5.3 Substring Search

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

Substring search

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

Typically $N \gg M$

```
pattern -> NEEDLE

text    --> I N A H A Y S T A C K NEEDLE I N A

match
```

Substring search applications

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

Typically $N \gg M$

```
pattern -> NEEDLE

text    --> I N A H A Y S T A C K NEEDLE I N A

match
```
**Substring search applications**

**Goal.** Find pattern of length $M$ in a text of length $N$. Typically $N \gg M$.

![Pattern match](http://citp.princeton.edu/memory)

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

- Identify patterns indicative of spam.
  - PROFITS
  - LOSE WEIGHT
  - herbal Viagra
  - There is no catch.
  - This is a one-time mailing.
  - This message is sent in compliance with spam regulations.

**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

**Screen scraping.** Extract relevant data from web page.

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

```html
<td class="yfnc_tablehead1" width="48%">Last Trade:<br />
<td class="yfnc_tabledata1" width="48%">Trade Time:</br />
```
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
class StockQuote {
    public static void main(String[] args) {
        String name = "http://finance.yahoo.com/q?s=";
        String text = in.readLine();
        int start = text.indexOf("Last Trade: ", 0);
        int to = text.indexOf("</b>", from);
        String price = text.substring(from + 3, to);
        StdOut.println(price);
    }
}
```

Brute-force substring search

Check for pattern starting at each text position.

```
i  j  i+j
0  1  2  3  4  5  6  7  8  9  10
  "ABACADABRA"
0   2   2   A  B  R  A  A  pat
1   0   1   A  B  R  A
2   1   3   A  B  R
3   0   3   A  B  R  A
4   1   5   A  B  R  A
5   0   5   A  B  R  A
6   4   10  A  B  R  A  match
```

Brute-force substring search: Java implementation

Check for pattern starting at each text position.

```
i  j  i+j
0  1  2  3  4  5  6  7  8  9  10
  "ABACADABRA"
4   3   7   A  D  A  C  R
5   0   5   A  D  A  C  R
```

```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;
        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
}
```
Brute-force substring search: worst case

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

```
i  j  i+j  0  1  2  3  4  5  6  7  8  9
  txt   A  A  A  A  A  A  A  A  A  A  A  A  A  A  B
  0   4   4   A  A  A  A  B
  1   4   5   A  A  A  A  B
  2   4   6   A  A  A  A  B
  3   4   7   A  A  A  A  B
  4   4   8   A  A  A  A  B
  5   5   10  A  A  A  A  B
```

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

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 # of already-matched chars (end of sequence in pattern).

```
i  j  0  1  2  3  4  5  6  7  8  9  10
  A  B  A  C  A  D  A  B  R  A  C
  7   3   A  D  A  C  R
  5   0   A  D  A  C  R
```

```java
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 = 0; }
    }
    if (j == M) return i - M;
    else return N;
}
```

Backup

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

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

