Outline

1. String Matching
   - Introduction
   - Karp-Rabin Algorithm
   - String Matching with Finite Automata
   - Knuth-Morris-Pratt algorithm
   - Boyer-Moore Algorithm
   - Suffix Trees
   - Suffix Arrays
   - Conclusion
Outline

1 String Matching
   • Introduction
     • Karp-Rabin Algorithm
     • String Matching with Finite Automata
     • Knuth-Morris-Pratt algorithm
     • Boyer-Moore Algorithm
     • Suffix Trees
     • Suffix Arrays
   • Conclusion
Pattern Matching Definition [1]

- Search for a string (pattern) in a large body of text
- $T[0..n-1]$ – The **text** (or **haystack**) being searched within
- $P[0..m-1]$ – The **pattern** (or **needle**) being searched for
- Strings over **alphabet** $\Sigma$
- Return smallest $i$ such that

\[ P[j] = T[i+j] \quad \text{for} \quad 0 \leq j \leq m-1 \]

- This is the first **occurrence** of $P$ in $T$
- If $P$ does not **occur** in $T$, return FAIL
- Applications:
  - Information Retrieval (text editors, search engines)
  - Bioinformatics
  - Data Mining
Pattern Matching Definition [2]

Example:

- $T = \text{“Where is he?”}
- $P_1 = \text{“he”}
- $P_2 = \text{“who”}

Definitions:

- **Substring** $T[i..j]$ $0 \leq i \leq j < n$: a string of length $j - i + 1$ which consists of characters $T[i], \ldots, T[j]$ in order
- A **prefix** of $T$: a substring $T[0..i]$ of $T$ for some $0 \leq i < n$
- A **suffix** of $T$: a substring $T[i..n - 1]$ of $T$ for some $0 \leq i \leq n - 1
General Idea of Algorithms

Pattern matching algorithms consist of guesses and checks:

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

- A check of a guess is a single position $j$ with $0 \leq j < m$ where we compare $T[i + j]$ to $P[j]$. We must perform $m$ checks of a single correct guess, but may make (many) fewer checks of an incorrect guess.

We will diagram a single run of any pattern matching algorithm by a matrix of checks, where each row represents a single guess.
Brute-force Algorithm

**Idea:** Check every possible guess.

\[
\text{BruteForce::patternMatching}(T[0..n-1], P[0..m-1])
\]

*\(T\): String of length \(n\) (text), *\(P\): String of length \(m\) (pattern)

1. \(\text{for } i \leftarrow 0 \text{ to } n - m \text{ do}\)
2. \(\text{if strcmp}(T[i..i+m-1], P) = 0\)
3. \(\text{return} \) “found at guess \(i\)”
4. \(\text{return} \) FAIL

**Note:** \(\text{strcmp}\) takes \(\Theta(m)\) time.

\[
\text{strcmp}(T[i..i+m-1], P[0..m-1])
\]

1. \(\text{for } j \leftarrow 0 \text{ to } m - 1 \text{ do}\)
2. \(\text{if } T[i+j] \text{ is before } P[j] \text{ in } \Sigma \text{ then return } -1\)
3. \(\text{if } T[i+j] \text{ is after } P[j] \text{ in } \Sigma \text{ then return } 1\)
4. \(\text{return} 0\)
Brute-Force Example

- Example: $T = \text{abbbababbab}$, $P = \text{abba}$

- What is the worst possible input?
  $P = a^{m-1}b$, $T = a^n$

- Worst case performance $\Theta((n - m + 1)m)$

- This is $\Theta(mn)$ e.g. if $m = n/2$. 
How to improve?

More sophisticated algorithms

- Do extra **preprocessing** on the pattern \( P \)
  - Karp-Rabin
  - Boyer-Moore
  - Deterministic finite automata (DFA), KMP
  - We **eliminate guesses** based on completed matches and mismatches.

