6.3 Substring Search

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: 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

"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:

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
256.44

% java StockQuote msft
19.68

brute force

Knuth-Morris-Pratt

Boyer-Moore

Rabin-Karp
### 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>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>10</td>
</tr>
</tbody>
</table>

entries in black match the text

entries in red are mismatches

entries in gray are for reference only

\[
\begin{align*}
  & A \quad B \quad C \\
  & D \quad A \quad B \quad R \quad A \\
  & A \quad B \quad R \quad A \\
  & A \quad B \quad R \quad A \\
  & A \quad B \quad R \quad A
\end{align*}
\]

```
public static int search(char[] pat, char[] txt)
{
  int M = pat.length;
  int N = txt.length;
  for (int i = 0; i < N - M; i++)
  {
    int j;
    for (j = 0; j < M; j++)
      if (txt[i+j] != pat[j])
        break;
    if (j == M) return i;
  }
  return N;
}
```

Worst case. \( \sim MN \) char compares.

### Brute-force substring search: Java implementation

Check for pattern starting at each text position.

```
public static int search(char[] pat, char[] txt){
  int M = pat.length;
  int N = txt.length;
  for (int i = 0; i < N - M; i++)
  {
    int j;
    for (j = 0; j < M; j++)
      if (txt[i+j] != pat[j])
        break;
    if (j == M) return i;
  }
  return N;  // not found
}
```

### 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>4</td>
<td>9</td>
</tr>
</tbody>
</table>

Brute-force substring search (worst case)
```

### Backup

In typical applications, we want to avoid backup in text stream.
- treat input as stream of data
- abstract model: `StdIn`

Backup

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

- treat input as stream of data
- abstract model: `StdIn`

Brute-force algorithm needs backup for every mismatch

```
  A \quad A \quad A \quad A \quad A \quad A \quad A \quad A \quad A \quad A
  A \quad A \quad A \\
  A \quad A \\
  A \quad A \\
  A \\
  A 
```

```
  A \quad A \quad A \\
  A \quad A \\
  A \\
  A
```

Approach 1: Maintain buffer of size \( M \) (build backup into `StdIn`)
Other approaches: 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).

```java
public static int search(char[] pat, char[] txt) {
    int j, M = pat.length;
    int i, N = txt.length;
    for (i = 0, j = 0; i < N && j < M; i++) {
        if (txt[i] == pat[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.

Practical challenge. Avoid backup in text stream.

Knuth-Morris-Pratt substring search

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

Remark. It is always possible to avoid backup.
Q. What pattern char do we compare to the next text char on **match**?
A. Easy: the next one.

Q. Which pattern char should we compare with the next text char?
A. Check each position, working from left to right.

Ex. Build table for **aaaac**.

<table>
<thead>
<tr>
<th>j</th>
<th>pat[j]</th>
<th>dfa[c][j]</th>
<th>state (pattern index)</th>
<th>text (pattern index)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>AAAAC</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td>A</td>
<td>ABABAC</td>
<td>ABABAC</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>ABABAC</td>
<td>ABABAC</td>
<td>ABABAC</td>
</tr>
<tr>
<td>3</td>
<td>B</td>
<td>A</td>
<td>ABABAC</td>
<td>ABABAC</td>
</tr>
<tr>
<td>4</td>
<td>C</td>
<td>A</td>
<td>ABABAC</td>
<td>ABABAC</td>
</tr>
</tbody>
</table>

Total cost: O(MR) char compares (stay tuned for a better method).
Deterministic finite state automaton (DFA)

DFA is abstract string-searching machine.
• Finite number of states (including start and halt).
• Exactly one transition for each input symbol.
• Accept if sequence of transitions leads to halt state.

Knuth-Morris Pratt algorithm: Build machine for pattern, simulate it on text.

KMP search: Java implementation

Key differences from brute-force implementation.
• Text pointer i never decrements.
• Need to precompute \( dfa[i][j] \) table from pattern.

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

KMP substring search: trace

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

---

DFA corresponding to the string \( A B A B A C \)

If in state \( j \) reading char \( c \):
- halt if \( j \) is \( 6 \)
- else move to state \( dfa[c][j] \)

mismatch:
- set \( j \) to \( dfa[txt[i]][j] \)
  implies pattern shift to align \( pat[j] \) with \( txt[i+1] \)

match:
- set \( j \) to \( dfa[txt[i]][j] = j+1 \)
Efficiently constructing the DFA for KMP substring search

Q1. What state X would the DFA be in if it were restarted to correspond to shifting the pattern one position to the right?

A1. Use the (partially constructed) DFA to find X!

