5.3 Substring Search

- introduction
- brute force
- Knuth–Morris–Pratt
- Boyer–Moore
5.3 Substring Search

- introduction
- brute force
- Knuth–Morris–Pratt
- Boyer–Moore
Substring search

Goal. Find pattern of length $m$ in a text of length $n$.

typically $n \gg m$

\[\text{pattern} \rightarrow \text{NEEDLE} \]
\[\text{text} \rightarrow \text{INAHAYSTACK NEEDLE INA} \]

\text{match}
Substring search applications

Goal. Find pattern of length $m$ in a text of length $n$.

Typically $n \gg m$

Search in a word processor or IDE.
Substring search applications

Goal. Find pattern of length $m$ in a text of length $n$.

Typically $n \gg m$

**Pattern** → NEEDLE

**Text** → INAHAYSTACK NEEDLE INA

match

Computer forensics.
Search memory or disk for signatures,
e.g., all URLs or RSA keys that the user has entered.
5.3 Substring Search

- introduction
- brute force
- Knuth–Morris–Pratt
- Boyer–Moore
Brute-force substring search

Check for pattern starting at each text position.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</td>
<td>A</td>
<td>D</td>
<td>A</td>
<td>B</td>
<td>R</td>
<td>A</td>
<td>C</td>
</tr>
</tbody>
</table>

Entries in red are mismatches.
Entries in gray are for reference only.
Entries in black match the text.

Match
Brute-force substring search: Java implementation

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>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

A B A C A D A B R A C

Brute-force substring search: Java implementation

```java
public static int search(String pat, String txt) {
    int m = pat.length();
    int n = txt.length();
    for (int i = 0; i <= n - m; i++) {
        int j; // number of characters that match
        for (j = 0; j < m; j++)
            if (txt.charAt(i+j) != pat.charAt(j))
                break;
        if (j == m) return i; // index in text where pattern starts
    }
    return n; // not found
}
```
What is the worst-case running time of brute-force substring search as a function of the pattern length $m$ and text length $n$?

A. $m + n$
B. $m^2$
C. $mn$
D. $n^2$
5.3 Substring Search

- introduction
- brute force
- Knuth–Morris–Pratt
- Boyer–Moore
Knuth–Morris–Pratt substring search

**Intuition.** Suppose we are searching in text for pattern `BAAAAAAAAAAAA`. 
- Suppose we match 5 chars in pattern, with mismatch on 6\textsuperscript{th} char.
Knuth–Morris–Pratt substring search

**Intuition.** Suppose we are searching in text for pattern B A A A A A A A A A A A A A A A A A A A A A A A A A A A A A. 
- Suppose we match 5 chars in pattern, with mismatch on 6\textsuperscript{th} char. 
- We know previous 6 chars in text must be B A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A. 
- Don’t need to compare any text character twice.

**Knuth–Morris–Pratt algorithm.** Clever method to always avoid comparing a text character more than once!
Deterministic finite state automaton (DFA)

A DFA is an abstract string-searching machine.

- Finite number of states (including start and halt).
- Exactly one state transition for each char in alphabet.
- Accept if sequence of state transitions leads to halt state.

**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]</td>
<td>A</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

If in state j reading char C:
  - if j is 6 halt and accept
  - else move to state dfa[c][j]

**graphical representation**
Knuth–Morris–Pratt demo: DFA simulation

A A B A C A A B A B A C A A A

pat.charAt(j) | 0 1 2 3 4 5
---|---
A | B A B A A C
A | 1 1 3 1 5 1
B | 0 2 0 4 0 4
C | 0 0 0 0 0 6

dfa[][] | A B C
---|---
A | A B C
B | B A C
C | C A B

Path: [j]
Interpretation of Knuth–Morris–Pratt DFA

Q. What is interpretation of DFA state after reading in $\text{txt}[i]$?

A. State = number of characters in pattern that have been matched.

Ex. DFA is in state 3 after reading in $\text{txt}[0..6]$.

