Index Structures in Biological Sequence Analysis
From Simplicity to Complexity and Back

Jens Stoye

AG Genominformatik, Technische Fakultät
Institute of Bioinformatics, Center of Biotechnology

Bielefeld University, Germany
Index structures in biological sequence analysis

1. Introduction

2. Suffix trees

3. Affix trees

4. Suffix arrays

5. The $q$-Gram index

6. Summary and Conclusion
Index structures in biological sequence analysis

1. Introduction
2. Suffix trees
3. Affix trees
4. Suffix arrays
5. The $q$-Gram index
6. Summary and Conclusion
Biological sequence analysis

The data:
- DNA sequences – may be very long, small alphabet \( A, C, G, T \)
- RNA sequences – usually moderately long
- protein sequences – usually short, larger alphabet size

The tasks:
- sequence comparison
- pattern matching
- pattern discovery

The challenges:
- efficient algorithms
- flexible tools
- statistical assessment of significance of results
- visualization
Biological sequence analysis

The data:
- DNA sequences – may be very long, small alphabet $A, C, G, T$
- RNA sequences – usually moderately long
- protein sequences – usually short, larger alphabet size

The tasks:
- sequence comparison
- pattern matching
- pattern discovery

The challenges:
- efficient algorithms
- flexible tools
- statistical assessment of significance of results
- visualization
Some applications

- **Sequence comparison**
  - alignment, multiple alignment
  - similar sequence $\rightarrow$ similar structure $\rightarrow$ similar function

- **Pattern matching**
  - mapping of *expressed sequence tags* (ESTs) on genomic DNA
  - targets of a given miRNA
  - palindromic or other RNA structural patterns
  - known repeats (for further exclusion from analysis)

- **Pattern discovery**
  - unknown promoter binding sites
  - repeats, tandem repeats
  - possible DNA methylation sites
Index structures

The result of **preprocessing** the data for faster search.
Index structures

The result of **preprocessing** the data for faster search.
Index structures

The result of **preprocessing** the data for faster search.

Many applications assume that the text is partitioned into words (natural language, syntactic tags, ...)

Genomic data is not divided into obvious "words"

→ we need indices that allow access to **any substring** of the text
Limitation:

- Most full-text indices allow only simple searches.
Full-text index structures

Limitation:
- Most full-text indices allow only simple searches.

But:
- Simple searches are often the core of more complex methods.
Full-text index structures

Limitation:
- Most full-text indices allow only simple searches.

But:
- Simple searches are often the core of more complex methods.

Example: degenerate repeats

Task: In a given string $S$ of length $n$, find all pairs of occurrences of substrings of length at least $\ell$ that differ by at most $k$ errors.
Finding degenerate repeats

Idea: A repeat of length $\ell$ with $k$ errors contains an exact match of length at least $s := \lfloor \ell/(k + 1) \rfloor$.

Algorithm:
1. Find all exact repeats of length $\geq s$. (Using an index)
2. Extend these by up to $k$ errors.
3. Report matches whenever length $\ell$ is reached.
Finding degenerate repeats

Idea: A repeat of length $\ell$ with $k$ errors contains an exact match of length at least $s := \lfloor \ell/(k + 1) \rfloor$.

Algorithm:
1. Find all exact repeats of length $\geq s$. (Using an index)
2. Extend these by up to $k$ errors.
3. Report matches whenever length $\ell$ is reached.
Finding degenerate repeats

Idea: A repeat of length \( \ell \) with \( k \) errors contains an exact match of length at least \( s := \lfloor \ell/(k + 1) \rfloor \).

Algorithm:

1. Find all exact repeats of length \( \geq s \). (Using an index)
2. Extend these by up to \( k \) errors.
3. Report matches whenever length \( \ell \) is reached.

\[ \begin{array}{c}
\text{Analysis:} \\
O(n + \zeta k) \text{ time} \\
E(\zeta) = O(n^2/s) 
\end{array} \]
Finding degenerate repeats

Idea: A repeat of length \( \ell \) with \( k \) errors contains an exact match of length at least \( s := \lfloor \ell/(k + 1) \rfloor \).

