String Matching – What’s behind Ctrl+F?

24 February 2020

Sebastian Wild
4 String Matching

4.1 Introduction
4.2 Brute Force
4.3 String Matching with Finite Automata
4.4 The Knuth-Morris-Pratt algorithm
4.5 Beyond Optimal? The Boyer-Moore Algorithm
4.6 The Rabin-Karp Algorithm
4.1 Introduction
Ubiquitous strings

\textit{string} = sequence of characters

- universal data type for . . . everything!
  - natural language texts
  - programs (source code)
  - websites
  - XML documents
  - DNA sequences
  - bitstrings
- . . . a computer’s memory $\rightsquigarrow$ ultimately any data is a string

$\rightsquigarrow$ many different tasks and algorithms
Ubiquitous strings

$string = sequence of characters$

▶ universal data type for . . . everything!
  ▶ natural language texts
  ▶ programs (source code)
  ▶ websites
  ▶ XML documents
  ▶ DNA sequences
  ▶ bitstrings
  ▶ . . . a computer’s memory  ~~~ ultimately any data is a string

~~~ many different tasks and algorithms

▶ This unit: finding (exact) occurrences of a pattern text.
  ▶ Ctrl+F
  ▶ grep
  ▶ computer forensics (e.g. find signature of file on disk)
  ▶ virus scanner

▶ basis for many advanced applications
Notations

- **alphabet** $\Sigma$: finite set of allowed **characters**; $\sigma = |\Sigma|$ \quad “a string over alphabet $\Sigma$”
  - letters (Latin, Greek, Arabic, Cyrillic, Asian scripts, . . . )
  - “what you can type on a keyboard”, Unicode characters
  - $\{0, 1\}$; nucleotides $\{A, C, G, T\}$; . . .

comprehensive standard character set including emoji and all known symbols
Notations

- \textit{alphabet} \( \Sigma \): finite set of allowed \textbf{characters}; \( \sigma = |\Sigma| \) \quad “a string over alphabet \( \Sigma \)”
  - letters \quad (Latin, Greek, Arabic, Cyrillic, Asian scripts, . . .)
  - “what you can type on a keyboard”, \quad Unicode characters
  - \{0, 1\}; \quad \text{nucleotides} \{A, C, G, T\}; \ldots
- \( \Sigma^n = \Sigma \times \cdots \times \Sigma \): strings of \textbf{length} \( n \in \mathbb{N}_0 \) \((n\text{-tuples})\)
- \( \Sigma^* = \bigcup_{n\geq 0} \Sigma^n \): set of \textbf{all} (finite) strings over \( \Sigma \)
- \( \Sigma^+ = \bigcup_{n\geq 1} \Sigma^n \): set of \textbf{all} (finite) \textbf{nonempty} strings over \( \Sigma \)
- \( \epsilon \in \Sigma^0 \): the \textbf{empty} string \quad (same for all alphabets)
Notations

- **alphabet** $\Sigma$: finite set of allowed characters; $\sigma = |\Sigma|$  
  “a string over alphabet $\Sigma$”
  - letters (Latin, Greek, Arabic, Cyrillic, Asian scripts, . . .)
  - “what you can type on a keyboard”, Unicode characters
  - $\{0, 1\}$; nucleotides $\{A, C, G, T\}$; . . .

- $\Sigma^n = \Sigma \times \cdots \times \Sigma$: strings of **length** $n \in \mathbb{N}_0$ ($n$-tuples)

- $\Sigma^* = \bigcup_{n \geq 0} \Sigma^n$: set of all (finite) strings over $\Sigma$

- $\Sigma^+ = \bigcup_{n \geq 1} \Sigma^n$: set of all (finite) **nonempty** strings over $\Sigma$

- $\varepsilon \in \Sigma^0$: the **empty** string  
  (same for all alphabets)

- for $S \in \Sigma^n$, write $S[i]$ (other sources: $S_i$) for $i$th character  
  $(0 \leq i < n)$

- for $S, T \in \Sigma^*$, write $ST = S \cdot T$ for **concatenation** of $S$ and $T$

- for $S \in \Sigma^n$, write $S[i..j]$ or $S_{i,j}$ for the **substring** $S[i] \cdot S[i+1] \cdots S[j]$  
  $(0 \leq i \leq j < n)$
  - $S[0..j]$ is a **prefix** of $S$; $S[i..n-1]$ is a **suffix** of $S$
  - $S[i..j] = S[i..j \leftarrow 1]$ (endpoint exclusive)  
  $\leadsto S = S[0..n]$
True or false: $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$

A  True  B  False
True or false: $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$

A True ✓ B False
String matching – Definition

Search for a string (pattern) in a large body of text

- **Input:**
  - $T \in \Sigma^n$: The text (haystack) being searched within
  - $P \in \Sigma^m$: The pattern (needle) being searched for; typically $n \gg m$

- **Output:**
  - the first occurrence (match) of $P$ in $T$: $\min \{ i \in [0..n-m) : T[i..i+m) = P \}$
  - or NO_MATCH if there is no such $i$ (“$P$ does not occur in $T$”)

- **Variant:** Find all occurrences of $P$ in $T$.
  - Can do that iteratively (update $T$ to $T[i+1..n]$ after match at $i$)

- **Example:**
  - $T$ = “Where is he?”
  - $P_1$ = “he” $\leadsto i = 1$
  - $P_2$ = “who” $\leadsto$ NO_MATCH

- string matching is implemented in Java in `String.indexOf`
4.2 Brute Force
Abstract idea of algorithms

Pattern matching algorithms consist of guesses and checks:

- A **guess** is a position $i$ such that $P$ might start at $T[i]$. Possible guesses (initially) are $0 \leq i \leq n - m$.

- A **check** of a guess is a pair $(i, j)$ where we compare $T[i + j]$ to $P[j]$.

- Note: need all $m$ checks to verify a single correct guess $i$, but it may take (many) fewer checks to recognize an incorrect guess.

- Cost measure: #character comparisons = #checks

\[ \text{cost} \leq n \cdot m \quad \text{(number of possible checks)} \]
# Brute-force method