- Do extra **preprocessing** on the text \( T \)
  - **Suffix-trees**
  - We **create a data structure** to find matches easily.
Outline

1. String Matching
   - Introduction
   - Karp-Rabin Algorithm
   - String Matching with Finite Automata
   - Knuth-Morris-Pratt algorithm
   - Boyer-Moore Algorithm
   - Suffix Trees
   - Suffix Arrays
   - Conclusion
Karp-Rabin Fingerprint Algorithm – Idea

**Idea:** use hashing to eliminate guesses

- Compute hash function for each guess, compare with pattern hash
- If values are unequal, then the guess cannot be an occurrence

**Example:** \( P = 5 \ 9 \ 2 \ 6 \ 5, \quad T = 3 \ 1 \ 4 \ 1 \ 5 \ 9 \ 2 \ 6 \ 5 \ 3 \ 5 \)

  ▶ Use standard hash-function: flattening + modular (radix \( R = 10 \)):

\[
h(x_0 \ldots x_4) = (x_0 x_1 x_2 x_3 x_4)_{10} \mod 97
\]

  ▶ \( h(P) = 59265 \mod 97 = 95. \)
Karp-Rabin Fingerprint Algorithm – First Attempt

Karp-Rabin-Simple::patternMatching($T, P$)
1. $h_P \leftarrow h(P[0..m-1])$
2. for $i \leftarrow 0$ to $n - m$
3. $h_T \leftarrow h(T[i..i+m-1])$
4. if $h_T = h_P$
5. if strcmp($T[i..i+m-1], P$) = 0
6. return “found at guess i”
7. return FAIL

- Never misses a match: $h(T[i..i+m-1]) \neq h(P) \Rightarrow$ guess $i$ is not $P$
- $h(T[i..i+m-1])$ depends on $m$ characters, so naive computation takes $\Theta(m)$ time per guess
- Running time is $\Theta(mn)$ if $P$ not in $T$ (how can we improve this?)
Karp-Rabin Fingerprint Algorithm – Fast Update

The initial hashes are called **fingerprints**.

**Crucial insight:** We can update these fingerprints 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: $15926 \mod 97 = ??$

**Observe:** $15926 = (41592 - 4 \cdot 10000) \cdot 10 + 6$

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

<table>
<thead>
<tr>
<th>Karp-Rabin-RollingHash::patternMatching$(T, P)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. $M \leftarrow$ suitable prime number</td>
</tr>
<tr>
<td>2. $h_P \leftarrow h(P[0..m-1])$</td>
</tr>
<tr>
<td>3. $h_T \leftarrow h(T[0..m-1])$</td>
</tr>
<tr>
<td>4. $s \leftarrow 10^{m-1} \mod M$</td>
</tr>
<tr>
<td>5. <strong>for</strong> $i \leftarrow 0$ to $n - m$</td>
</tr>
<tr>
<td>6. <strong>if</strong> $h_T = h_P$</td>
</tr>
<tr>
<td>7. <strong>if</strong> strcmp$(T[i..i+m-1], P) = 0$</td>
</tr>
<tr>
<td>8. <strong>return</strong> “found at guess $i$”</td>
</tr>
<tr>
<td>9. <strong>if</strong> $i &lt; n - m$ // compute hash-value for next guess</td>
</tr>
<tr>
<td>10. $h_T \leftarrow ((h_T - T[i] \cdot s) \cdot 10 + T[i+m]) \mod M$</td>
</tr>
<tr>
<td>11. <strong>return</strong> “FAIL”</td>
</tr>
</tbody>
</table>

- Choose “table size” $M$ at **random** to be huge prime
- Expected running time is $O(m + n)$
- $\Theta(mn)$ worst-case, but this is (unbelievably) unlikely
Outline

1 String Matching
   • Introduction
   • Karp-Rabin Algorithm
   • String Matching with Finite Automata
   • Knuth-Morris-Pratt algorithm
   • Boyer-Moore Algorithm
   • Suffix Trees
   • Suffix Arrays
   • Conclusion
String Matching with Finite Automata

**Example:** Automaton for the pattern \( P = \text{ababaca} \)

You should be familiar with:
- finite automaton, DFA, NFA, converting NFA to DFA
- transition function \( \delta \), states \( Q \), accepting states \( F \)

- The above finite automation is an **NFA**
- State \( q \) expresses “we have seen \( P[0..q-1] \)”
  - NFA accepts \( T \) if and only if \( T \) contains ababaca
  - But evaluating NFAs is very slow.
String matching with DFA

Can show: There exists an equivalent small DFA.

