5.3 Substring Search

- brute force
- Knuth-Morris-Pratt
- Boyer-Moore
- Rabin-Karp

Substring search

Goal. Find pattern of length $M$ in a text of length $N$.

Typically $N \gg M$

Applications

- Parsers
- Spam filters
- Digital libraries
- Screen scrapers
- Word processors
- Web search engines
- Electronic surveillance
- Natural language processing
- Computational molecular biology
- FBI's Digital Collection System 3000
- Feature detection in digitized images
- ...

Application: spam filtering

Identify patterns indicative of spam.

- PROFITS
- LOSE WEIGHT
- herbal Viagra
- There is no catch.
- LOW MORTGAGE RATES
- This is a one-time mailing.
- This message is sent in compliance with spam regulations.
Application: electronic surveillance

Need to monitor all internet traffic. (security)

No way. (privacy)

Well, we're mainly interested in "ATTACK AT DAWN"

OK. Build a machine that just looks for that.

"ATTACK AT DAWN" substring search machine found

Application: screen scraping

Goal. Extract relevant data from web page.

Ex. Find string delimited by <b> and </b> after first occurrence of pattern Last Trade:

```
<table>
<thead>
<tr>
<th>Company</th>
<th>Last Trade</th>
<th>Volume</th>
</tr>
</thead>
<tbody>
<tr>
<td>Google (GOOG)</td>
<td>564.35</td>
<td>4.20M</td>
</tr>
<tr>
<td>Microsoft (MSFT)</td>
<td>26.04</td>
<td>2.10M</td>
</tr>
</tbody>
</table>
```

Screen scraping: Java implementation

Java library. The indexOf() method in Java's string library returns the index of the first occurrence of a given string, starting at a given offset.

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

```
% java StockQuote goog
564.35
% java StockQuote msft
26.04
```
Brute-force substring search

Check for pattern starting at each text position.

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>i+j</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>10</td>
</tr>
</tbody>
</table>

**txt** is A A A A A A A A B

**pat** is A B R A

Entries in red are mismatches.

Entries in gray are for reference only.

Return 5 when j is M

Brute-force substring search: worst case

Brute-force algorithm can be slow if text and pattern are repetitive.

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>i+j</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>10</td>
</tr>
</tbody>
</table>

**txt** is A A A A A A A A B

**pat** is A B R A

Worst case. ～MN char compares.

Backup

In typical applications, we want to avoid backup in text stream.

- Treat input as stream of data.
- Abstract model: standard input.

Brute-force algorithm needs backup for every mismatch.

**txt** is A A A A A A A A A

**pat** is A A A A A A A A B

Backup

Approach 1. Maintain buffer of size M (build backup into standard input).

Approach 2. Stay tuned.
Brute-force substring search: alternate implementation

Same sequence of char compares as previous implementation.
• i points to end of sequence of already-matched chars in text.
• j stores number of already-matched chars (end of sequence in pattern).

```
public static int search(String pat, String txt) {
    int i, N = txt.length();
    int j, M = pat.length();
    for (i = 0, j = 0; i < N && j < M; i++) {
        if (txt.charAt(i) == pat.charAt(j)) j++;
        else { i -= j; j = 0; }
    }
    if (j == M) return i - M;
    else            return N;
}
```

Algorithmic challenges in substring search

Brute-force is often not good enough.

Theoretical challenge. Linear-time guarantee. ← fundamental algorithmic problem

Practical challenge. Avoid backup in text stream. ← often no room or time to save text

Now is the time for all people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for many good people to come to the aid of their party. Now is the time for all good Republicans to come to the aid of their party. Now is the time for all good Democrats to come to the aid of their party. Now is the time for all people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for all of the good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for all of the good people to come to the aid of their party. Now is the time for all good Democrats to come to the aid of their party. Now is the time for all good Republicans to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for many or all good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for all of the good people to come to the aid of their party. Now is the time for all good Democrats to come to the aid of their party.

Knuth-Morris-Pratt substring search

Intuition. Suppose we are searching in text for pattern BAAAAAAAA.
• Suppose we match 5 chars in pattern, with mismatch on 6th char.
• We know previous 6 chars in text are BAAAA.
• Don’t need to back up text pointer!

Knuth-Morris-Pratt algorithm. Clever method to always avoid backup. (!)