Algorithm:
1. Find all exact repeats of length \( \geq s \). (Using an index)
2. Extend these by up to \( k \) errors.
3. Report matches whenever length \( \ell \) is reached.

Analysis: \( O(n + \zeta k) \) time with \( E(\zeta) = O(n^2/s) \).
Finding degenerate repeats

Idea: A repeat of length $\ell$ with $k$ errors contains an exact match of length at least $s := \lfloor \ell / (k + 1) \rfloor$.

Algorithm:
1. Find all exact repeats of length $\geq s$. (Using an index)
2. Extend these by up to $k$ errors.
3. Report matches whenever length $\ell$ is reached.

Analysis: $O(n + \zeta k)$ time with $E(\zeta) = O(n^2/4^s)$. 
Given a string $S$ and a pattern $P$, find all exact/approximate occurrences of $P$ in $S$. 
String matching

Given a string $S$ and a pattern $P$, find all exact/approximate occurrences of $P$ in $S$.

(A) Online: no preprocessing of the text, linear search time

Exact string matching
- Finite automata, e.g. Knuth-Morris-Pratt, Aho-Corasick
- Boyer-Moore
- Boyer-Moore-Horspool

Approximate string matching
- Sellers’ algorithm (dynamic programming)
- FASTA, BLAST (heuristc methods)
String matching

(B) **Offline:** preprocessing of the text, sublinear search time

Examples of full-text index structures:

- Suffix tree
- Patricia trie
- Directed acyclic word graph
- Suffix array
- String B tree
- Suffix cactus
- Suffix vector
- Factor oracle
- Enhanced suffix array
- Affix tree
- q-Gram index
String matching

(B) Offline: preprocessing of the text, sublinear search time

Examples of full-text index structures:

- Suffix tree
- Patricia trie
- Directed acyclic word graph
- Suffix array
- String B tree
- Suffix cactus
- Suffix vector
- Factor oracle
- Enhanced suffix array
- Affix tree
- q-Gram index
### Exact string matching online and offline

**Theoretical results:**

**Online search** in $O(n + m)$ time possible

**Offline search** in $O(m)$ time after $O(n)$ time preprocessing

<table>
<thead>
<tr>
<th></th>
<th>Online</th>
<th>Offline</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 pattern search</td>
<td>$O(n + m)$</td>
<td>$O(n + m)$</td>
</tr>
<tr>
<td>$k$ pattern searches</td>
<td>$O(k(n + m))$</td>
<td>$O(n + km)$</td>
</tr>
</tbody>
</table>

where $n = \text{text length}$, $m = \text{pattern length}$
Index structures in biological sequence analysis

1. Introduction
2. Suffix trees
3. Affix trees
4. Suffix arrays
5. The $q$-Gram index
6. Summary and Conclusion
A suffix of a string $S$ of length $n$ is a substring of $S$ that ends at position $n$.

The suffix tree of $S$, $T(S)$, is a rooted tree whose edges are labeled with strings such that

- the paths from the root to the leaves are the suffixes of $S$;
- all edges leaving a node begin with different characters.
Suffix Tree: Definition

- A **suffix** of a string $S$ of length $n$ is a substring of $S$ that ends at position $n$.
- The **suffix tree** of $S$, $T(S)$, is a rooted tree whose edges are labeled with strings such that
  - the paths from the root to the leaves are the suffixes of $S$;
  - all edges leaving a node begin with different characters.

$$S = \text{T A T A T} \$ \quad \text{P = A T A}$$
Suffix Tree: Definition

- A suffix of a string $S$ of length $n$ is a substring of $S$ that ends at position $n$.
- The suffix tree of $S$, $T(S)$, is a rooted tree whose edges are labeled with strings such that
  - the paths from the root to the leaves are the suffixes of $S$;
  - all edges leaving a node begin with different characters.

$$S = TATATAT\,\$$$
$$P = ATA$$
A suffix of a string $S$ of length $n$ is a substring of $S$ that ends at position $n$.