<table>
<thead>
<tr>
<th>txt</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>B</td>
<td>C</td>
<td>B</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</td>
<td>A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pat</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</td>
</tr>
</tbody>
</table>

prefix of $\text{pat}[]$

length of longest prefix of $\text{pat}[]$ that is a suffix of $\text{txt}[0..i]$
Which state is the DFA in after processing the following input?

B A A B A B A B

A. 0
B. 1
C. 3
D. 4
Which state is the DFA in after processing the following input?


A. 0
B. 1
C. 3
D. 4
E. 5
Knuth–Morris–Pratt substring search: Java implementation

Key differences from brute-force implementation.
- Need to precompute $\text{dfa}[][]$ from pattern.
- Each text character compared (at most) once.

```java
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;
}
```

Running time.
- Simulate DFA on text: at most $n$ character accesses.
- Build DFA: how to do efficiently? [tricky algorithm ahead]
Constructing the DFA for KMP substring search for A B A B A C
How to build DFA from pattern?

Include one state for each character in pattern (plus accept state).
How to build DFA from pattern?

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

- First \( j \) characters of pattern have already been matched.
- Next char matches.
- Now first \( j+1 \) characters of pattern have been matched.

<table>
<thead>
<tr>
<th>( \text{pat.charAt}(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>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

DFA:

- State 0: A → 1 (A)
- State 1: B → 2 (AB)
- State 2: A → 3 (ABA)
- State 3: B → 4 (ABAB)
- State 4: A → 5 (ABABA)
- State 5: C → 6 (ABABAC)
How to build DFA from pattern?

Mismatch transition. If in state \( j \) and next char \( c \) != \( \text{pat.charAt}(j) \), then the last \( j-1 \) characters of input are \( \text{pat}[1..j-1] \), followed by \( c \).

To compute \( \text{dfa}[c][j] \): Simulate \( \text{pat}[1..j-1] \) on DFA and take transition \( c \).

Ex. \( \text{dfa}[\text{'A'}][5] = 1 \)  \( \text{dfa}[\text{'B'}][5] = 4 \)

<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>( \text{pat.charAt}(j) )</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</td>
</tr>
</tbody>
</table>

Simulation of BABAB
How to build DFA from pattern?

Mismatch transition. If in state \( j \) and next char \( c \neq \text{pat.charAt}(j) \), then the last \( j-1 \) characters of input are \( \text{pat}[1..j-1] \), followed by \( c \).

To compute \( \text{dfa}[c][j] \): Simulate \( \text{pat}[1..j-1] \) on DFA and take transition \( c \).

**Ex.** \( \text{dfa}[\text{'A'}][5] = 1 \)
from state \( x \),
take transition 'A'
\( = \text{dfa}[\text{'A'}][x] \)

\( \text{dfa}[\text{'B'}][5] = 4 \)
from state \( x \),
take transition 'B'
\( = \text{dfa}[\text{'B'}][x] \)

\( x' = 0 \)
from state \( x \),
take transition 'C'
\( = \text{dfa}[\text{'C'}][x] \)
Knuth–Morris–Pratt demo: DFA construction in linear time

Linear in the size of the DFA, which is $R_m$.

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

```
pat.charAt(j)  0  1  2  3  4  5
A   B   A   B   A   C
A   1   1   3   1   5   1
B   0   2   0   4   0   4
C   0   0   0   0   0   6
```
Constructing the DFA for KMP substring search: Java implementation

For each state $j$:

- Copy $\text{dfa}[][x]$ to $\text{dfa}[][j]$ for mismatch case.
- Set $\text{dfa}[\text{pat.charAt}(j)][j]$ to $j+1$ for match case.
- Update $x$.

```java
public KMP(String pat) {
    this.pat = pat;
    m = pat.length();
    dfa = new int[R][m];
    dfa[pat.charAt(0)][0] = 1;
    for (int x = 0, j = 1; j < m; j++)
    {
        for (int c = 0; c < R; c++)
            dfa[c][j] = dfa[c][x];
        dfa[pat.charAt(j)][j] = j+1;
        x = dfa[pat.charAt(j)][x];
    }
}
```

Running time. $m$ character accesses (but space/time proportional to $R \cdot m$).
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 character accessed once when constructing the DFA; each text character accessed (at most) once 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 $m$. 