Text pointer backup in substring searching

```
public class Text {
    public void setCharAt(int i, char c) {
        text[i] = c;
    }
    public char getCharAt(int i) {
        return text[i];
    }
}
```
Deterministic finite state automaton (DFA)

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.

**DFA corresponding to the string A B A B A C**

**graphical representation**

```
<table>
<thead>
<tr>
<th>pat.charAt(j)</th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>dfa[j][i]</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>
```

**internal representation**

```
<table>
<thead>
<tr>
<th>j</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>pat.charAt(j)</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>dfa[j][i]</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>
```

**KMP substring search: trace**

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

read this char B C B A A B A C A A B A B A C A A
in this state 0 0 0 1 1 2 3 4 5 6
go to state A B A B A C A B A B A C A B A B A C
match
set j to dfa[pat.charAt(5)][j] = dfa[pat.charAt(4)][j] = j+1
set j to dfa[pat.charAt(3)][j] implies pattern shift to align pat.charAt(j) with txt.charAt(i-2)
match
next j = dfa[pat.charAt(4)][j] = dfa[pat.charAt(3)][j] = dfa[pat.charAt(2)][j] = j+1
```

**Trace of KMP substring search (DFA simulation) for A B A B A C**

```
0 1 2 3 4 5
pat.charAt(j) A B A B A C
dfa[j][] 0 1 3 1 5 1
```

**Interpretation of Knuth-Morris-Pratt DFA**

**Q.** What is interpretation of DFA state after reading in $\text{txt}[i]$?
**A.** State is number of characters in pattern that have been matched. (length of longest prefix of $\text{pat}[i]$ that is a suffix of $\text{txt}[0..i]$)

**Ex.** DFA is in state 3 after reading in character $\text{txt}[6]$.

```
<table>
<thead>
<tr>
<th>txt</th>
<th>B</th>
<th>C</th>
<th>B</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>pat</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</td>
<td></td>
</tr>
</tbody>
</table>
```

**KMP search: Java implementation**

