for abaaab X + 'a' = 4
  - update X to state for p[1..7] = abaaabb X + 'b' = 0
FSA Construction for KMP

FSA construction for KMP.
- FSA builds itself!

Example. Building FSA for "aabaaabb".

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

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

<table>
<thead>
<tr>
<th>Search Pattern</th>
<th>j</th>
<th>pattern[1..j]</th>
<th>X</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>a a b a a a b b</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>a</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

b

0 1

a

1 2

b 0 0

FSA Construction for KMP

<table>
<thead>
<tr>
<th>Search Pattern</th>
<th>j</th>
<th>pattern[1..j]</th>
<th>X</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>a a b a a a b b</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>a</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

b

0 1 2

a

1 2 2

b 0 0 3

FSA Construction for KMP

<table>
<thead>
<tr>
<th>Search Pattern</th>
<th>j</th>
<th>pattern[1..j]</th>
<th>X</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>a a b a a a b b</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>a</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

b

0 1 2 3

a

1 2 2 4

b 0 0 3 0

FSA Construction for KMP

<table>
<thead>
<tr>
<th>Search Pattern</th>
<th>j</th>
<th>pattern[1..j]</th>
<th>X</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>a a b a a a b b</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>a</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

b

0 1 2 3 4

a

1 2 2 4 5

b 0 0 3 0 0

FSA Construction for KMP

<table>
<thead>
<tr>
<th>Search Pattern</th>
<th>j</th>
<th>pattern[1..j]</th>
<th>X</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>a a b a a a b b</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>a</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

b

0 1 2 3

a

1 2 2 4 5

a 0 0 3 0 0

b

0 1 2 3 4

4 a a a a

b 2 0
FSA Construction for KMP

Code for FSA construction in KMP algorithm.

```c
void kmpinit(char p[], int next[]) {
    int j, X = 0, M = strlen(p);next[0] = 0;
    for (j = 1; j < M; j++) {
        if (p[X] == p[j]) {
            next[j] = next[X];
            X = X + 1;
        } else {
            next[j] = X + 1;X = next[X];
        }
    }
}
```
**Specialized KMP Implementation**

Specialized C program for \texttt{aabaaabb} pattern.

```c
int kmpsearch(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;
}
```

Ultimate search program for \texttt{aabaaabb} pattern.
- Machine language version of above.

**Summary of KMP**

KMP summary.
- Build FSA from pattern.
- Run FSA on text.
- O(M + N) worst case string search.
- Good efficiency for patterns and texts with much repetition.
  - binary files
  - graphics formats
- Less useful for text strings.
- On-line algorithm.
  - virus scanning
  - Internet spying

**History of KMP**

History of KMP.
- Inspired by theorem of Cook that says O(M + N) algorithm should be possible.
- Discovered in 1976 independently by two groups.
- Knuth-Pratt.
- Morris was hacker trying to build an editor.
  - annoying problem that you needed a buffer when performing text search

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

**Boyer-Moore**

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

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

\[
\begin{array}{c|c}
\text{Index} & g \ 4 \\
\hline
i & 2 \\
\hline
n & 3 \\
\hline
s & 0 \\
\hline
\gamma & 1 \\
\hline
r & -1
\end{array}
\]

\text{Text strings consisting of string}

\text{Pattern string}

Mismatch \quad \text{Match} \quad \text{No comparison}

---

Boyer-Moore

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

\text{Text}

\text{bad character heuristic}

---

Boyer-Moore

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

\text{Text}

\text{strong good suffix}

---

Boyer-Moore

Boyer-Moore analysis.
- \(O(N / M)\) average case if given letter usually doesn’t occur in string.
  - English text: 10 character search string, 26 char alphabet
  - time decreases as pattern length increases
  - sublinear in input size!
- \(O(M + N)\) worst-case with Galil variant.
  - proof is quite difficult
Karp-Rabin

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

Example.
- Hash "table" size = 97.

<table>
<thead>
<tr>
<th>Search Text</th>
<th>Search Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 1 4 1 5 9 2 6 5</td>
<td>5 9 2 6 5</td>
</tr>
</tbody>
</table>

59265 % 97 = 95

Karp-Rabin

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 = 10
- Previous hash: 41592 % 97 = 76
- Next hash: 15926 % 97

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

Karp-Rabin

Idea: use hashing.
- Compute hash function for each text position.

Problems.
- Need full compare on hash match to guard against collisions.
  - 59265 % 97 = 95
  - 59362 % 97 = 95

- Hash function depends on M characters.
  - running time on search miss = MN

Karp-Rabin (Sedgewick, p. 290)

```c
#include <math.h>

#define q 3355439 // table size
#define d 256 // radix

int rksearch(char p[], char t[]) {
    int i, j, dM = 1, h1 = 0, h2 = 0;
    int M = strlen(p), N = strlen(t);
    for (j = 1; j < M; j++)              // precompute d^M % q
          dM = (d * dM) % q;
    for (j = 0; j < M; j++) {
        h1 = (h1*d + p[j]) % q;           // hash of pattern
        h2 = (h2*d + t[j]) % q;           // hash of text
    }
    for (i = M; i < N; i++) {
        if (h1 == h2) return i - M;       // match found
        h2 = (h2 - a[i-M]*dM) % q;       // remove high order digit
          h2 = (h2*d + a[i]) % q;        // insert low order digit
    }
    return -1;                           // not found
}
```
Karp-Rabin

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

Randomized algorithms.
- Monte Carlo: don't check for collisions.
  - algorithm can be wrong but running time guaranteed linear
- Las Vegas: if collision, start over with new random table size.
  - algorithm always correct, but running time is expected linear

Advantages.
- Extends to 2d patterns and other generalizations.