The suffix tree of $S$, $T(S)$, is a rooted tree whose edges are labeled with strings such that

- the paths from the root to the leaves are the suffixes of $S$;
- all edges leaving a node begin with different characters.
**Suffix Tree: Definition**

- A suffix of a string $S$ of length $n$ is a substring of $S$ that ends at position $n$.
- The suffix tree of $S$, $T(S)$, is a rooted tree whose edges are labeled with strings such that
  - the paths from the root to the leaves are the suffixes of $S$;
  - all edges leaving a node begin with different characters.
A **suffix** of a string $S$ of length $n$ is a substring of $S$ that ends at position $n$.

The **suffix tree** of $S$, $T(S)$, is a rooted tree whose edges are labeled with strings such that

- the paths from the root to the leaves are the suffixes of $S$;
- all edges leaving a node begin with different characters.
A suffix of a string $S$ of length $n$ is a substring of $S$ that ends at position $n$.

The suffix tree of $S$, $T(S)$, is a rooted tree whose edges are labeled with strings such that

- the paths from the root to the leaves are the suffixes of $S$;
- all edges leaving a node begin with different characters.
A suffix of a string $S$ of length $n$ is a substring of $S$ that ends at position $n$.

The suffix tree of $S$, $T(S)$, is a rooted tree whose edges are labeled with strings such that:

- the paths from the root to the leaves are the suffixes of $S$;
- all edges leaving a node begin with different characters.

\[ S = \text{TATAT}\$ \]
\[ P = \text{TATT} \]
A suffix of a string $S$ of length $n$ is a substring of $S$ that ends at position $n$.

The suffix tree of $S$, $T(S)$, is a rooted tree whose edges are labeled with strings such that

- the paths from the root to the leaves are the suffixes of $S$;
- all edges leaving a node begin with different characters.
Suffix Tree: Definition

- A suffix of a string $S$ of length $n$ is a substring of $S$ that ends at position $n$.
- The suffix tree of $S$, $T(S)$, is a rooted tree whose edges are labeled with strings such that
  - the paths from the root to the leaves are the suffixes of $S$;
  - all edges leaving a node begin with different characters.

$S = T A T A T \$ \\
$P = T A T T$

![Diagram of a suffix tree](image)
A suffix of a string $S$ of length $n$ is a substring of $S$ that ends at position $n$.

The suffix tree of $S$, $T(S)$, is a rooted tree whose edges are labeled with strings such that

- the paths from the root to the leaves are the suffixes of $S$;
- all edges leaving a node begin with different characters.

\[ S = \text{T A T A T $} \]
\[ P = \text{T A T T} \]
A suffix of a string $S$ of length $n$ is a substring of $S$ that ends at position $n$.

The suffix tree of $S$, $T(S)$, is a rooted tree whose edges are labeled with strings such that

- the paths from the root to the leaves are the suffixes of $S$;
- all edges leaving a node begin with different characters.

\[
S = \text{TATAT}\$
\]

\[
P = \text{TATT}
\]
A suffix of a string $S$ of length $n$ is a substring of $S$ that ends at position $n$.

The suffix tree of $S$, $T(S)$, is a rooted tree whose edges are labeled with strings such that
  - the paths from the root to the leaves are the suffixes of $S$;
  - all edges leaving a node begin with different characters.
A larger example

\[ S = \text{Mississippi} \]$
Suffix tree properties

- \( T(S) \) represents exactly the substrings of \( S \).
- \( T(S) \) allows to enumerate these substrings and their locations in \( S \) in a convenient way.
- This is very useful for many pattern recognition problems, for example:
  - exact string matching as part of other applications, e.g. detecting DNA contamination
  - all-pairs suffix-prefix matching, important in fragment assembly
  - finding repeats and palindromes, tandem repeats, degenerate repeats
  - DNA primer design
  - DNA chip design
  - ...

See also:
Space usage of suffix trees

Observation: $T(S)$ requires $\mathcal{O}(n)$ space.

Proof sketch:

1. $T(S)$ has at most $n$ leaves.
2. Each internal node is branching $\Rightarrow$ at most $n - 1$ internal nodes.
3. A tree with at most $2n - 1$ nodes has at most $2n - 2$ edges.
4. Each node requires constant space.
5. Each edge label is a substring of $S$ $\Rightarrow$ pair of pointers $(i, j)$ into $S$. 

$S = \text{Mississippi} \, $
Space usage of suffix trees

Observation: \( T(S) \) requires \( \mathcal{O}(n) \) space.