```plaintext
procedure bruteForceSM(T[0..n], P[0..m])
    for i := 0, ..., n - m - 1 do
        for j := 0, ..., m - 1 do
            if T[i + j] ≠ P[j] then break inner loop
        if j == m then return i
    return NO_MATCH
```

- try all guesses $i$
- check each guess (left to right); stop early on mismatch
- essentially the implementation in Java!

## Example:

| $T$ = abbbababbbab |
| $P$ = abba |

→ 15 char cmps (vs $n \cdot m = 44$) not too bad!
Brute-force method

1 procedure bruteForceSM(T[0..n), P[0..m])
2     for i := 0, . . . , n − m − 1 do
3         for j := 0, . . . , m − 1 do
4             if T[i + j] ≠ P[j] then break inner loop
5             if j == m then return i
6     return NO_MATCH

► try all guesses i
► check each guess (left to right); stop early on mismatch
► essentially the implementation in Java!

Example:
T = abbbababbab
P = abba

~ 15 char cmps
(vs n · m = 44)
not too bad!
Brute-force method – Discussion

👍 Brute-force method can be good enough
  - typically works well for natural language text
  - also for random strings

👎 but: can be as bad as it gets!

Worst possible input: $P = a^{m-1} b,$ $T = a^n$

Worst-case performance: $(n - m + 1) \cdot m$ for $m \leq n/2$ that is $\Theta(m n)$
Brute-force method – Discussion

👍 Brute-force method can be good enough
  ▶ typically works well for natural language text
  ▶ also for random strings

👎 but: can be as bad as it gets!

Worst possible input: \( P = a^{m-1}b, \)
\( T = a^n \)

Worst-case performance: \((n - m + 1) \cdot m\)
\( \sim \) for \( m \leq n/2 \) that is \( \Theta(mn) \)

👍 Bad input: lots of self-similarity in \( T! \) \( \sim \) can we exploit that?

👍 brute force does ‘obviously’ stupid repetitive comparisons \( \sim \) can we avoid that?
Roadmap

- **Approach 1** (this week): Use *preprocessing* on the pattern $P$ to eliminate guesses (avoid ‘obvious’ redundant work)
  - Deterministic finite automata (DFA)
  - Knuth-Morris-Pratt algorithm
  - Boyer-Moore algorithm
  - Rabin-Karp algorithm

- **Approach 2** (≈ Unit 6): Do preprocessing on the text $T$
  Can find matches in time *independent of text size(!)*
  - inverted indices
  - Suffix trees
  - Suffix arrays
4.3 String Matching with Finite Automata
Clicker Question

Do you know what regular expressions, NFAs and DFAs are, and how to convert between them?

A. Never heard of this; are these new emoji?
B. Heard the terms, but don’t remember conversion methods.
C. Had that in my undergrad course (memories fading a bit).
D. Sure, I could do that blindfolded!

pingo.upb.de/622222
Theoretical Computer Science to the rescue!

- string matching = deciding whether \( T \in \Sigma^* \cdot P \cdot \Sigma^* \)
- \( \Sigma^* \cdot P \cdot \Sigma^* \) is *regular* formal language
- \( \exists \) deterministic finite automaton (DFA) to recognize \( \Sigma^* \cdot P \cdot \Sigma^* \)
- can check for occurrence of \( P \) in \( |T| = n \) steps!
Theoretical Computer Science to the rescue!

- **string matching** = deciding whether $T \in \Sigma^* \cdot P \cdot \Sigma^*$
- $\Sigma^* \cdot P \cdot \Sigma^*$ is *regular* formal language

$\exists$ deterministic finite automaton (DFA) to recognize $\Sigma^* \cdot P \cdot \Sigma^*$

$\exists$ can check for occurrence of $P$ in $|T| = n$ steps!

Job done!
Theoretical Computer Science to the rescue!

- string matching = deciding whether $T \in \Sigma^* \cdot P \cdot \Sigma^*$
- $\Sigma^* \cdot P \cdot \Sigma^*$ is regular formal language

$\exists$ deterministic finite automaton (DFA) to recognize $\Sigma^* \cdot P \cdot \Sigma^*$

$\exists$ can check for occurrence of $P$ in $|T| = n$ steps!

Job done! WTF!?
Theoretical Computer Science to the rescue!

- string matching = deciding whether $T \in \Sigma^* \cdot P \cdot \Sigma^*$
- $\Sigma^* \cdot P \cdot \Sigma^*$ is regular formal language

\[ \exists \text{deterministic finite automaton (DFA) to recognize } \Sigma^* \cdot P \cdot \Sigma^* \]

\[ \to \text{can check for occurrence of } P \text{ in } |T| = n \text{ steps!} \]

Job done!  

WTF!?

We are not quite done yet.

- (Problem 0: programmer might not know automata and formal languages . . . )
- Problem 1: existence alone does not give an algorithm!
- Problem 2: automaton could be very big!
String matching with DFA

- Assume first, we already have a deterministic automaton
- How does string matching work?

Example:

\( T = \text{aabacaababacaa} \)
\( P = \text{ababac\(\alpha\))} \)

![DFA diagram]

<table>
<thead>
<tr>
<th>text:</th>
<th>a</th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>c</th>
<th>a</th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>c</th>
<th>a</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>state:</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
</tbody>
</table>
String matching with DFA

- Assume first, we already have a deterministic automaton
- How does string matching work?

Example:

\[ T = \text{aabacaababacaa} \]
\[ P = \text{ababaca} \]

![DFA Diagram]

<table>
<thead>
<tr>
<th>text:</th>
<th>a</th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>c</th>
<th>a</th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>c</th>
<th>a</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>state:</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
String matching DFA – Intuition

Why does this work?

▶ Main insight: \[ \text{Invariant} \]

State \( q \) means: "we have seen \( P[0..q) \) until here (but not any longer prefix of \( P \)"

▶ If the next text character \( c \) does not match, we know:

(i) text seen so far ends with \( P[0..q) \cdot c \)
(ii) \( P[0..q) \cdot c \) is not a prefix of \( P \)
(iii) without reading \( c \), \( P[0..q) \) was the longest prefix of \( P \) that ends here.

\[ \sim \] New longest matched prefix will be (weakly) shorter than \( q \)

\[ \sim \] All information about the text needed to determine it is contained in \( P[0..q) \cdot c ! \)
NFA instead of DFA?

It remains to construct the DFA.

▶ trivial part:
NFA instead of DFA?

It remains to *construct* the DFA.

- trivial part:

- that actually is a *nondeterministic finite automaton* (NFA) for $\Sigma^* P \Sigma^*$