- Easy to test whether $P$ is in $T$.
- But how do we find the arcs?
- We will not give the details of this since there is an even better automaton.
Outline

1. String Matching
   - Introduction
   - Karp-Rabin Algorithm
   - String Matching with Finite Automata
   - Knuth-Morris-Pratt algorithm
   - Boyer-Moore Algorithm
   - Suffix Trees
   - Suffix Arrays
   - Conclusion
Use a new type of transition $\times$ ("failure"): 
- Use this transition only if no other fits.
- Does **not** consume a character.
- With these rules, computations of the automaton are deterministic. (But it is formally not a valid DFA.)

Can store **failure-function** in an array $F[0..m-1]$ 
- The failure arc from state $j$ leads to $F[j-1]$

Given the failure-array, we can easily test whether $P$ is in $T$: Automaton accepts $T$ if and only if $T$ contains ababaca
Knuth-Morris-Pratt Algorithm

KMP::patternMatching(T, P)
1. \( F \leftarrow \text{failureArray}(P) \)
2. \( i \leftarrow 0 \) // current character of \( T \) to parse
3. \( j \leftarrow 0 \) // current state: we have seen \( P[0..j-1] \)
4. \( \text{while } i < n \text{ do} \)
5. \( \quad \text{if } P[j] = T[i] \)
6. \( \quad \quad \text{if } j = m - 1 \)
7. \( \quad \quad \quad \text{return} \) “found at guess \( i - m + 1 \)”
8. \( \quad \quad \text{else} \)
9. \( \quad \quad \quad i \leftarrow i + 1 \)
10. \( \quad \quad \quad j \leftarrow j + 1 \)
11. \( \quad \text{else} \) // i.e. \( P[j] \neq T[i] \)
12. \( \quad \quad \text{if } j > 0 \)
13. \( \quad \quad \quad j \leftarrow F[j - 1] \)
14. \( \quad \text{else} \)
15. \( \quad \quad i \leftarrow i + 1 \)
16. \( \text{return} \) FAIL
String matching with KMP – Example

Example: \( T = \text{ababababaca}, \ P = \text{ababaca} \)

\[
\begin{array}{cccccccccccccccc}
\Sigma - a & a & b & a & b & a & b & b & c & a & b & a & b & a & b & a & c & a \\
& a & b & (a) & b & (a) & (a) & b & (a) & (a) & (b) & (b) & (b) & (b) & (b) & (b) & (b) & (b) \\
\end{array}
\]

\( q: \) 1 2 3 4 5 3, 4 2, 0 0 1 2 3 4 5 6 7

(after reading this character)
String matching with KMP – Failure-function

Assume we reach state \( j+1 \) and now have mismatch.

| \( T: \) | ...matched \( P[0..j] \) | \( \times \) |
| current guess | \( ....P[0..j]...... \) |

- Can eliminate “shift by 1” if \( P[1..j] \neq P[0..j-1] \).
- Can eliminate “shift by 2” if \( P[1..j] \) does not end with \( P[0..j-2] \).
- Generally eliminate guess if that prefix of \( P \) is not a suffix of \( P[1..j] \).
- So want longest prefix \( P[0..\ell-1] \) that is a suffix of \( P[1..j] \).
- The \( \ell \) characters of this prefix are matched, so go to state \( \ell \).

\[
F[j] = \text{head of failure-arc from state } j+1 = \text{length of the longest prefix of } P \text{ that is a suffix of } P[1..j].
\]
KMP Failure Array – Example

$F[j]$ is the length of the longest prefix of $P$ that is a suffix of $P[1..j]$.

