Module 9: Tries and String Matching

CS 240 - Data Structures and Data Management

Mark Petrick
Based on lecture notes by many previous cs240 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Fall 2017
Pattern Matching

- 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 the first \( i \) 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

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** 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.

```
BruteforcePM(T[0..n − 1], P[0..m − 1])
T: String of length n (text), P: String of length m (pattern)
1.    for i ← 0 to n − m do
2.        match ← true
3.        j ← 0
4.            while j < m and match do
5.                if T[i + j] = P[j] then
6.                    j ← j + 1
7.                else
8.                    match ← false
9.            if match then
10.                return i
11.        return FAIL
```
Example: $T = abbbabababbab$, $P = abba$

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

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

$m \leq n/2 \Rightarrow \Theta(mn)$
Pattern Matching

More sophisticated algorithms

- Deterministic finite automata (DFA)
- KMP, Boyer-Moore and Rabin-Karp
- Do extra **preprocessing** on the pattern $P$
- We **eliminate guesses** based on completed matches and mismatches.
There is a string-matching automaton for every pattern $P$. It is constructed from the pattern in a preprocessing step before it can be used to search the text string.

**Example:** Automaton for the pattern $P = \text{ababaca}$
String matching with finite automata

Let $P$ the pattern to search, of length $m$. Then

- the states of the automaton are $0, \ldots, m$
- the transition function $\delta$ of the automaton is defined as follows, for a state $q$ and a character $c$ in $\Sigma$:

$$\delta(q, c) = \ell(P[0..q-1]c),$$

where

- $P[0..q-1]c$ is the concatenation of $P[0..q-1]$ and $c$
- for a string $s$, $\ell(s) \in \{0, \ldots, m\}$ is the length of the longest prefix of $P$ that is also a suffix of $s$.

Graphically, this corresponds to

$$q \quad c \quad \delta(q, c)$$
String matching with finite automata

Let $T$ be the text string of length $n$, $P$ the pattern to search of length $m$ and $\delta$ the transition function of a finite automaton for pattern $P$.

FINITE-AUTOMATON-MATCHER($T$, $\delta$, $m$)

$$n \leftarrow \text{length}[T]$$

$$q \leftarrow 0$$

for $i \leftarrow 0$ to $n - 1$ do

$$q \leftarrow \delta(q, T[i])$$

if $q = m$

then print ”Pattern occurs with shift” $i - (m - 1)$

Idea of proof: the state after reading $T[i]$ is $\ell(T[0..i])$. 
String matching with finite automata

- Matching time on a text string of length $n$ is $\Theta(n)$

- This does not include the preprocessing time required to compute the transition function $\delta$. There exists an algorithm with $O(m|\Sigma|)$ preprocessing time.

- Altogether, we can find all occurrences of a length-$m$ pattern in a length-$n$ text over a finite alphabet $\Sigma$ with $O(m|\Sigma|)$ preprocessing time and $\Theta(n)$ matching time.
KMP Algorithm

- Knuth-Morris-Pratt algorithm (1977)
- Compares the pattern to the text in **left-to-right**
- **Shifts** the pattern more **intelligently** than the brute-force algorithm
- When a mismatch occurs, how much can we shift the pattern (reusing knowledge from previous matches)?

\[ T = \begin{array}{ccccccc}
  a & b & c & d & c & a & b & c & ? & ? & ? \\
  a & b & c & d & c & a & b & a & \hline
  & & & & & a & b & c & d & c & a
\end{array} \]

- **KMP Answer**: this depends on the largest prefix of \( P[0..j] \) that is a suffix of \( P[1..j] \)
KMP Failure Array

T:  a b b c a b c d . . .
P:  a b b c a b a a a

what next slide would match with the text?
## KMP Failure Array

<table>
<thead>
<tr>
<th>T:</th>
<th>a</th>
<th>b</th>
<th>b</th>
<th>c</th>
<th>a</th>
<th>b</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>P:</td>
<td>a</td>
<td>b</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>a</td>
<td>b</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td></td>
</tr>
</tbody>
</table>
KMP Failure Array

\[
\begin{align*}
T: & \quad a \ b \ b \ c \ a \ b \ b \ c \ d \ldots \\
P: & \quad a \ b \ b \ c \ a \ b \ a \ a \\
X & \quad a \ b \ b \ c \ a \ b \ a \ a \\
X & \quad a \ b \ b \ c \ a \ b \ a \ a \\
\end{align*}
\]
KMP Failure Array

T: a b b c a b c d ...

P: a b b c a b a a

X

X

X

X

a b b c a b a a

a b b c a b a a

a b b c a b a a

a b b c a b a a
KMP Failure Array

Suppose we have a match up to position $T[i - 1] = P[j - 1]$, but not at the next position.