Proof sketch:

1. \( T(S) \) has at most \( n \) leaves.
2. Each internal node is branching \( \Rightarrow \) at most \( n - 1 \) internal nodes.
3. A tree with at most \( 2n - 1 \) nodes has at most \( 2n - 2 \) edges.
4. Each node requires constant space.
5. Each edge label is a substring of \( S \) \( \Rightarrow \) pair of pointers \((i, j)\) into \( S \).
Space usage of suffix trees

Observation: \( T(S) \) requires \( \mathcal{O}(n) \) space.

Proof sketch:

1. \( T(S) \) has at most \( n \) leaves.
2. Each internal node is branching \( \Rightarrow \) at most \( n - 1 \) internal nodes.
3. A tree with at most \( 2n - 1 \) nodes has at most \( 2n - 2 \) edges.
4. Each node requires constant space.
5. Each edge label is a substring of \( S \) \( \Rightarrow \) pair of pointers \( (i, j) \) into \( S \).
**Space usage of suffix trees**

**Observation:** \( T(S) \) requires \( \mathcal{O}(n) \) space.

**Proof sketch:**

1. \( T(S) \) has at most \( n \) leaves.
2. Each internal node is branching \( \Rightarrow \) at most \( n - 1 \) internal nodes.
3. A tree with at most \( 2n - 1 \) nodes has at most \( 2n - 2 \) edges.
4. Each node requires constant space.
5. Each edge label is a substring of \( S \) \( \Rightarrow \) pair of pointers \((i, j)\) into \( S\).
Space usage of suffix trees

Observation: $T(S)$ requires $\mathcal{O}(n)$ space.

Proof sketch:

1. $T(S)$ has at most $n$ leaves.
2. Each internal node is branching $\Rightarrow$ at most $n - 1$ internal nodes.
3. A tree with at most $2n - 1$ nodes has at most $2n - 2$ edges.
4. Each node requires constant space.
5. Each edge label is a substring of $S$ $\Rightarrow$ pair of pointers $(i, j)$ into $S$. 

$$S = \text{Mississippi}$$
Space usage of suffix trees

**Observation:** $T(S)$ requires $O(n)$ space.

**Proof sketch:**

1. $T(S)$ has at most $n$ leaves.
2. Each internal node is branching $\Rightarrow$ at most $n - 1$ internal nodes.
3. A tree with at most $2n - 1$ nodes has at most $2n - 2$ edges.
4. Each node requires constant space.
5. Each edge label is a substring of $S$ $\Rightarrow$ pair of pointers $(i, j)$ into $S$. 
Space usage of suffix trees

Observation: $T(S)$ requires $O(n)$ space.

Proof sketch:

1. $T(S)$ has at most $n$ leaves.
2. Each internal node is branching $\Rightarrow$ at most $n - 1$ internal nodes.
3. A tree with at most $2n - 1$ nodes has at most $2n - 2$ edges.
4. Each node requires constant space.
5. Each edge label is a substring of $S \Rightarrow$ pair of pointers $(i, j)$ into $S$. 

$S = \text{Mississippi}$

![Diagram of a suffix tree with leaf labels and node labels indicating edge pointers]
Space usage of suffix trees

Observation: $T(S)$ requires $O(n)$ space.

Proof sketch:

1. $T(S)$ has at most $n$ leaves.
2. Each internal node is branching $\Rightarrow$ at most $n - 1$ internal nodes.
3. A tree with at most $2n - 1$ nodes has at most $2n - 2$ edges.
4. Each node requires constant space.
5. Each edge label is a substring of $S$ $\Rightarrow$ pair of pointers $(i, j)$ into $S$. 

\[ S = \text{Mississippi} \]

1 2 3 4 5 6 7 8 9 10 11 12
Space usage of suffix trees

**Observation:** $T(S)$ requires $\mathcal{O}(n)$ space.

**Proof sketch:**

1. $T(S)$ has at most $n$ leaves.
2. Each internal node is branching $\Rightarrow$ at most $n - 1$ internal nodes.
3. A tree with at most $2n - 1$ nodes has at most $2n - 2$ edges.
4. Each node requires constant space.
5. Each edge label is a substring of $S \Rightarrow$ pair of pointers $(i, j)$ into $S$. 