A. A A A B B B B C C C
B. A B A B A
C. 0 1 2 3

Q2. Why is that relevant?

A2. We want the same transitions for the next state on mismatch

X = dfa[pat[j]][X];

A2. (continued), and a different transition (to j+1) on match

dfa[pat[j]][j] = j+1

Constructing the DFA for KMP substring search: example

DFA simulations to compute restart states for A, B, A, B, A, C

Constructing the DFA for KMP substring search

Q1. What state X would the DFA be in if it were restarted to correspond to shifting the pattern one position to the right?

A1. Use the (partially constructed) DFA to find X!

A. A A A B B B B C C C
B. A B A B A
C. 0 1 2 3

Q2. Why is that relevant?

A2. We want the same transitions for the next state on mismatch

X = dfa[pat[j]][X];

A2. (continued), and a different transition (to j+1) on match

dfa[pat[j]][j] = j+1

Important note:

- no need to restart DFA
- remember last restart state in X
- use DFA to update X

X = dfa[pat[j]][X]
Constructing the DFA for KMP substring search: example

For each $j$:
- Copy $dfa[X][j]$ to $dfa[j][j]$ for mismatch case.
- Set $dfa[pat[j]][j]$ to $j+1$ for match case.
- Update $X$.

```java
public KMP(int R, char[] pat) {
    this.pat = pat;
    M = pat.length;
    dfa = new int[R][M];
    dfa[pat[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[j]][j] = j+1;
        X = dfa[pat[j]][X];
    }
}
```

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.** We access each pattern char once when construction DFA, and we access each text char once (in the worst case) when simulating the DFA on given text.

**Remark.** Takes time and space proportional to $RM$ to build $dfa[] []$, but with cleverness, can reduce time and space to $M$.

Knuth-Morris-Pratt: brief history

**Brief history.**
- Inspired by esoteric theorem of Cook.
- 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 a text editor
- Theory meets practice.

Stephen Cook  Don Knuth  Jim Morris  Vaughan Pratt
Boyer-Moore: mismatched character heuristic

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

Boyer-Moore: Java implementation

```java
public int search(char[] 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[j] != txt[i+j])
                skip = Math.max(1, j - right[txt[i+j]]);
        if (skip == 0) return i;
    }
    return N;
}
```

Q. How much to skip?
A. Compute $right[c] =$ rightmost occurrence of character $c$ in $pat[]$.

```java
int[] right = new int[R];
for (int c = 0; c < R; c++)
    right[c] = -1;
for (int j = 0; j < M; j++)
    right[pat[j]] = j;
```
Boyer-Moore: analysis

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

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

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

- Used in Unix, emacs.

Rabin-Karp fingerprint search

**Basic idea.**
- Compute a hash of \( \text{pat}[0..M) \).
- Compute a hash of \( \text{txt}[i..M+i) \) for each \( i \).
- If pattern hash = text substring hash, check for a match.

Modular hash function. Using the notation \( t_i \) for \( \text{txt}[i] \), 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.

```java
// Compute hash for key[0..M-1]
private int hash(char[] key, int M) {
    int h = 0;
    for (int j = 0; j < M; j++)
        h = (R * h + key[j]) % Q;
    return h;
}
```
Efficiently computing the hash function

**Challenge.** How to efficiently compute $x_{i+1}$ given that we know $x_i$.

$$
\begin{align*}
x_i &= t_i R^{M-1} + t_{i+1} R^{M-2} + \ldots + t_{i+M-1} R^0 \\
x_{i+1} &= t_{i+1} R^M + t_{i+2} R^{M-1} + \ldots + t_{i+M} R^1
\end{align*}
$$

**Key observation.** Can do it in constant time!

$$x_{i+1} = (x_i - t_i R^{M-1}) R + t_{i+1}$$

---

Rabin-Karp: Java implementation

```java
public class RabinKarp {
    private char[] pat;      // the pattern
    private int patHash;     // pattern hash value
    private int M;           // pattern length
    private int Q = 8355967; // modulus
    private int R;           // radix
    private int RM;          // R^(M-1) % Q

    public RabinKarp(int R, char[] pat) {
        this.R = R;
        this.pat = pat;
        this.M = pat.length;
        RM = 1;
        for (int i = 1; i <= M-1; i++)
            RM = (R * RM) % Q;
        patHash = hash(pat);
    }

    private int hash(char[] key) {
        /* as before */
    }

    public int search(char[] txt) {
        /* see next slide */
    }
}
```