Consider $P = ababaca$

<table>
<thead>
<tr>
<th>$j$</th>
<th>$P[1..j]$</th>
<th>Prefixes of $P$</th>
<th>longest</th>
<th>$F[j]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\Lambda$</td>
<td>$\Lambda, a, ab, aba, abab, ababa, \ldots$</td>
<td>$\Lambda$</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>$b$</td>
<td>$\Lambda, a, ab, aba, abab, ababa, \ldots$</td>
<td>$\Lambda$</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>$ba$</td>
<td>$\Lambda, a, ab, aba, abab, ababa, \ldots$</td>
<td>$a$</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>$bab$</td>
<td>$\Lambda, a, ab, aba, abab, ababa, \ldots$</td>
<td>$ab$</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>$baba$</td>
<td>$\Lambda, a, ab, aba, abab, ababa, \ldots$</td>
<td>$aba$</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>$babac$</td>
<td>$\Lambda, a, ab, aba, abab, ababa, \ldots$</td>
<td>$\Lambda$</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>$babaca$</td>
<td>$\Lambda, a, ab, aba, abab, ababa, \ldots$</td>
<td>$a$</td>
<td>1</td>
</tr>
</tbody>
</table>

This can clearly be computed in $O(m^3)$ time, but we can do better!
Computing the Failure Array

\[ \text{KMP::failureArray}(P) \]
\[ P: \text{String of length } m \text{ (pattern)} \]
1. \[ F[0] \leftarrow 0 \]
2. \[ j \leftarrow 1 \] // index within parsed text
3. \[ \ell \leftarrow 0 \] // reached state
4. \[ \textbf{while } j < m \textbf{ do} \]
5. \[ \quad \textbf{if } P[j] = P[\ell] \]
6. \[ \quad \ell \leftarrow \ell + 1 \]
7. \[ \quad F[j] \leftarrow \ell \]
8. \[ \quad j \leftarrow j + 1 \]
9. \[ \quad \textbf{else if } \ell > 0 \]
10. \[ \quad \ell \leftarrow F[\ell - 1] \]
11. \[ \quad \textbf{else} \]
12. \[ \quad F[j] \leftarrow 0 \]
13. \[ \quad j \leftarrow j + 1 \]

**Correctness-idea:** \( F[j] \) is defined via pattern matching of \( P \) in \( P[1..j] \). So KMP uses itself! Already-built parts of \( F[\cdot] \) are used to expand it.
KMP – Runtime

failureArray

- Consider how $2j - \ell$ changes in each iteration of the while loop
  - $j$ and $\ell$ both increase by 1 $\Rightarrow$ $2j - \ell$ increases  –OR–
  - $\ell$ decreases ($F[\ell - 1] < \ell$) $\Rightarrow$ $2j - \ell$ increases  –OR–
  - $j$ increases $\Rightarrow$ $2j - \ell$ increases

- Initially $2j - \ell \geq 0$, at the end $2j - \ell \leq 2m$
- So no more than $2m$ iterations of the while loop.
- Running time: $\Theta(m)$

KMP main function

- failureArray can be computed in $\Theta(m)$ time
- Same analysis gives at most $2n$ iterations of the while loop since $2i - j \leq 2n$.
- Running time KMP altogether: $\Theta(n + m)$
Outline

1 String Matching
   • Introduction
   • Karp-Rabin Algorithm
   • String Matching with Finite Automata
   • Knuth-Morris-Pratt algorithm
   • Boyer-Moore Algorithm
     • Suffix Trees
     • Suffix Arrays
   • Conclusion
Boyer-Moore Algorithm

Fastest pattern matching on English text.

Important components:

- **Reverse-order searching**: Compare $P$ with a guess moving backwards

When a mismatch occurs, choose the better of the following two options:

- **Bad character jumps**: Eliminate guesses based on mismatched characters of $T$.

- **Good suffix jumps**: Eliminate guesses based on matched suffix of $P$. 
Forward-searching vs. reverse-searching

\(P\): aldo
\(T\): whereiswaldo

Forward-searching:

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

- \(w\) does not occur in \(P\).
  \(\Rightarrow\) shift pattern past \(w\).
- \(h\) does not occur in \(P\).
  \(\Rightarrow\) shift pattern past \(h\).

With forward-searching, no guesses are ruled out.

Reverse-searching:

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

- \(r\) does not occur in \(P\).
  \(\Rightarrow\) shift pattern past \(r\).
- \(w\) does not occur in \(P\).
  \(\Rightarrow\) shift pattern past \(w\).