$S = \text{Mississippi} \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11 \quad 12$
Space usage of suffix trees

Observation: \( T(S) \) requires \( \mathcal{O}(n) \) space.

Proof sketch:

1. \( T(S) \) has at most \( n \) leaves.
2. Each internal node is branching \( \Rightarrow \) at most \( n - 1 \) internal nodes.
3. A tree with at most \( 2n - 1 \) nodes has at most \( 2n - 2 \) edges.
4. Each node requires constant space.
5. Each edge label is a substring of \( S \) \( \Rightarrow \) pair of pointers \((i, j)\) into \( S\).
Space usage of suffix trees

Observation: $T(S)$ requires $O(n)$ space.

Proof sketch:

1. $T(S)$ has at most $n$ leaves.
2. Each internal node is branching $\Rightarrow$ at most $n - 1$ internal nodes.
3. A tree with at most $2n - 1$ nodes has at most $2n - 2$ edges.
4. Each node requires constant space.
5. Each edge label is a substring of $S$ $\Rightarrow$ pair of pointers $(i, j)$ into $S$. 
Space usage of suffix trees

Observation: \( T(S) \) requires \( \mathcal{O}(n) \) space.

Proof sketch:

1. \( T(S) \) has at most \( n \) leaves.
2. Each internal node is branching \( \Rightarrow \) at most \( n - 1 \) internal nodes.
3. A tree with at most \( 2n - 1 \) nodes has at most \( 2n - 2 \) edges.
4. Each node requires constant space.
5. Each edge label is a substring of \( S \) \( \Rightarrow \) pair of pointers \((i, j)\) into \( S \).
Space usage of suffix trees

Observation: $T(S)$ requires $\mathcal{O}(n)$ space.

Proof sketch:

1. $T(S)$ has at most $n$ leaves.
2. Each internal node is branching $\Rightarrow$ at most $n - 1$ internal nodes.
3. A tree with at most $2n - 1$ nodes has at most $2n - 2$ edges.
4. Each node requires constant space.
5. Each edge label is a substring of $S$ $\Rightarrow$ pair of pointers $(i, j)$ into $S$. 

$S = \text{Mississippi} \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11 \quad 12$
Space usage of suffix trees

Observation: \( T(S) \) requires \( \mathcal{O}(n) \) space.

Proof sketch:

1. \( T(S) \) has at most \( n \) leaves.
2. Each internal node is branching \( \Rightarrow \) at most \( n - 1 \) internal nodes.
3. A tree with at most \( 2n - 1 \) nodes has at most \( 2n - 2 \) edges.
4. Each node requires constant space.
5. Each edge label is a substring of \( S \) \( \Rightarrow \) pair of pointers \((i, j)\) into \( S \).
Space usage of suffix trees

Observation: $T(S)$ requires $O(n)$ space.

Proof sketch:

1. $T(S)$ has at most $n$ leaves.
2. Each internal node is branching $\Rightarrow$ at most $n - 1$ internal nodes.
3. A tree with at most $2n - 1$ nodes has at most $2n - 2$ edges.
4. Each node requires constant space.
5. Each edge label is a substring of $S$ $\Rightarrow$ pair of pointers $(i, j)$ into $S$. 

$S = \text{Mississippi}$

![Diagram of suffix tree with labels and indices]
Space usage of suffix trees

Observation: $T(S)$ requires $\mathcal{O}(n)$ space.

Proof sketch:

1. $T(S)$ has at most $n$ leaves.
2. Each internal node is branching $\Rightarrow$ at most $n - 1$ internal nodes.
3. A tree with at most $2n - 1$ nodes has at most $2n - 2$ edges.
4. Each node requires constant space.
5. Each edge label is a substring of $S$ $\Rightarrow$ pair of pointers $(i, j)$ into $S$. 

$S = \text{Mississippi}$
Space usage of suffix trees

Observation: \( T(S) \) requires \( \mathcal{O}(n) \) space.