```
public int search(String txt) {
    int i, j, N = txt.length();
    for (i = 0, j = 0; i < N && j < M; i++)
        j = dfa[txt.charAt(i)][j];
    if (j == M) return i - M;
    else        return N;
}
```

**Key differences from brute-force implementation.**
- Text pointer $i$ never decrements.
- Need to precompute $dfa[][]$ from pattern.

**Running time.**
- Simulate DFA on text: at most $N$ character accesses.
- Build DFA: how to do efficiently? [warning: tricky algorithm ahead]
KMP search: Java implementation

Knuth-Morris-Pratt construction

Include one state for each character in pattern (plus accept state).

Match transition. If in state \( j \) and next char \( c = \text{pat.charAt}(j) \), then go to state \( j+1 \).

Match transition. For each state \( j \), \( \text{dfa[pat.charAt(j)]}[j] = j+1 \).

Building DFA from pattern: easy case

Knuth-Morris-Pratt construction

Match transition. If in state \( j \) and next char \( c = \text{pat.charAt}(j) \), then go to state \( j+1 \).

Match transition. For each state \( j \), \( \text{dfa[pat.charAt(j)]}[j] = j+1 \).

Knuth-Morris-Pratt construction

Match transition. If in state \( j \) and next char \( c = \text{pat.charAt}(j) \), then go to state \( j+1 \).

Match transition. For each state \( j \), \( \text{dfa[pat.charAt(j)]}[j] = j+1 \).
Building DFA from pattern: tricky case

Mismatch transition. Suppose DFA is in state $j$ and next char is $c \neq \text{pat.charAt}(j)$

- $\ldots A B A B A C \ldots$ match
- $A B A B A A$ mismatch case 1
- $A B A B A B$ mismatch case 2

Brute-force solution: Back up $j-1$ chars and restart.

Key idea 1. The last $j$ characters of input are known to be $\text{pat[1..j-1]}$, so use the DFA itself to find what state ($X$) it would be in if restarted after backup.

Key idea 2. Maintain the value of $X$ while constructing DFA.

Ex.

$\text{pat[1..j-1]} = B A B A$

$X = 3$

\[
\begin{align*}
\text{dfa[\text{A}]()[5]} & = \text{dfa[\text{A}][X]} = 1 \\
\text{dfa[\text{B}]()[5]} & = \text{dfa[\text{B}][X]} = 4 \\
\text{dfa[\text{C}]()[5]} & = 6 \\
\text{new } X & = \text{dfa[\text{C}][X]} = 0
\end{align*}
\]

Knuth-Morris-Pratt construction

Mismatch transition. For each state $j$ and char $c \neq \text{pat.charAt}(j)$,

$\text{dfa}[c][j] = \text{dfa}[c][X]$: then update $X = \text{dfa[\text{pat.charAt}(j)][X]}$.

Constructing the DFA for KMP substring search for $A B A B A C$
**Knuth-Morris-Pratt construction**

**Mismatch transition.** For each state \( j \) and char \( c \) != \( \text{pat.charAt}(j) \), \( \text{dfa}[c][j] = \text{dfa}[c][X] \); then update \( X = \text{dfa}[	ext{pat.charAt}(j)][X] \).

![Diagram](image1)

**Constructing the DFA for KMP substring search for A B A B A C**

**Mismatch transition.** For each state \( j \) and char \( c \) != \( \text{pat.charAt}(j) \), \( \text{dfa}[c][j] = \text{dfa}[c][X] \); then update \( X = \text{dfa}[	ext{pat.charAt}(j)][X] \).

![Diagram](image2)

**Knuth-Morris-Pratt construction**

**Mismatch transition.** For each state \( j \) and char \( c \) != \( \text{pat.charAt}(j) \), \( \text{dfa}[c][j] = \text{dfa}[c][X] \); then update \( X = \text{dfa}[	ext{pat.charAt}(j)][X] \).

![Diagram](image3)

**Running time.** \( M \) character accesses (but space proportional to \( RM \)).
**KMP substring search analysis**

**Proposition.** KMP substring search accesses no more than $M + N$ chars to search for a pattern of length $M$ in a text of length $N$.

**Pf.** Each pattern char accessed once when constructing the DFA; each text char accessed once (in the worst case) when simulating the DFA.

**Proposition.** KMP constructs $\text{dfa}[]$ in time and space proportional to $RM$.

**Larger alphabets.** Improved version of KMP constructs $\text{nfa}[]$ in time and space proportional to $M$.

---

**Knuth-Morris-Pratt application**

A string $s$ is a cyclic rotation of $t$ if $s$ and $t$ have the same length and $s$ is a suffix of $t$ followed by a prefix of $t$.

<table>
<thead>
<tr>
<th>yes</th>
<th>ROTATED STRING</th>
<th>yes</th>
<th>ABABABABABA</th>
<th>no</th>
<th>ROTATED STRING</th>
<th>GNIRTSDETATOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROTATED STRING</td>
<td>ABABABABABA</td>
<td>no</td>
<td>STRING</td>
<td>GNIRTSDETATOR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Problem.** Given two strings $s$ and $t$, design a linear-time algorithm that determines if $s$ is a cyclic rotation of $t$.

**Solution.**
- Check that $s$ and $t$ are the same length.
- Search for $s$ in $t + t$ using KMP.

---

**Knuth-Morris-Pratt: brief history**

- Independently discovered by two theoreticians and a hacker.
  - Knuth: inspired by esoteric theorem, discovered linear-time algorithm
  - Pratt: made running time independent of alphabet size
  - Morris: built a text editor for the CDC 6400 computer
- Theory meets practice.

---

**Knuth-Morris-Pratt application**

A string $s$ is a cyclic rotation of $t$ if $s$ and $t$ have the same length and $s$ is a suffix of $t$ followed by a prefix of $t$.

<table>
<thead>
<tr>
<th>yes</th>
<th>ROTATED STRING</th>
<th>yes</th>
<th>ABABABABABA</th>
<th>no</th>
<th>ROTATED STRING</th>
<th>GNIRTSDETATOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROTATED STRING</td>
<td>ABABABABABA</td>
<td>no</td>
<td>STRING</td>
<td>GNIRTSDETATOR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Problem.** Given two strings $s$ and $t$, design a linear-time algorithm that determines if $s$ is a cyclic rotation of $t$.

**Solution.**
- Check that $s$ and $t$ are the same length.
- Search for $s$ in $t + t$ using KMP.
**Intuition.**

- Scan characters in pattern from right to left.
- Can skip \( M \) text chars when finding one not in the pattern.

---

**Mismatched character heuristic**

**Q. How much to skip?**

**A.** Compute \( \text{right}[c] \) = rightmost occurrence of character \( c \) in \( \text{pat} \).