Define $F[j - 1]$ as the index we will have to check in $P$, after we bring the pattern to its next possible position (previous example: $j = 6$, $F[5] = 2$).

This can be computed by trying all sliding positions until finding the first one matching the text (as in previous example). We can do better:

- any possible sliding position corresponds to a prefix of $P[0..j - 1]$ that is also a strict suffix of it = a suffix of $P[1..j - 1]$
- the next possible sliding position corresponds to the largest such prefix / suffix
- we let $F[j - 1]$ be the length of this prefix / suffix.
KMP Failure Array

Schematically:

\[ T \]
\[ P \]

first mismatch

\[ j \]
\[ i \]

\[ j - 1 \]
KMP Failure Array

Schematically:

\[ T \]

\[ P \]

Next slide that matches with T

next slide that matches with T

\[ j - 1 \]

\[ j \]
KMP Failure Array

Schematically:

\[ T \]

\[ P \]

next slide that matches with \( T \)

\( j - 1 \)

\( j \)
KMP Failure Array

Schematically:

\[ T \]

\[ P \]

\[ F[j - 1] \]

\[ F[j - 1] \]
KMP Failure Array

Schematically:

T

P

Slide

no need to check for matching with T

comparing with T starts from here

$F[j - 1]$
KMP Failure Array

- $F[0] = 0$
- $F[j]$, for $j > 0$, is the length of the largest prefix of $P[0..j]$ that is also a suffix of $P[1..j]$
- Consider $P = \text{abacaba}$

<table>
<thead>
<tr>
<th>$j$</th>
<th>$P[1..j]$</th>
<th>$P$</th>
<th>$F[j]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>—</td>
<td>abacaba</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>b</td>
<td>abacaba</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>ba</td>
<td>abacaba</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>bac</td>
<td>abacaba</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>baca</td>
<td>abacaba</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>ocab</td>
<td>abacaba</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>bacaba</td>
<td>abacaba</td>
<td>3</td>
</tr>
</tbody>
</table>
Computing the Failure Array

**failureArray(P)**

*P*: String of length *m* (pattern)

1. \( F[0] \leftarrow 0 \)
2. \( i \leftarrow 1 \)
3. \( j \leftarrow 0 \)
4. while \( i < m \) do  
5. \hspace{1em} if \( P[i] = P[j] \) then  
6. \hspace{2em} \( F[i] \leftarrow j + 1 \)
7. \hspace{2em} \( i \leftarrow i + 1 \)
8. \hspace{2em} \( j \leftarrow j + 1 \)
9. \hspace{1em} else if \( j > 0 \) then  
10. \hspace{2em} \( j \leftarrow F[j - 1] \)
11. \hspace{1em} else  
12. \hspace{2em} \( F[i] \leftarrow 0 \)
13. \hspace{2em} \( i \leftarrow i + 1 \)
KMP Algorithm

\[ \text{KMP}(T, P), \text{ to return the first match} \]

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

1. \( F \leftarrow \text{failureArray}(P) \)
2. \( i \leftarrow 0 \)
3. \( j \leftarrow 0 \)
4. \( \text{while } i < n \text{ do} \)
5. \( \quad \text{if } T[i] = P[j] \text{ then} \)
6. \( \quad \quad \text{if } j = m - 1 \text{ then} \)
7. \( \quad \quad \quad \text{return } i - j \quad // \text{match} \)
8. \( \quad \quad \text{else} \)
9. \( \quad \quad i \leftarrow i + 1 \)
10. \( \quad j \leftarrow j + 1 \)
11. \( \quad \text{else} \)
12. \( \quad \quad \text{if } j > 0 \text{ then} \)
13. \( \quad \quad \quad j \leftarrow F[j - 1] \)
14. \( \quad \quad \text{else} \)
15. \( \quad \quad i \leftarrow i + 1 \)
16. \( \quad \text{return } -1 \quad // \text{ no match} \)
KMP: Example

\[ P = \text{abacaba} \]

\[ T = \underline{abaxy}abacabbaababacaba \]

<table>
<thead>
<tr>
<th>j</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>( F[j] )</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Exercise: continue with \( T = \underline{abaxy}abacacabbaababacaba \)
KMP: Analysis

failureArray

- At each iteration of the while loop, at least one of the following happens:
  1. $i$ increases by one, or
  2. the index $i - j$ increases by at least one ($F[j - 1] < j$)
- There are no more than $2m$ iterations of the while loop
- Running time: $\Theta(m)$