```
Backup algorithm needs backup for every mismatch.
```

```
\[
\begin{array}{c}
\text{matched chars} \\
\text{mismatch} \\
\text{shift pattern right one position}
\end{array}
\]
```

Approach 1. Maintain buffer of last \( M \) characters.
Approach 2. Stay tuned.

Algorithmic challenges in substring search

Brute-force is not always good enough.

**Theoretical challenge.** Linear-time guarantee.

**Practical challenge.** Avoid backup in text stream.

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 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 good people to come to the aid of their party.
```java
public static int search(String pat, String txt) {
    explicit backup
    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 = 0; }
    }
    if (j == M) return i - M;
    else return N;
}
```

often no room or time to save text
5.3 SUBSTRING SEARCH

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

Knuth-Morris-Pratt substring search

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

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

- 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.

- internal representation

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
C & 0 & 2 & 0 & 4 & 0 \\
\end{array}
\]

if in state \(j\) reading char \(c\):
- if \(j\) is 6 halt and accept
- else move to state \(\text{dfa}[c][j]\)

- graphical representation

Knuth-Morris-Pratt demo: DFA simulation

- A A B A C A A B A B A C A A

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
A & B & A & B & A & C \\
A & A & B & 1 & 3 & 1 \\
B | 0 & 2 & 0 & 4 & 0 & 4 \\
C | 0 & 0 & 0 & 0 & 0 & 6 \\
\end{array}
\]
Knuth-Morris-Pratt demo: DFA simulation

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

Knuth-Morris-Pratt substring search: Java implementation

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

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? [warning: tricky algorithm ahead]

Interpretation of Knuth-Morris-Pratt DFA

Q. What is interpretation of DFA state after reading in txt[i]?
A. State = number of characters in pattern that have been matched.

Ex. DFA is in state 3 after reading in txt[0..6].

Knuth-Morris-Pratt substring search: Java implementation

Key differences from brute-force implementation.
- Need to precompute dfa[][] from pattern.
- Text pointer i never decrements.
- Could use input stream.

public int search(In in)
{
    int i, j;
    for (i = 0; j = 0; !in.isEmpty() && j < M; i++)
        j = dfa[in.readChar()][j];
    if (j == M) return i - M;
    else return NOT_FOUND;
}
Knuth-Morris-Pratt demo: DFA construction

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

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

0 1 2 3 4 5 6

How to build DFA from pattern?

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

Match transition. If in state j and next char c == 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
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 $c$. Running time. Seems to require $j$ steps.

**Ex.** $\text{dfa}[\text{\textquoteleft A\textquoteright}][5] = 1$; $\text{dfa}[\text{\textquoteleft B\textquoteright}][5] = 4$

simulate BABA; take transition \textquoteleft A\textquoteright; $\text{dfa}[\text{\textquoteleft A\textquoteright}][3]$

simulate BABA; take transition \textquoteleft B\textquoteright; $\text{dfa}[\text{\textquoteleft B\textquoteright}][3]$

How to build DFA from pattern?

Knuth-Morris-Pratt demo: DFA construction in linear time

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

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

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 $c$. Running time. Takes only constant time if we maintain state $X$.

**Ex.** $\text{dfa}[\text{\textquoteleft A\textquoteright}][5] = 1$; $\text{dfa}[\text{\textquoteleft B\textquoteright}][5] = 4$ $X = 0$

from state $X$, take transition \textquoteleft A\textquoteright; $\text{dfa}[\text{\textquoteleft A\textquoteright}][X]$

from state $X$, take transition \textquoteleft B\textquoteright; $\text{dfa}[\text{\textquoteleft B\textquoteright}][X]$

from state $X$, take transition \textquoteleft C\textquoteright; $\text{dfa}[\text{\textquoteleft C\textquoteright}][X]$

Knuth-Morris-Pratt demo: DFA construction in linear time

Constructing the DFA for KMP substring search for A B A B A C
Knuth-Morris-Pratt: brief history

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

5.3 Substring Search

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

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 \( dfa[] \) in time and space proportional to \( RM \).

Larger alphabets. Improved version of KMP constructs \( nfa[] \) in time and space proportional to \( M \).
**Intuition.**
- Scan characters in pattern from right to left.
- Can skip as many as $M$ text chars when finding one not in the pattern.

**Case 1.** Mismatch character not in pattern.

**Case 2a.** Mismatch character in pattern.

**Case 2b.** Mismatch character in pattern (but heuristic no help).
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></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>txt</td>
<td></td>
<td></td>
<td></td>
<td>E</td>
<td>L</td>
<td>E</td>
</tr>
<tr>
<td>pat</td>
<td>N</td>
<td>E</td>
<td>E</td>
<td>D</td>
<td>L</td>
<td>E</td>
</tr>
</tbody>
</table>

i ---->

after

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>txt</td>
<td></td>
<td></td>
<td></td>
<td>E</td>
<td>L</td>
<td>E</td>
</tr>
<tr>
<td>pat</td>
<td>N</td>
<td>E</td>
<td>E</td>
<td>D</td>
<td>L</td>
<td>E</td>
</tr>
</tbody>
</table>
```

Mismatch character 'E' in pattern: increment i by 1

```
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; // match
    }
    return N;
}
```

Boyer-Moore: Java implementation

```
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;
```

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

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

```
i skip  0 1 2 3 4 5 6 7 8 9
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>txt</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>pat</td>
<td>A</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>B</td>
<td>B</td>
</tr>
</tbody>
</table>
```

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

- Introduction
- Brute force
- Knuth-Morris-Pratt
- Boyer-Moore
- Rabin-Karp

Rabin-Karp fingerprint search

Basic idea = modular hashing.

- Compute a hash of pat[0..M-1].
- For each i, compute a hash of txt[i..M+i-1].
- If pattern hash = text substring hash, check for a match.