[Diagram of KMP NFA for ABABAC]
5.3 Substring Search

- introduction
- brute force
- Knuth–Morris–Pratt
- Boyer–Moore
Intuition.

- Scan characters in pattern from right to left.
- Can skip as many as $m$ text chars when finding one not in the pattern.
Boyer–Moore: mismatched character heuristic

Q. How much to skip?

Case 1. Mismatch character not in pattern.

Mismatch character T not in pattern: increment i one character beyond T
Boyer–Moore: mismatched character heuristic

Q. How much to skip?

Case 2a. Mismatch character in pattern.

Mismatch character N in pattern: align text N with rightmost (why?) pattern N
Boyer–Moore: mismatched character heuristic

Q. How much to skip?

Case 2b. Mismatch character in pattern (but heuristic no help).

Before

<table>
<thead>
<tr>
<th>before</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>txt</td>
<td>. . . . . . E L E . . . . . .</td>
</tr>
<tr>
<td>pat</td>
<td>N E E D L E</td>
</tr>
</tbody>
</table>

Aligned with rightmost E?

<table>
<thead>
<tr>
<th>aligned with rightmost E?</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>txt</td>
<td>. . . . . . E L E . . . . . .</td>
</tr>
<tr>
<td>pat</td>
<td>N E E D L E</td>
</tr>
</tbody>
</table>

Mismatch character E in pattern: align text E with rightmost pattern E?
Boyer–Moore: mismatched character heuristic

Q. How much to skip?

Case 2b. Mismatch character in pattern (but heuristic no help).

Mismatch character E in pattern: increment i by 1
Which text character is compared with the E next in Boyer–Moore?

A. R (index 5)
B. O (index 6)
C. O (index 12)
D. O (index 13)
Which text character is compared with the E next in Boyer–Moore?

A. O  
B. R  
C. E  
D. J

Substring search: quiz 6

text → B O O Y E R O B E R T M O O R E J S

pattern → M O O R E
Boyer–Moore:  mismatched character heuristic

Q.  How much to skip?

A.  Precompute index of rightmost occurrence of character \( c \) in pattern. 
    
    \((-1 \text{ if character not in pattern})\)

```java
right = new int[R];
for (int c = 0; c < R; c++)
    right[c] = -1;
for (int j = 0; j < m; j++)
    right[pat.charAt(j)] = j;
```

<table>
<thead>
<tr>
<th>( c )</th>
<th>N</th>
<th>E</th>
<th>E</th>
<th>D</th>
<th>L</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>B</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>C</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>D</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>-1</td>
<td>1</td>
<td>2</td>
<td></td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>M</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>N</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Boyer-Moore skip table computation
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 $\sim 3n$ character compares by adding a KMP-like rule to guard against repetitive patterns.
Which substring search algorithm does Java’s `indexOf()` method use?

A. Brute-force search
B. Knuth–Morris–Pratt
C. Boyer–Moore
D. None of the above
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>Extra Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>brute force</td>
<td>—</td>
<td>$MN$</td>
<td>$1$</td>
</tr>
<tr>
<td>Knuth-Morris-Pratt</td>
<td>full DFA (Algorithm 5.6)</td>
<td>$2N$</td>
<td>$MR$</td>
</tr>
<tr>
<td></td>
<td>mismatch transitions only</td>
<td>$3N$</td>
<td>$M$</td>
</tr>
<tr>
<td>Boyer-Moore</td>
<td>full algorithm</td>
<td>$3N$</td>
<td>$R$</td>
</tr>
<tr>
<td></td>
<td>mismatched char heuristic</td>
<td>$MN$</td>
<td>$R$</td>
</tr>
<tr>
<td></td>
<td>only (Algorithm 5.7)</td>
<td>$N / M$</td>
<td></td>
</tr>
</tbody>
</table>

*† probabilistic guarantee, with uniform hash function*