KMP

- failureArray can be computed in $\Theta(m)$ time
- At each iteration of the while loop, at least one of the following happens:
  1. $i$ increases by one, or
  2. the index $i - j$ increases by at least one ($F[j - 1] < j$)
- There are no more than $2n$ iterations of the while loop
- Running time: $\Theta(n)$
Boyer-Moore Algorithm

Based on three key ideas:

- **Reverse-order searching**: Compare $P$ with a subsequence of $T$ moving backwards

- **Bad character jumps**: When a mismatch occurs at $T[i] = c$
  - If $P$ contains $c$, we can shift $P$ to align the last occurrence of $c$ in $P$ with $T[i]$
  - Otherwise, we can shift $P$ to align $P[0]$ with $T[i + 1]$

- **Good suffix jumps**: If we have already matched a suffix of $P$, then get a mismatch, we can shift $P$ forward to align with the previous occurrence of that suffix (with a mismatch from the suffix we read). If none exists, look for the longest prefix of $P$ that is a suffix of what we read. Similar to failure array in KMP.

- Can skip large parts of $T$
Bad character examples

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

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>o</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>o</td>
</tr>
<tr>
<td></td>
<td>o</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>a</td>
<td>l</td>
<td>d</td>
</tr>
</tbody>
</table>

6 comparisons (checks)

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

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<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>e</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(r)</td>
<td>e</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(m)</td>
<td>o</td>
<td>o</td>
</tr>
</tbody>
</table>

7 comparisons (checks)
Good suffix examples

\[ P = \text{sells}_\cdot \text{shells} \]

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

\[ P = \text{odetofood} \]

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

- Good suffix moves further than bad character for 2nd guess.
- Bad character moves further than good suffix for 3rd guess.
- This is out of range, so pattern not found.
Last-Occurrence Function

- **Preprocess** the pattern $P$ and the alphabet $\Sigma$
- Build the **last-occurrence function** $L$ mapping $\Sigma$ to integers
- $L(c)$ is defined as
  - the largest index $i$ such that $P[i] = c$ or
  - $-1$ if no such index exists
- Example: $\Sigma = \{a, b, c, d\}$, $P = abacab$

$$
\begin{array}{ccccc}
  c & a & b & c & d \\
  L(c) & 4 & 5 & 3 & -1 \\
\end{array}
$$

- The last-occurrence function can be computed in time $O(m + |\Sigma|)$
- In practice, $L$ is stored in a size-$|\Sigma|$ array.
Good Suffix array

- Again, we **preprocess** $P$ to build a table.
- **Suffix skip array** $S$ of size $m$: for $0 \leq i < m$, $S[i]$ is the largest index $j$ such that $P[i+1..m-1] = P[j+1..j+m-1-i]$ and $P[j] \neq P[i]$.
- **Note**: in this calculation, any negative indices are considered to make the given condition true (these correspond to letters that we might not have checked yet).
- Similar to KMP failure array, with an extra condition.
- Computed similarly to KMP failure array in $\Theta(m)$ time.
Good Suffix array

Example: \( P = \text{bonobobo} \)

<table>
<thead>
<tr>
<th></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>
</tr>
</thead>
<tbody>
<tr>
<td>( P[i] )</td>
<td>b</td>
<td>o</td>
<td>n</td>
<td>o</td>
<td>b</td>
<td>o</td>
<td>b</td>
<td>o</td>
</tr>
<tr>
<td>( S[i] )</td>
<td>−6</td>
<td>−5</td>
<td>−4</td>
<td>−3</td>
<td>2</td>
<td>−1</td>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>

- Computed similarly to KMP failure array in \( \Theta(m) \) time.
Boyer-Moore Algorithm

**boyer-moore(T,P)**

1. \( L \leftarrow \text{last occurrence array computed from } P \)
2. \( S \leftarrow \text{good suffix array computed from } P \)
3. \( i \leftarrow m - 1, \quad j \leftarrow m - 1 \)
4. \( \text{while } i < n \text{ and } j \geq 0 \text{ do} \)
5. \( \quad \text{if } T[i] = P[j] \text{ then} \)
6. \( \quad i \leftarrow i - 1 \)
7. \( \quad j \leftarrow j - 1 \)
8. \( \quad \text{else} \)
9. \( \quad i \leftarrow i + m - 1 - \min(L[T[i]], S[j]) \)
10. \( \quad j \leftarrow m - 1 \)
11. \( \quad \text{if } j = -1 \text{ return } i + 1 \)
12. \( \quad \text{else return } \text{FAIL} \)