~~ We *could* use the NFA directly for string matching:

- at any point in time, we are in a *set of states*
- accept when one of them is final state

**Example:** Previous versions of this example were missing states; this is the correct version:

<table>
<thead>
<tr>
<th>text:</th>
<th>a</th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>c</th>
<th>a</th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>c</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>state:</td>
<td>0</td>
<td>0,1</td>
<td>0,1</td>
<td>0,2</td>
<td>0,1,3</td>
<td>0</td>
<td>0,1</td>
<td>0,2</td>
<td>0,1,3</td>
<td>0,2,4</td>
<td>0,1,3,5</td>
<td>0,6</td>
<td>0,1,7</td>
</tr>
</tbody>
</table>

But maintaining a whole set makes this slow . . .
Computing DFA directly

You have an NFA and want a DFA?
Simply apply the power-set construction
(and maybe DFA minimization)!

The powerset method has exponential state blow up!
I guess I might as well use brute force ...
Computing DFA directly

You have an NFA and want a DFA?
Simply apply the power-set construction
(and maybe DFA minimization)!

The powerset method has exponential state blow up!
I guess I might as well use brute force ...

Ingenious algorithm by Knuth, Morris, and Pratt: construct DFA inductively:
Suppose we add character $P[j]$ to automaton $A_{j-1}$ for $P[0..j-1]$

- add new state and matching transition $\sim$ easy
- for each $c \neq P[j]$, we need $\delta(j, c)$ (transition from $j$ when reading $c$)
Computing DFA directly

You have an NFA and want a DFA?
Simply apply the power-set construction
(and maybe DFA minimization):

The powerset method has exponential state blow up!
I guess I might as well use brute force...

Ingenious algorithm by Knuth, Morris, and Pratt: construct DFA inductively:

Suppose we add character \(P[j]\) to automaton \(A_{j-1}\) for \(P[0..j-1]\)

- add new state and matching transition \(\leadsto\) easy
- for each \(c \neq P[j]\), we need \(\delta(j, c)\) (transition from \(j\) when reading \(c\))
- \(\delta(j, c) = \) length of the longest prefix of \(P[0..j]c\) that is a suffix of \(P[1..j]c\)
  \(= \) state of automaton after reading \(P[1..j]c\)
  \(\leq j \leadsto\) can use known automaton \(A_{j-1}\) for that!

\(\leadsto\) can directly compute \(A_j\) from \(A_{j-1}\)!

\(\nabla\) seems to require simulating automata \(m \cdot \sigma\) times

State \(q\) means:
"we have seen \(P[0..q]\) until here (but not any longer prefix of \(P\)"

\(\nabla\)
Computing DFA efficiently

- **KMP’s second insight:** simulations in one step differ only in last symbol

  ~ maintain state \( x \), the state after reading \( P[1..j-1] \).
  - copy its transitions
  - update \( x \) by following transitions for \( P[j] \)

**Demo:** Algorithms videos of Sedgewick and Wayne

https://cuvids.io/app/video/194/watch
String matching with DFA – Discussion

- **Time:**
  - Matching: $n$ table lookups for DFA transitions
  - building DFA: $\Theta(m\sigma)$ time (constant time per transition edge).
  - $\sim \Theta(m\sigma + n)$ time for string matching.

- **Space:**
  - $\Theta(m\sigma)$ space for transition matrix.

👍 fast matching time actually: hard to beat!

👍 total time asymptotically optimal for small alphabet (for $\sigma = O(n/m)$)

👎 substantial space overhead, in particular for large alphabets
4.4 The Knuth-Morris-Pratt algorithm
Failure Links

- Recall: String matching with is DFA fast, but needs table of $m \times \sigma$ transitions.
- in fast DFA construction, we used that all simulations differ only by last symbol

〜 **KMP’s third insight:** do this last step of simulation from state $x$ during matching! … but how?
Failure Links

- Recall: String matching with is DFA fast, but needs table of $m \times \sigma$ transitions.

- in fast DFA construction, we used that all simulations differ only by last symbol

≈ KMP’s third insight: do this last step of simulation from state $x$ during matching! . . . but how?

Answer: Use a new type of transition, the failure links

- Use this transition (only) if no other one fits.
- $\times$ does not consume a character. ≈ might follow several failure links

≈ Computations are deterministic (but automaton is not a real DFA.)
Failure link automaton – Example

Example: \( T = abababaaca, \ P = ababaca \)

\[
\begin{array}{cccccccccccc}
T: & a & b & a & b & a & b & a & a & a & b & a & b \\
0 & 1 & 2 & 3 & 4 & 5,8 & 4 & 5,3,1,0 & 1
\end{array}
\]
**Failure link automaton – Example**

**Example:** $T = \text{abababaaca}$, $P = \text{ababaca}$

For failure link construction, simulate on $P(\ldots \cdot) = \text{babac \ldots}$

![Diagram of automaton]

<table>
<thead>
<tr>
<th>$\Sigma - a$</th>
<th>$a$</th>
<th>$b$</th>
<th>$a$</th>
<th>$b$</th>
<th>$a$</th>
<th>$b$</th>
<th>$a$</th>
<th>$a$</th>
<th>$b$</th>
<th>$a$</th>
<th>$b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>$P$</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>(\times)</td>
<td>(\times)</td>
<td>(\times)</td>
<td>(\times)</td>
<td>(\times)</td>
<td>(\times)</td>
</tr>
<tr>
<td>$q$</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>3,4</td>
<td>5</td>
<td>3,1,0,1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

(after reading this character)
Clicker Question

What is the worst-case time to process one character in a failure-link automaton for $P[0..m]$?

A. $\Theta(1)$
B. $\Theta(\log m)$
C. $\Theta(m)$
D. $\Theta(m^2)$

pingo.upb.de/622222
What is the worst-case time to process one character in a failure-link automaton for $P[0..m)$?

- $\Theta(1)$
- $\Theta(\log m)$
- $\Theta(m)$
- $\Theta(m^2)$
The Knuth-Morris-Pratt Algorithm

procedure KMP(T[0..n - 1], P[0..m - 1])

fail[0..m] := failureLinks(P)

i := 0 // current position in T
q := 0 // current state of KMP automaton

while i < n do

if T[i] == P[q] then

i := i + 1; q := q + 1

if q == m then

return i - q // occurrence found

else // i.e. T[i] ≠ P[q]

if q ≥ 1 then

q := fail[q] // follow one ×

else

i := i + 1

end while

return NO_MATCH

► only need single array fail for failure links

► (procedure failureLinks later)
The Knuth-Morris-Pratt Algorithm