This \textit{bad character heuristic} works well with reverse-searching.
Bad character heuristic details

\[ P : \text{paper} \]

\[ T : \text{feedallpoorparrots} \]

- Mismatched character in the text is \texttt{a}
- Shift the guess until \texttt{a} in \textit{P} aligns with \texttt{a} in \textit{T}
  - All skipped guessed are impossible since they do not match \texttt{a}
- Shift the guess until \texttt{last p} in \textit{P} aligns with \texttt{p} in \textit{T}
  - Use “last” since we cannot rule out this guess.
- As before, shift completely past \texttt{o} since \texttt{o} is not in \textit{P}.
- Finding \texttt{r} does not help ⇒ shift by one unit.
  - Here the other strategy will do better.
Last-Occurrence Array

- Build the **last-occurrence array** $L$ mapping $\Sigma$ to integers
- $L[c]$ is the largest index $i$ such that $P[i] = c$
- We will see soon: If $c$ is not in $P$, then we should set $L[c] = -1$

Pattern: paper

<table>
<thead>
<tr>
<th>char</th>
<th>p</th>
<th>a</th>
<th>e</th>
<th>r</th>
<th>all others</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L[\cdot]$</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>–1</td>
</tr>
</tbody>
</table>

We can build this in time $O(m + |\Sigma|)$ with simple for-loop

```
BoyerMoore::lastOccurrenceArray(P[0..m-1])
1. initialize array $L$ indexed by $\Sigma$ with all $-1$
2. for $j \leftarrow 0$ to $m-1$ do $L[P[j]] \leftarrow j$
3. return $L$
```

- But how should we do the update?
Bad character heuristic formula

We will always compare $T[i]$ and $P[j]$. How to update at a mismatch?

“Good” case: $L[c] < j$, so $c$ is left of $P[j]$.

- $j^{\text{new}} = m - 1$ (we re-start the search from the right end)
- $i^{\text{new}} = \text{corresponding index in } T$. What is it?
  - $\Delta_1 = \text{amount that we should shift} = j^{\text{old}} - L[c]$
  - $\Delta_2 = \text{how much we had compared} = (m - 1) - j^{\text{old}}$
  - $i^{\text{new}} = i^{\text{old}} + \Delta_2 + \Delta_1 = i^{\text{old}} + (m - 1) - L[c]$
Bad character heuristic formula

**Bad case 1:** \( c \) does not occur in \( P \).

We want to shift past \( T[i_{\text{old}}] \), so need \( i_{\text{new}} = i_{\text{old}} + m \)

What value of \( L[c] \) would achieve this automatically?

- formula was \( i_{\text{new}} = i_{\text{old}} + (m-1) - L[c] \)
- \( \Rightarrow \) set \( L[c] := -1 \)
Bad character heuristic formula

**Bad case 2:** $L[c] > j$, so $c$ is right of $P[j]$.

$$i^{\text{new}} = i^{\text{old}} + \Delta_2 + \Delta_1 = i^{\text{old}} + 1 + (m-1) - j^{\text{old}}$$

**Unified formula for all cases:**

$$i^{\text{new}} = i^{\text{old}} + (m-1) - \min\{L[c], j^{\text{old}} - 1\}$$
Boyer-Moore::patternMatching(T, P)
1. \( L \leftarrow \text{lastOccurrenceArray}(P) \)
2. \( S \leftarrow \text{good suffix array computed from } P \)
3. \( i \leftarrow m - 1, \quad j \leftarrow m - 1 \)
4. while \( i < n \) and \( j \geq 0 \) do
5. \( \text{if } T[i] = P[j] \)
6. \( i \leftarrow i - 1 \)
7. \( j \leftarrow j - 1 \)
8. \( \text{else} \)
9. \( i \leftarrow i + m - 1 - \min\{L[T[i]], j - 1, S[j]\} \)
10. \( j \leftarrow m - 1 \)
11. \( \text{if } j = -1 \text{ return } \text{“found at } T[i+1..i+m]” \)
12. \( \text{else return } \text{FAIL} \)

\( S \) will be explained below.