**Exercise:** Prove that \( i - j \) always increases on lines 9–10.
Boyer-Moore algorithm conclusion

- Worst-case running time $\in O(n + |\Sigma|)$
- This complexity is difficult to prove.
- Worst-case running time $O(nm)$ if we want to report all occurrences
- On typical **English text** the algorithm probes approximately 25% of the characters in $T$
- Faster than KMP in practice on English text.
Rabin-Karp Fingerprint Algorithm

Idea: use hashing

- Compute hash function for each text position
- No explicit hash table: just compare with pattern hash
- If a match of the hash value of the pattern and a text position found, then compares the pattern with the substring by naive approach

Example:
Hash "table" size = 97
Search Pattern $P$: 5 9 2 6 5
Search Text $T$: 3 1 4 1 5 9 2 6 5 3 5 8 9 7 9 3 2 3 8 4 6
Hash function: $h(x) = x \mod 97$ and $h(P) = 95$.
$31415 \mod 97 = 84$
$14159 \mod 97 = 94$
$41592 \mod 97 = 76$
$15926 \mod 97 = 18$
$59265 \mod 97 = 95$
Rabin-Karp Fingerprint Algorithm

Guaranteeing correctness

- Need full compare on hash match to guard against collisions
  - $59265 \mod 97 = 95$
  - $59362 \mod 97 = 95$

Running time

- Hash function depends on $m$ characters
- Running time is $\Theta(mn)$ for search miss (how can we fix this?)
Rabin-Karp Fingerprint Algorithm

The initial hashes are called **fingerprints**. Rabin & Karp discovered a way to update these fingerprints in constant time.

**Idea:**
To go from the hash of a substring in the text string to the next hash value only requires constant time.

- Use previous hash to compute next hash
- $O(1)$ time per hash, except first one
Rabin-Karp Fingerprint Algorithm

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

Observation:

$$15926 \mod 97 = (41592 - (4 \times 10000)) \times 10 + 6$$
$$= (76 - (4 \times 9)) \times 10 + 6$$
$$= 406$$
$$= 18$$
Rabin-Karp Fingerprint Algorithm

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

Main advantage:
- Extends to 2d patterns and other generalizations
Suffix Tries and Suffix Trees

- 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 $T$.

We will call a trie that stores all suffixes of a text $T$ a **suffix trie**, and the compressed suffix trie of $T$ a **suffix tree**.
Suffix Trees

- Build the suffix trie, i.e. the trie containing all the suffixes of the text
- Build the suffix tree by compressing the trie above (like in Patricia trees)
- Store two indexes $l, r$ on each node $v$ (both internal nodes and leaves) where node $v$ corresponds to substring $T[l..r]$
Suffix Trie: Example

$T = \text{bananaban}$

$\{\text{bananaban, ananaban, nanaban, anaban, naban, aban, ban, an, n}\}$
Suffix Tree (compressed suffix trie): Example

$T = \text{bananaban}$

$\{\text{bananaban, ananaban, nanaban, anaban, naban, aban, ban, an, n}\}$
Suffix Trees: Pattern Matching

To search for pattern $P$ of length $m$:

- Similar to Search in compressed trie with the difference that we are looking for a prefix match rather than a complete match.
- If we reach a leaf with a corresponding string length less than $m$, then search is unsuccessful.
- Otherwise, we reach a node $v$ (leaf or internal) with a corresponding string length of at least $m$.
- It only suffices to check the first $m$ characters against the substring of the text between indices of the node, to see if there indeed is a match.
- We can then visit all children of the node to report all matches.
Suffix Tree: Example

\( T = \text{bananaban} \)

\( P = \text{ana} \)

<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>
</tr>
</thead>
<tbody>
<tr>
<td>( T[i] )</td>
<td>( b )</td>
<td>( a )</td>
<td>( n )</td>
<td>( a )</td>
<td>( n )</td>
<td>( a )</td>
<td>( b )</td>
<td>( a )</td>
<td>( n )</td>
<td>( $ )</td>
</tr>
</tbody>
</table>
Suffix Tree: Example

\[ T = \text{bananaban} \]

\[ P = \text{ban} \]
Suffix Tree: Example

$T = \text{bananaban}$

$P = \text{nana}$
Suffix Tree: Example

\( T = \text{bananaban} \)

\( P = \text{bbn} \) not found

\begin{itemize}
  \item \( T[i] \) at each node indicates the last occurrence of \( T[0..i] \) in \( T \).
  \item \( P \) is not found in \( T \).
\end{itemize}
# Pattern 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.:</strong></td>
<td>–</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))</td>
<td>(O(n))</td>
<td>(O(n))  (\text{(often better)})</td>
<td>(\tilde{O}(n + m)) (\text{(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 +</td>
</tr>
</tbody>
</table>