```plaintext
procedure KMP(T[0..n-1], P[0..m-1])
    fail[0..m] := failureLinks(P)
    i := 0 // current position in T
    q := 0 // current state of KMP automaton
    while i < n do
        if T[i] == P[q] then
            i := i + 1; q := q + 1
            if q == m then
                return i - q // occurrence found
        else // i.e. T[i] ≠ P[q]
            if q ≥ 1 then
                q := fail[q] // follow one ×
            else
                i := i + 1
        end while
    return NO_MATCH
```

- only need single array `fail` for failure links
- (procedure `failureLinks` later)

**Analysis:** (matching part)

- always have `fail[j] < j` for `j ≥ 1`
- in each iteration
  - either advance position in text (`i := i + 1`)
  - or shift pattern forward (guess `i - j`)
- each can happen at most `n` times
- $\Rightarrow \leq 2n$ symbol comparisons!
  $\Rightarrow O(1)$ time per character on average
Computing failure links

- failure links point to error state $x$ (from DFA construction)

- run same algorithm, but store $\text{fail}[j] := x$ instead of copying all transitions

```plaintext
procedure failureLinks($P[0..m-1]$)
    fail[0] := 0
    $x := 0$
    for $j := 1, \ldots, m-1$ do
        fail[$j$] := $x$
        // update failure state using failure links:
        while $P[x] \neq P[j]$
            if $x == 0$ then
                $x := -1$; break
            else
                $x := \text{fail}[x]$
            end if
        end while
        $x := x + 1$
    end for
```
Computing failure links

- failure links point to error state $x$ (from DFA construction)

$\rightsarrow$ run same algorithm, but store $\text{fail}[j] := x$ instead of copying all transitions