**Can show:** ‘\( j - 1 \)’ is not needed in line 9 since \( \min\{L[T[i]], S[j]\} \leq j - 1 \)
Good Suffix Heuristic

$S[j]$ expresses

“since $P[j+1..m-1]$ was matched, how much should we shift?”

| $P$: onobbobo |
| $T$: onoooboo o i b b o u n d a r y |

- Do smallest shift so that $obo$ fits in the new guess.
- Doing examples is easy, but the formula is complicated (no details)
- $S[.]$ computable (similar to KMP failure function) in $\Theta(m)$ time.

Summary:

- Boyer-Moore performs very well (even without good suffix heuristic).
- On typical English text Boyer-Moore looks at only $\approx 25\%$ of $T$
- Worst-case run-time for is $O(mn)$, but in practice much faster.
  [There are ways to ensure $O(n)$ run-time. No details.]
1 String Matching

- Introduction
- Karp-Rabin Algorithm
- String Matching with Finite Automata
- Knuth-Morris-Pratt algorithm
- Boyer-Moore Algorithm
- Suffix Trees
- Suffix Arrays
- Conclusion
What if we want to search for many patterns \( P \) within the same fixed text \( T \)?

Idea: Preprocess the text \( T \) rather than the pattern \( P \)

Observation: \( P \) is a substring of \( T \) if and only if \( P \) is a prefix of some suffix of \( T \).

So want to store all suffixes of \( T \) in a trie.

To save space:
- Use a compressed trie.
- Store suffixes implicitly via indices into \( T \).

This is called a suffix tree.
Trie of suffixes: Example

$T = \text{bananaban}$ has suffixes

$\{\text{bananaban, ananaban, nanaban, anaban, naban, aban, ban, an, n, } \Lambda\}$
Tries of suffixes

Store suffixes via indices:

\[ T = \begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\text{b} & \text{a} & \text{n} & \text{a} & \text{n} & \text{a} & \text{b} & \text{a} & \text{n} & \$ \\
\end{array} \]
Suffix tree

Suffix tree: Compressed trie of suffixes

\[ T = \text{banana\$} \]

\[
\begin{array}{c}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\text{\$} & \text{b} & \text{a} & \text{n} & \text{a} & \text{n} & \text{a} & \text{b} & \text{a} & \text{n} \\
\end{array}
\]
More on Suffix Trees

Building:

- Text $T$ has $n$ characters and $n + 1$ suffixes
- We can build the suffix tree by inserting each suffix of $T$ into a compressed trie. This takes time $\Theta(n^2|\Sigma|)$.
- There is a way to build a suffix tree of $T$ in $\Theta(n|\Sigma|)$ time. This is quite complicated and beyond the scope of the course.

Pattern Matching:

- Essentially search for $P$ in compressed trie.
  Some changes are needed, since $P$ may only be prefix of stored word.
- Run-time: $O(|\Sigma|m)$.

Summary: Theoretically good, but construction is slow or complicated, and lots of space-overhead $\Rightarrow$ rarely used.
Outline

1 String Matching
   • Introduction
   • Karp-Rabin Algorithm
   • String Matching with Finite Automata
   • Knuth-Morris-Pratt algorithm
   • Boyer-Moore Algorithm
   • Suffix Trees
   • Suffix Arrays
   • Conclusion
Suffix Arrays

- Relatively recent development (popularized in the 1990s)
- Sacrifice some performance for simplicity:
  - Slightly slower (by a log-factor) than suffix trees.
  - Much easier to build.
  - Much simpler pattern matching.
  - Very little space; only one array.

Idea:

- Store suffixes implicitly (by storing start-indices)
- Store *sorting permutation* of the suffixes of $T$. 

Biedl, Schost, Veksler  (SCS, UW)
Suffix Array Example

Text $T$: \[\text{banana}\text{banana}^\$$

<table>
<thead>
<tr>
<th>$i$</th>
<th>suffix $T[i..n-1]$</th>
<th>$j$</th>
<th>$A^s[j]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>bananaban$</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>ananaban$</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>nanaban$</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>anaban$</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>naban$</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>aban$</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>ban$</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>an$</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>n$</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>$</td>
<td>9</td>
<td>2</td>
</tr>
</tbody>
</table>

sort lexicographically

Suffix array: \[9\;5\;7\;3\;1\;6\;0\;8\;4\;2\]
Suffix Array Construction

- Easy to construct using **MSD-Radix-Sort**.
  - Fast in practice; suffixes are unlikely to share many leading characters.
  - But worst-case run-time is $\Theta(n^2)$
    - $n$ rounds of recursions (have $n$ chars)
    - Each round takes $\Theta(n)$ time (bucket-sort)

- **Idea**: We do not need $n$ rounds!