Efficiently computing the hash function

Modular hash function. Using the notation ti for txt.charAt(i), we wish to compute

\[ x_i = t_i R^{i-1} + t_{i+1} R^{i-2} + \ldots + t_{i+M-1} R^0 \pmod Q \]

Intuition. M-digit, base-R integer, modulo Q.

Horner’s method. Linear-time method to evaluate degree-M polynomial.

Modular arithmetic

Math trick. To keep numbers small, take intermediate results modulo Q.

Ex. \[(10000 + 535) \times 1000 \pmod{997} \]
\[= (30 + 535) \times 3 \pmod{997} \]
\[= 1695 \pmod{997} \]
\[= 698 \pmod{997} \]

Two useful modular arithmetic identities

\[(a + b) \pmod Q = ((a \pmod Q) + (b \pmod Q)) \pmod Q \]
\[(a \times b) \pmod Q = ((a \pmod Q) \times (b \pmod Q)) \pmod Q \]
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 "rolling" hash function in constant time!

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

<table>
<thead>
<tr>
<th>$i$</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>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_i$</td>
<td>314</td>
<td>442</td>
<td>442</td>
<td>613</td>
<td>314</td>
<td>314</td>
<td>314</td>
<td>314</td>
<td>314</td>
<td>314</td>
<td>314</td>
<td>314</td>
<td>314</td>
<td>314</td>
<td>314</td>
<td></td>
</tr>
</tbody>
</table>

Key computation in Rabin-Karp substring search

- Subtract leading digit
- Multiply by radix
- Add new trailing digit

Rabin-Karp: Java implementation

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

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

    public long hash(String pat, int M) {
        long hash = 0;
        for (int i = 0; i < M; i++)
            hash = (hash * R + pat.charAt(i)) % Q;
        return hash;
    }

    private int search(String txt) {
        int N = txt.length();
        int txtHash = hash(txt, M);
        if (patHash == txtHash) return 0;
        for (int i = 0; i < N; i++)
            if (patHash == txtHash)
                return i - M + 1;
        return N;
    }
}
```

Rabin-Karp substring search example

**First R entries:** Use Horner’s rule.

**Remaining entries:** Use rolling hash (and % to avoid overflow).

<table>
<thead>
<tr>
<th>$i$</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>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_i$</td>
<td>314</td>
<td>442</td>
<td>442</td>
<td>613</td>
<td>314</td>
<td>314</td>
<td>314</td>
<td>314</td>
<td>314</td>
<td>314</td>
<td>314</td>
<td>314</td>
<td>314</td>
<td>314</td>
<td>314</td>
<td></td>
</tr>
</tbody>
</table>

Rabin-Karp: Java implementation (continued)

**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 = 0; i < N; i++)
        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 \( MN^2 \)), then the probability of a false collision is about \( 1/N \).

**Practice.** Choose \( Q \) to be a large prime (but not so large 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 \( MN \)).

---

Rabin-Karp fingerprint search

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

**Disadvantages.**
- Arithmetic ops slower than char compares.
- Las Vegas version requires backup.
- Poor 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</th>
<th>version</th>
<th>operation count</th>
<th>backup in input?</th>
<th>correct?</th>
<th>extra space</th>
<th>guarantee</th>
</tr>
</thead>
<tbody>
<tr>
<td>brute force</td>
<td>—</td>
<td>( MN )</td>
<td>1.1 N</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>1.1 N</td>
<td>no</td>
<td>yes</td>
<td>MR</td>
</tr>
<tr>
<td></td>
<td>mismatch transitions only</td>
<td>( 3N )</td>
<td>1.1 N</td>
<td>no</td>
<td>yes</td>
<td>( M )</td>
</tr>
<tr>
<td>Boyer-Moore</td>
<td>full algorithm (Algorithm 5.7)</td>
<td>( 3N )</td>
<td>( N/M )</td>
<td>yes</td>
<td>yes</td>
<td>R</td>
</tr>
<tr>
<td></td>
<td>mismatched char heuristic only</td>
<td>( MN )</td>
<td>( N/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 )</td>
<td>7 N</td>
<td>no</td>
<td>yes†</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Las Vegas</td>
<td>( 7N^{*} )</td>
<td>7 N</td>
<td>yes</td>
<td>yes</td>
<td>1</td>
</tr>
</tbody>
</table>

† probabilistic guarantee, with uniform hash function