```
procedure failureLinks($P[0..m-1]$)
    $\text{fail}[0] := 0$
    $x := 0$
    for $j := 1, \ldots, m-1$ do
        $\text{fail}[j] := x$
        // update failure state using failure links:
        while $P[x] \neq P[j]$
            if $x == 0$ then
                $x := -1$; break
            else
                $x := \text{fail}[x]$
            end if
        end while
        $x := x + 1$
    end for
```

Analysis:

- $m$ iterations of for loop
- while loop always decrements $x$
- $x$ is incremented only once per iteration of for loop

$\rightsarrow \leq m$ iterations of while loop in total

$\rightsarrow \leq 2m$ symbol comparisons
Knuth-Morris-Pratt – Discussion

▶ Time:
  ▶ \( \leq 2n + 2m = O(n + m) \) character comparisons
  ▶ clearly must at least read both \( T \) and \( P \)
  ≈ KMP has optimal worst-case complexity!

▶ Space:
  ▶ \( \Theta(m) \) space for failure links

👍 total time asymptotically optimal (for any alphabet size)
👍 reasonable extra space
Clicker Question

What are the main advantages of the KMP string matching (using the failure-link automaton) over string matching with DFAs? Check all that apply.

A. faster preprocessing on pattern
B. faster matching in text
C. fewer character comparisons
D. uses less space
E. makes running time independent of $\sigma$
F. I don’t have to do automata theory
Clicker Question

What are the main advantages of the KMP string matching (using the failure-link automaton) over string matching with DFAs? Check all that apply.

A. faster preprocessing on pattern ✓
B. faster matching in text
C. fewer character comparisons
D. uses less space ✓
E. makes running time independent of \( \sigma \) ✓
F. I don’t have to do automata theory

pingo.upb.de/622222
The KMP prefix function

- It turns out that the failure links are useful beyond KMP

- A slight variation is more widely used: (for historic reasons)
  the (KMP) prefix function $F : [1..m - 1] \rightarrow [0..m - 1]$:

  \[ F[j] \text{ is the length of the longest prefix of } P[0..j] \]
  \[ \text{that is a suffix of } P[1..j]. \]

- Can show: $\text{fail}[j] = F[j - 1]$ for $j \geq 1$, and hence

  \[ \text{fail}[j] = \text{length of the longest prefix of } P[0..j] \]
  \[ \text{that is a suffix of } P[1..j]. \]
4.5 Beyond Optimal? The Boyer-Moore Algorithm
Motivation

- KMP is an optimal algorithm, isn’t it? What else could we hope for?
**Motivation**

- KMP is an optimal algorithm, isn’t it? What else could we hope for?

- KMP is “only” optimal in the worst-case (and up to constant factors)

- how many comparisons do we need for the following instance?

  \[ T = \text{aaaaaaaaaaaaaaaaaa}, \ P = xxxx \]

  - there are no matches
  - we can *certify* the correctness of that output with only 4 comparisons:

  |   | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a |
  |---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
  | T | x |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
  |   | x |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
  |   |   |   | x | x |   | x |   | x |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

  ~~~ We did *not* even read most text characters!
Boyer-Moore Algorithm

» Let’s check guesses *from right to left*!

» If we are lucky, we can eliminate several shifts in one shot!
Boyer-Moore Algorithm

- Let’s check guesses *from right to left*!
- If we are lucky, we can eliminate several shifts in one shot!

⚠ must avoid (excessive) redundant checks, e.g., for $T = a^n$, $P = ba^{m-1}$

~~ New rules:

- **Bad character jumps**: Upon mismatch at $T[i] = c$:
  - If $P$ does not contain $c$, shift $P$ entirely past $i$!
  - Otherwise, shift $P$ to align the *last occurrence* of $c$ in $P$ with $T[i]$.

- **Good suffix jumps**:
  Upon a mismatch, shift so that the already matched *suffix* of $P$ aligns with a previous occurrence of that suffix (or part of it) in $P$.
  (Details follow; ideas similar to KMP failure links)

~~ two possible shifts (next guesses); use larger jump.
Boyer-Moore Algorithm – Code

```plaintext
procedure boyerMoore(T[0..n-1], P[0..m-1])
    λ := computeLastOccurrences(P)
    γ := computeGoodSuffixes(P)
    i := 0 // current guess
    while i ≤ n - m
        j := m - 1 // next position in P to check
        while j ≥ 0 ∧ P[j] == T[i + j] do
            j := j - 1
        if j == -1 then
            return i
        else
            i := i + max{j - λ[T[i + j]], γ[j]}
    return NO_MATCH
```

- λ and γ explained below
- shift forward is larger of two heuristics
- shift is always positive (see below)
Bad character examples

\[ P = \text{aldo} \]
\[ T = \text{where is waldo} \]

\[ P = \text{moore} \]
\[ T = \text{boyer moore} \]
## Bad character examples

\[ P = \text{aldo} \]
\[ T = \text{where is waldo} \]

\[
\begin{array}{ccccccc}
0 & & & & & & \\
 & 0 & & & & & \\
\end{array}
\]

\[ P = \text{moore} \]
\[ T = \text{boyer moore} \]

\[
\begin{array}{ccccccc}
 & & & & & & \\
 & & & & & & \\
\end{array}
\]
Bad character examples

\[ P = \text{aldo} \]
\[ T = \text{where is waldo} \]

\[ P = \text{moore} \]
\[ T = \text{boyer moore} \]
Bad character examples

\[ P = \text{aldo} \]
\[ T = \text{where is waldo} \]

\[
\begin{array}{cccccc}
| & | & | & | & | & | \\
| & | & | & | & | & | \\
| & | & | & | & | & | \\
| & | & | & | & | & | \\
0 & | & | & | & | & |
\end{array}
\]

\[ P = \text{mooore} \]
\[ T = \text{boyer moore} \]

\[
\begin{array}{cccccc}
| & | & | & | & | & | \\
| & | & | & | & | & |
\end{array}
\]
Bad character examples

\[
P = \text{aldo} \\
T = \text{where is waldo}
\]

\[
\begin{array}{cccccccc}
\text{0} & & & & & & & \\
\text{0} & & & & & & \text{l} & \text{d} & \text{o} \\
\end{array}
\]

\[
P = \text{moore} \\
T = \text{boyer moore}
\]

\[
\begin{array}{cccccccc}
\text{0} & & & & & & & \\
\text{l} & & & & & & \text{r} & \text{o} & \text{e} \\
\end{array}
\]
### Bad character examples