  - Consider sub-array after one round.
  - These have same leading char. Ties are broken by rest of words.
  - But rest of words are also suffixes $\Rightarrow$ sorted elsewhere
  - We can double length of sorted part every round.

  - $O(\log n)$ rounds enough $\Rightarrow O(n \log n)$ **run-time**

- Construction-algorithm: MSD-radix-sort plus some bookkeeping
  - needs only one extra array
  - easy to implement

- You do not need to know details.
Pattern matching in suffix arrays

- Suffix array stores suffixes (implicitly) in sorted order.
- **Idea**: apply binary search!

\[ P = \text{ban:} \]

<table>
<thead>
<tr>
<th>( \ell \rightarrow )</th>
<th>( j )</th>
<th>( A^s[j] )</th>
<th>( T[A^s[j]..n-1] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>9</td>
<td>$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>aban$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>an$</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>anaban$</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>ananaban$</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>ban$</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>bananaban$</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>n$</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>naban$</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>nanaban$</td>
<td></td>
</tr>
</tbody>
</table>

- \( O(\log n) \) comparisons.
- Each comparison is `strcmp(P, T[A^s[\nu]..A^s[\nu + m - 1]])`
- \( O(m) \) time per comparison => **run-time** \( O(m \log n) \)
Pattern matching in suffix arrays

**SuffixArray-search**($A^s[0...n−1], P[0..m−1]$)

$A^s$: suffix array of $T$, $P$: pattern

1. $\ell \leftarrow 0$, $r \leftarrow n−1$
2. while ($\ell < r$)
3. $\nu \leftarrow \lfloor \frac{\ell+r}{2} \rfloor$
4. $i \leftarrow A^s[\nu]$  // Suffix is $T[i..n−1]$
5. $s \leftarrow \text{strcmp}(T[i..i+m−1], P)$
6. // Assuming strcmp handles “out of bounds” suitably
7. if ($s < 0$) do $\ell \leftarrow \nu + 1$
8. else if ($s > 0$) do $r \leftarrow \nu − 1$
9. else return “found at guess $T[i..i+m−1]$”
10. if strcmp($T, P, A^s[\ell], A^s[\ell]+m−1) = 0$
11. return “found at guess $T[\ell..\ell+m−1]”$
12. return FAIL
Outline

1. String Matching
   - Introduction
   - Karp-Rabin Algorithm
   - String Matching with Finite Automata
   - Knuth-Morris-Pratt algorithm
   - Boyer-Moore Algorithm
   - Suffix Trees
   - Suffix Arrays
   - Conclusion
### String Matching Conclusion

<table>
<thead>
<tr>
<th></th>
<th>Brute-Force</th>
<th>Karp-Rabin</th>
<th>DFA</th>
<th>Knuth-Morris-Pratt</th>
<th>Boyer-Moore</th>
<th>Suffix Tree</th>
<th>Suffix Array</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Preproc.</strong></td>
<td>—</td>
<td>$O(m)$</td>
<td>$O(m</td>
<td>\Sigma</td>
<td>)$</td>
<td>$O(m)$</td>
<td>$O(m+</td>
</tr>
<tr>
<td><strong>Search time</strong></td>
<td>$O(nm)$</td>
<td>$O(n+m)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(m)$</td>
<td>$O(m \log n)$</td>
</tr>
<tr>
<td></td>
<td>expected</td>
<td></td>
<td></td>
<td></td>
<td>or better</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Extra space</strong></td>
<td>—</td>
<td>$O(1)$</td>
<td>$O(m</td>
<td>\Sigma</td>
<td>)$</td>
<td>$O(m)$</td>
<td>$O(m+</td>
</tr>
</tbody>
</table>

- Our algorithms stopped once they have found one occurrence.
- Most of them can be adapted to find *all* occurrences within the same worst-case run-time.