```java
right = new int[R];
for (int c = 0; c < R; c++)
    right[c] = -1;
for (int j = 0; j < M; j++)
    right[\text{pat.charAt(j)}] = j;
```

---

**Mismatched character heuristic**

Character not in pattern? Set \( \text{right}[c] \) to \(-1\).
**Boyer-Moore: mismatched character heuristic**

**Q.** How much to skip?

**A.** Compute \( \text{right}[c] \) = rightmost occurrence of character \( c \) in \( \text{pat} \).

Heuristic no help? Increment \( i \) and reset \( j \) to \( M-1 \).

**Boyer-Moore: Java implementation**

```java
public int search(String txt) {
    int N = txt.length();
    int M = pat.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;
    }
    return N;
}
```

**Boyer-Moore: analysis**

**Property.** Substring search with the Boyer-Moore mismatched character heuristic takes about \( \sim N/M \) character compares to search for a pattern of length \( M \) in a text of length \( N \).

**Worst-case.** Can be as bad as \( \sim MN \).

**Boyer-Moore variant.** Can improve worst case to \( \sim 3N \) by adding a KMP-like rule to guard against repetitive patterns.

---

Michael Rabin, Turing Award '76
and Dick Karp, Turing Award '85
Rabin-Karp fingerprint search

Basic idea = 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.

Modular hash function. Using the notation \( \text{keyCharAt}(i) \), 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.

Rabin-Karp substring search example

Efficiently computing the hash function

Challenge. How to efficiently compute \( x_{i+1} \) given that we know \( x_i \),
\[
x_i = t_i R^{M-1} + t_{i+1} R^{M-2} + \ldots + t_{i+M-1} R^0
\]
\[
x_{i+1} = t_{i+1} R^{M-1} + t_{i+2} R^{M-2} + \ldots + t_{i+M} R^0
\]

Key property. Can update hash function in constant time!
\[
x_{i+1} = x_i R - t_i R^M + t_{i+M}
\]

// Compute hash for M-digit key
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;
}
Rabin-Karp: Java implementation

```java
class RabinKarp {
    private long patHash; // pattern hash value
    private int M;       // pattern length
    private long Q;      // modulus
    private int R;       // radix
    private long RM;     // R^(M-1) % Q

    public RabinKarp(String pat) {
        M = pat.length();
        R = 256;
        Q = longRandomPrime();
        RM = 1;
        for (int i = 1; i <= M-1; i++)
            RM = (R * RM) % Q;
        patHash = hash(pat, M);
    }

    private long hash(String key, int M) {
        // as before
    }

    public int search(String txt) {
        int N = txt.length();
        int txtHash = hash(txt, M);
        if (patHash == txtHash) return 0;
        for (int i = M; i < N; i++)
            txtHash = (txtHash + Q - RM*txt.charAt(i-M) % Q) % Q;
        txtHash = (txtHash*R + txt.charAt(i)) % Q;
        if (patHash == txtHash) return i - M + 1;
        return N;
    }
}
```

Monte Carlo version. Return match if hash match.

```java
public int search(String txt) {
    int N = txt.length();
    int txtHash = hash(txt, M);
    if (patHash == txtHash) return 0;
    for (int i = M; i < N; i++)
        txtHash = (txtHash + Q - RM*txt.charAt(i-M) % Q) % Q;
    txtHash = (txtHash*R + txt.charAt(i)) % Q;
    if (patHash == txtHash) return i - M + 1;
    return N;
}
```

Las Vegas version. Check for substring match if hash match; continue search if false collision.

Rabin-Karp analysis

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 as 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 \( M N \)).

Rabin-Karp fingerprint search

Advantages.
- Extends to 2d patterns.
- Extends to finding multiple patterns.

Disadvantages.
- Arithmetic ops slower than char compares.
- Poor worst-case guarantee.
- Requires backup.

Q. How would you extend Rabin-Karp to efficiently search for any one of \( P \) possible patterns in a text of length \( N \)?
Substring search cost summary

Cost of searching for an $M$-character pattern in an $N$-character text.

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

† probabilistic guarantee, with uniform hash function