\[
P = \text{aldo}
\]
\[
T = \text{where is waldo}
\]

\[
\begin{array}{cccccc}
\text{a} & \text{l} & \text{d} & \text{o} & \text{w} & \text{h} \\
\text{e} & \text{r} & \text{e} & \text{i} & \text{s} & \text{w} \\
\text{a} & \text{l} & \text{d} & \text{o} & \text{a} & \text{l} \\
\end{array}
\]

\[
P = \text{moore}
\]
\[
T = \text{boyer moore}
\]

\[
\begin{array}{cccccc}
\text{m} & \text{o} & \text{o} & \text{r} & \text{e} & \text{w} \\
\text{o} & \text{y} & \text{e} & \text{r} & \text{m} & \text{o} \\
\text{o} & \text{o} & \text{r} & \text{e} & \text{e} & \text{r} \\
\end{array}
\]
Bad character examples

\[ P = \text{aldo} \]
\[ T = \text{where is waldo} \]

\[
\begin{array}{cccccccc}
\text{a} & \text{l} & \text{d} & \text{o} & \text{w} & \text{h} & \text{e} & \text{r} & \text{e} & \text{i}\text{s} & \text{w} & \text{a} & \text{l} & \text{d} & \text{o} \\
\hline
0 & & & & & & & & & & & & & \\
0 & & & & & & & & & & & & & \\
& & & & & & & & & & & & &\\
0 & & & & & & & & & al & d & o \\
\end{array}
\]

\[ \rightsquigarrow 6 \text{ characters not looked at} \]

\[ P = \text{moore} \]
\[ T = \text{boyer moore} \]

\[
\begin{array}{cccccccc}
\text{m} & \text{o} & \text{o} & \text{r} & \text{e} & \text{b} & \text{o} & \text{y} & \text{e} & \text{r} & \text{m} & \text{o} & \text{o} & \text{r} & \text{e} \\
\end{array}
\]
Bad character examples

\[ P = a l d o \]
\[ T = w h e r e i s w a l d o \]

\[ \sim \sim \] 6 characters not looked at

\[ P = m o o r e \]
\[ T = b o y e r m o o r e \]
Bad character examples

\[ P = a l d o \]
\[ T = w h e r e i s w a l d o \]

\[ \sim \sim \sim \] 6 characters not looked at

\[ P = m o o r e \]
\[ T = b o y e r \ underline{m} o o r e \]
Bad character examples

\[
P = \text{aldo} \\
T = \text{where is waldo}
\]

6 characters not looked at

\[
P = \text{moore} \\
T = \text{boyer moore}
\]
Bad character examples

\[ P = a l d o \]
\[ T = \text{where is waldo} \]

\[ \text{\textasciicircum\textasciicircum 6 characters not looked at} \]

\[ P = m o o r e \]
\[ T = \text{boyeyer moore} \]
Bad character examples

$P = a l d o$
$T = w h e r e i s w a l d o$

$\sim 6$ characters not looked at

$P = m o o r e$
$T = b o y e r m o o r e$

$\sim 4$ characters not looked at
Last-Occurrence Function

- Preprocess pattern $P$ and alphabet $\Sigma$
- *last-occurrence function* $\lambda[c]$ defined as
  - the largest index $i$ such that $P[i] = c$ or
  - $-1$ if no such index exists
Last-Occurrence Function

- Preprocess pattern $P$ and alphabet $\Sigma$

- **last-occurrence function** $\lambda[c]$ defined as
  - the largest index $i$ such that $P[i] = c$ or
  - $-1$ if no such index exists

**Example:** $P = \text{moore}$

<table>
<thead>
<tr>
<th></th>
<th>c</th>
<th>m o r e</th>
<th>all others</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda[c]$</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

$P = \text{moore}$

$T = \text{boyer moore}$

$i = 0, \quad j = 4, \quad T[i + j] = r, \quad \lambda[r] = 3$

$\leadsto$ shift by $j - \lambda[T[i + j]] = 1$

$\lambda$ easily computed in $O(m + |\Sigma|)$ time.

- store as array $\lambda[0..\sigma - 1]$. 
Good suffix examples

1. $P = \text{sells} \cup \text{shells}$
Good suffix examples

1. $P = \underline{sells} \underline{shells}$
Good suffix examples

1. $P = \text{sells}_u \text{shells}$
Good suffix examples

1. $P = \text{sells}_w \text{shells}$

2. $P = \text{odetofood}$
Good suffix examples

1. \( P = \text{sells}_u \text{shells} \)

2. \( P = \text{odetofood} \)
Good suffix examples

1. \( P = \text{sells} \_ \text{shells} \)

\[
\begin{array}{cccccccccccc}
    s & h & e & i & l & a & s & e & l & l & s & \_ & s & h & e & l & l & s \\
\end{array}
\]

\( (e) \) (l) (l) (s)

2. \( P = \text{odetofood} \)

\[
\begin{array}{cccccccccccccc}
    i & l & i & k & e & f & o & o & d & f & r & o & m & m & e & x & i & c & o \\
\end{array}
\]

\( (o) \) (d)

\[\text{matched suffix}\]

\[\text{Crucial ingredient: longest suffix of } P[j+1..m-1] \text{ that occurs earlier in } P.\]

\[\text{2 cases (as illustrated above)}\]

1. complete suffix occurs in \( P \) \( \leadsto \) characters left of suffix are not known to match
2. part of suffix occurs at beginning of \( P \)
Good suffix jumps