---

Rabin-Karp substring search example

```java
public int search(char[] txt) {
    int N = txt.length;
    if (N < M) return N;
    int offset = hashSearch(txt);
    if (offset == N) return N;
    for (int i = 0; i < M; i++)
        if (pat[i] != txt[offset + i])
            return N;
    return offset;
}
```

```java
private int hashSearch(char[] txt) {
    int N = txt.length;
    int txtHash = hash(txt);
    if (patHash == txtHash) return 0;
    for (int i = M; i < N; i++) {
        txtHash = (txtHash + Q - RM*txt[i-M] % Q) % Q;
        txtHash = (txtHash*R + txt[i]) % Q;
        if (patHash == txtHash) return i - M + 1;
    }
    return N;
}
```

---

Rabin-Karp: Java implementation (continued)

```java
public int search(char[] txt) {
    int N = txt.length;
    if (N < M) return N;
    int offset = hashSearch(txt);
    if (offset == N) return N;
    for (int i = 0; i < M; i++)
        if (pat[i] != txt[offset + i])
            return N;
    return offset;
}
```

---

Rabin-Karp substring search example

```plaintext

<table>
<thead>
<tr>
<th>txt[]</th>
<th>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 1 4 1 5 9 2 6 5 1 5 8 7 9 3</td>
<td></td>
</tr>
<tr>
<td>0 3 % 997 = 3</td>
<td></td>
</tr>
<tr>
<td>1 3 1 % 997 = (3*10 + 1) % 997 = 31</td>
<td></td>
</tr>
<tr>
<td>2 1 4 % 997 = (31*10 + 4) % 997 = 314</td>
<td></td>
</tr>
<tr>
<td>3 1 4 1 % 997 = (314*10 + 1) % 997 = 150</td>
<td></td>
</tr>
<tr>
<td>4 1 4 1 5 % 997 = (150*10 + 5) % 997 = 508</td>
<td></td>
</tr>
<tr>
<td>5 1 4 1 5 9 % 997 = ((508 + 3*997 - 310)*10 + 9) % 997 = 201</td>
<td></td>
</tr>
<tr>
<td>6 4 1 5 9 2 % 997 = ((201 + 1*997 - 310)*10 + 2) % 997 = 715</td>
<td></td>
</tr>
<tr>
<td>7 1 5 9 2 6 % 997 = ((715 + 4*997 - 310)*10 + 6) % 997 = 971</td>
<td></td>
</tr>
<tr>
<td>8 5 9 2 6 5 % 997 = ((971 + 5*997 - 310)*10 + 5) % 997 = 442</td>
<td></td>
</tr>
<tr>
<td>9 9 2 6 5 3 % 997 = ((442 + 5*997 - 310)*10 + 3) % 997 = 929</td>
<td></td>
</tr>
<tr>
<td>10 9 2 6 5 3 % 997 = ((929 + 9*997 - 310)*10 + 5) % 997 = 913</td>
<td></td>
</tr>
</tbody>
</table>
```
Rabin-Karp analysis

Property 4. Rabin-Karp substring search is extremely likely to be linear-time.

Worst-case. Takes time proportional to MN.
• In worst case, all substrings hash to same value.
• Then, need to check for match at each text position.

Theory. If Q is a sufficiently large random prime (about \(MN^2\)), then probability of a false collision is about \(1/N\) = expected running time is linear.

Practice. Choose Q to avoid integer overflow. Under reasonable assumptions, probability of a collision is about \(1/Q\) = linear in practice.

Rabin-Karp fingerprint search

Advantages.
• Extends to 2D patterns.
• Extends to finding multiple patterns.

Disadvantages.
• Arithmetic ops slower than char compares.
• No worst-case guarantee.

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 (data structure)</th>
<th>Operation count</th>
<th>Backup in input?</th>
<th>Space grows with input?</th>
</tr>
</thead>
<tbody>
<tr>
<td>brute force</td>
<td>(MN)</td>
<td>yes</td>
<td>1</td>
</tr>
<tr>
<td>Knuth-Morris-Pratt (full DFA)</td>
<td>2(N)</td>
<td>no</td>
<td>MR</td>
</tr>
<tr>
<td>Knuth-Morris-Pratt (mismatch transitions only)</td>
<td>3(N)</td>
<td>no</td>
<td>(M)</td>
</tr>
<tr>
<td>Boyer-Moore</td>
<td>3(N)</td>
<td>yes</td>
<td>(R)</td>
</tr>
<tr>
<td>Boyer-Moore (mismatched character heuristic only)</td>
<td>(MN)</td>
<td>yes</td>
<td>(R)</td>
</tr>
<tr>
<td>Rabin-Karp†</td>
<td>7(N)†</td>
<td>no</td>
<td>1</td>
</tr>
</tbody>
</table>

† probabilistic guarantee, with uniform hash function