- Precompute good suffix jumps $\gamma[0..m - 1]$: 
  - For $0 \leq j < m$, $\gamma[j]$ stores shift if search failed at $P[j]$
  - At this point, had $T[i+j+1..i+m-1] = P[j+1..m-1]$, but $T[i] \neq P[j]$
# Good suffix jumps

- Precompute **good suffix jumps** $\gamma[0..m - 1]$:  
  - For $0 \leq j < m$, $\gamma[j]$ stores shift if search failed at $P[j]$  
  - At this point, had $T[i+j+1..i+m-1] = P[j+1..m-1]$, but $T[i] \neq P[j]$

$\therefore \gamma[j]$ is the shift $m - 1 - \ell$ for the largest $\ell$ such that
  - $P[j+1\ldots m-1]$ is a suffix of $P[0\ldots \ell]$ and $P[j] \neq P[\ell-\ell+m+j+1]$

<table>
<thead>
<tr>
<th></th>
<th>h</th>
<th>e</th>
<th>l</th>
<th>l</th>
<th>s</th>
<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\times$</td>
<td></td>
<td></td>
<td>(e)</td>
<td></td>
<td>(l)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- OR-

- $P[0\ldots \ell]$ is a suffix of $P[j+1,\ldots, m-1]$

<table>
<thead>
<tr>
<th></th>
<th>o</th>
<th>f</th>
<th>o</th>
<th>o</th>
<th>d</th>
<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(o)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(d)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---
Good suffix jumps

- Precompute **good suffix jumps** $\gamma[0..m-1]$:
  - For $0 \leq j < m$, $\gamma[j]$ stores shift if search failed at $P[j]$
  - At this point, had $T[i+j+1..i+m-1] = P[j+1..m-1]$, but $T[i] \neq P[j]$

$\sim \gamma[j]$ is the shift $m - 1 - \ell$ for the largest $\ell$ such that
  - $P[j+1...m-1]$ is a suffix of $P[0...\ell]$ and $P[j] \neq P[\ell-m+j+1]$

- OR-
  - $P[0...\ell]$ is a suffix of $P[j+1,..,m-1]$

- Computable (similar to KMP failure function) in $\Theta(m)$ time.

- **Note:** You do not need to know how to find the values $\gamma[j]$ for the exam, but you should be able to find the next guess on examples.
Boyer-Moore algorithm – Discussion

👍 Worst-case running time $\in O(n + m + |\Sigma|)$ if $P$ does not occur in $T$.
(follows from not at all obvious analysis!)

👎 As given, worst-case running time $\Theta(nm)$ if we want to report all occurrences
  ▶ To avoid that, have to keep track of implied matches.
  (tricky because they can be in the “middle” of $P$)
  ⇐▶ Note: KMP reports all matches in $O(n + m)$ without modifications!

👍 On typical English text, Boyer Moore probes only approx. 25% of the characters in $T$!
  ~ Faster than KMP on English text.

👍 requires moderate extra space $\Theta(m + \sigma)$
Clicker Question

How does Boyer-Moore (BM) compare to Knuth-Morris-Pratt (KMP)? Check all correct statements. They refer to the number of symbol comparisons, ignoring preprocessing.

A BP \leq KMP for all inputs
B BP \leq KMP for some inputs
C KMP \leq BM for all inputs
D KMP \leq BM for some inputs
E BM \leq KMP if there are no matches
Clicker Question

How does Boyer-Moore (BM) compare to Knuth-Morris-Pratt (KMP)?
Check all correct statements. They refer to the number of symbol comparisons, ignoring preprocessing.

A $BP \leq KMP$ for all inputs
B $BP \leq KMP$ for some inputs ✓
C $KMP \leq BM$ for all inputs
D $KMP \leq BM$ for some inputs ✓
E $BM \leq KMP$ if there are no matches ✓
4.6 The Rabin-Karp Algorithm
Space – The final frontier

- Knuth-Morris-Pratt has great worst case and real-time guarantees
- Boyer-Moore has great typical behavior
- What else to hope for?
Space – The final frontier

- Knuth-Morris-Pratt has great worst case and real-time guarantees
- Boyer-Moore has great typical behavior
- What else to hope for?

- All require $\Omega(m)$ extra space;
  can be substantial for large patterns!
- Can we avoid that?
Rabin-Karp Fingerprint Algorithm – Idea

Idea: use *hashing* (but without explicit hash tables)

- Precompute & store only *hash* of pattern
- Compute hash for each guess
- If hashes agree, check characterwise
Rabin-Karp Fingerprint Algorithm – Idea

**Idea:** use hashing (but without explicit hash tables)

- Precompute & store only hash of pattern
- Compute hash for each guess
- If hashes agree, check characterwise

**Example:** (treat (sub)strings as decimal numbers)

\[ P = 59265 \]
\[ T = 3141592653589793238 \]

Hash function: \( h(x) = x \mod 97 \)

\[ h(P) = 95. \]
Rabin-Karp Fingerprint Algorithm – Idea

**Idea:** use *hashing* (but without explicit hash tables)

- Precompute & store only hash of pattern
- Compute hash for each guess
- If hashes agree, check characterwise

**Example:** (treat (sub)strings as decimal numbers)

\[ P = 59265 \]
\[ T = 3141592653589793238 \]

Hash function: \( h(x) = x \mod 97 \)
\[ h(P) = 95. \]

\[
\begin{array}{cccccccccccccccccc}
3 & 1 & 4 & 1 & 5 & 9 & 2 & 6 & 5 & 3 & 5 & 8 & 9 & 7 & 9 & 3 & 2 & 3 & 8 \\
\hline
\end{array}
\]

\[ h(31415) = 84 \]
\[ h(14159) = 94 \]
\[ h(41592) = 76 \]
\[ h(15926) = 18 \]
\[ h(59262) = 95 \]
Rabin-Karp Fingerprint Algorithm – First Attempt

```
procedure rabinKarpSimplistic(T[0..n - 1], P[0..m - 1])

M := suitable prime number
hp := computeHash(P[0..m - 1], M)

for i := 0, . . . , n - m do
    ht := computeHash(T[i..i + m - 1], M)
    if ht == hp then
        if T[i..i + m - 1] == P // m comparisons
            then return i
    return NO_MATCH
```

▶ never misses a match since $h(S_1) \neq h(S_2)$ implies $S_1 \neq S_2$

▶ $h(T[k..k+m-1])$ depends on $m$ characters  ⇔ naive computation takes $\Theta(m)$ time

⇔ Running time is $\Theta(mn)$ for search miss . . . can we improve this?
Rabin-Karp Fingerprint Algorithm – Fast Rehash

- **Crucial insight:** We can update hashes in constant time.
  - Use previous hash to compute next hash
  - $O(1)$ time per hash, except first one

for above hash function!
Rabin-Karp Fingerprint Algorithm – Fast Rehash

- **Crucial insight:** We can update hashes in constant time.
  - Use previous hash to compute next hash
  - $O(1)$ time per hash, except first one

Example:
- Pre-compute: $10000 \mod 97 = 9$
- Previous hash: $41592 \mod 97 = 76$
- Next hash: $15926 \mod 97 = ??
Rabin-Karp Fingerprint Algorithm – Fast Rehash

- **Crucial insight:** We can update hashes in constant time.
  - Use previous hash to compute next hash
  - $O(1)$ time per hash, except first one

**Example:**
- Pre-compute: $10000 \mod 97 = 9$
- Previous hash: $41592 \mod 97 = 76$
- Next hash: $15926 \mod 97 = ??$

**Observation:**

\[
15926 \mod 97 = (41592 - (4 \cdot 10000)) \cdot 10 + 6 \mod 97 \\
= (76 - (4 \cdot 9)) \cdot 10 + 6 \mod 97 \\
= 406 \mod 97 = 18
\]
Rabin-Karp Fingerprint Algorithm – Code

- use a convenient radix $R \geq \sigma$ \hspace{1cm} ($R = 10$ in our examples; $R = 2^k$ is faster)

- Choose modulus $M$ at random to be huge prime \hspace{1cm} (randomization against worst-case inputs)

- all numbers remain $\leq 2R^2$ \hspace{1cm} $\sim O(1)$ time arithmetic on word-RAM

```plaintext
procedure rabinKarp(T[0..n - 1], P[0..m - 1], R)
    M := suitable prime number
    hP := computeHash(P[0..m - 1], M)
    hT := computeHash(T[0..m - 1], M)
    s := $R^{m-1}$ mod $M$
    for $i := 0, \ldots, n - m$ do
        if $h_T = h_P$ then
            if $T[i..i + m - 1] = P$
                return $i$
        if $i < n - m$ then
            $h_T := ((h_T - T[i] \cdot s) \cdot R + T[i + m]) \mod M$
    return NO_MATCH
```

Rabin-Karp – Discussion

👍 Expected running time is $O(m + n)$

👎 $\Theta(mn)$ worst-case;
   but this is very unlikely

👍 Extends to 2D patterns and other generalizations

👍 Only constant extra space!
Clicker Question

Suppose we apply only the hashing part of Rabin-Karp (drop the check if $T[i..i+m] = P$, and only return $i$). Check all correct statements about the resulting algorithm.

- A The algorithm can miss occurrences of $P$ in $T$ (false negatives).
- B The algorithm can report positions that are not occurrences (false positives).
- C The running time is $\Theta(nm)$ in the worst case.
- D The running time is $\Theta(n + m)$ in the worst case.
- E The running time is $\Theta(n)$ in the worst case.

[Link: pingo.upb.de/622222]
Suppose we apply only the hashing part of Rabin-Karp (drop the check if $T[i..i + m] = P$, and only return $i$). Check all correct statements about the resulting algorithm.

A. The algorithm can miss occurrences of $P$ in $T$ (false negatives).
B. The algorithm can report positions that are not occurrences (false positives). ✓
C. The running time is $\Theta(nm)$ in the worst case.
D. The running time is $\Theta(n + m)$ in the worst case. ✓
E. The running time is $\Theta(n)$ in the worst case.
# String Matching Conclusion

<table>
<thead>
<tr>
<th></th>
<th>Brute-Force</th>
<th>DFA</th>
<th>KMP</th>
<th>BM</th>
<th>RK</th>
<th>Suffix trees*</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Preproc. time</strong></td>
<td>—</td>
<td>$O(m</td>
<td>\Sigma</td>
<td>)$</td>
<td>$O(m)$</td>
<td>$O(m + \sigma)$</td>
</tr>
<tr>
<td><strong>Search time</strong></td>
<td>$O(nm)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$ (often better)</td>
<td>$O(n + m)$ (expected)</td>
<td>$O(m)$</td>
</tr>
<tr>
<td><strong>Extra space</strong></td>
<td>—</td>
<td>$O(m</td>
<td>\Sigma</td>
<td>)$</td>
<td>$O(m)$</td>
<td>$O(m + \sigma)$</td>
</tr>
</tbody>
</table>

* (see Unit 6)