Proof sketch:

1. \( T(S) \) has at most \( n \) leaves.
2. Each internal node is branching \( \Rightarrow \) at most \( n - 1 \) internal nodes.
3. A tree with at most \( 2n - 1 \) nodes has at most \( 2n - 2 \) edges.
4. Each node requires constant space.
5. Each edge label is a substring of \( S \) \( \Rightarrow \) pair of pointers \((i, j)\) into \( S\).
Space usage of suffix trees

Observation: $T(S)$ requires $\mathcal{O}(n)$ space.

Proof sketch:

1. $T(S)$ has at most $n$ leaves.
2. Each internal node is branching $\Rightarrow$ at most $n - 1$ internal nodes.
3. A tree with at most $2n - 1$ nodes has at most $2n - 2$ edges.
4. Each node requires constant space.
5. Each edge label is a substring of $S$ $\Rightarrow$ pair of pointers $(i, j)$ into $S$. 

$S = \text{M i s s i s s i p p i }$
Space usage of suffix trees

Observation: $T(S)$ requires $O(n)$ space.

Proof sketch:

1. $T(S)$ has at most $n$ leaves.
2. Each internal node is branching $\Rightarrow$ at most $n - 1$ internal nodes.
3. A tree with at most $2n - 1$ nodes has at most $2n - 2$ edges.
4. Each node requires constant space.
5. Each edge label is a substring of $S$ $\Rightarrow$ pair of pointers $(i, j)$ into $S$. 

$S = \text{Mississippi} \quad S = \text{Mississippi} \quad S = \text{Mississippi} \quad S = \text{Mississippi}$

$1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11 \quad 12$
Representation of suffix trees

$S = \text{T A T A T }$ $\$

1 2 3 4 5 6

(2,3) (1,1) (6,6)

(6,6) (2,3) (6,6)

(4,6) (6,6) (6,6)

(4,6)

Standard representation of trees:

- Store nodes as records with child and sibling pointer.

⇒ about $32n$ bytes in the worst case
Representation of suffix trees

$S = \text{T A T A T }\$  

Standard representation of trees:
- Store nodes as records with child and sibling pointer.
⇒ about $32n$ bytes in the worst case

More efficient representation: (Giegerich, Kurtz & JS, SP&E 2003)
- Avoid storing redundant information.
⇒ below $12n$ bytes in the worst case, $8.5n$ on average
Construction of suffix trees

Theorem [Weiner, 1973]: \( T(S) \) can be constructed in \( O(n) \) time.


A simpler algorithm: Write-Only, Top-Down (WOTD).

Analysis: \( O(n^2) \) worst-case, \( O(n \log n) \) expected time, \( O(n) \) space.

Note: The WOTD algorithm is well suited for a lazy construction.
Construction of suffix trees

Theorem [Weiner, 1973]: $T(S)$ can be constructed in $\mathcal{O}(n)$ time.


Analysis: $O(n^2)$ worst-case, $O(n \log n)$ expected time, $O(n)$ space.

Note: The WOTD algorithm is well suited for a lazy construction.
Construction of suffix trees

Theorem [Weiner, 1973]: $T(S)$ can be constructed in $O(n)$ time.


A simpler algorithm: **Write-Only, Top-Down** (WOTD).
Construction of suffix trees

Theorem [Weiner, 1973]: $T(S)$ can be constructed in $O(n)$ time.


A simpler algorithm: Write-Only, Top-Down (WOTD).
Construction of suffix trees

Theorem [Weiner, 1973]: $T(S)$ can be constructed in $O(n)$ time.


A simpler algorithm: Write-Only, Top-Down (WOTD).

\[
\begin{array}{c}
T \ A \ T \ A \ T \ \$
\hline
A \ T \ A \ T \ \$
\hline
T \ A \ T \ \$
\hline
A \ T \ \$
\hline
T \ \$
\hline
\$
\end{array}
\Rightarrow
\begin{array}{c}
A \ \$
\hline
T \ \$
\hline
A \ \$
\hline
T \ \$
\hline
\$
\end{array}
\Rightarrow
\begin{array}{c}
A \ \$
\hline
T \ \$
\hline
A \ \$
\hline
T \ \$
\hline
\$
\end{array}
\]

Analysis:
$O(n^2)$ worst-case, $O(n \log n)$ expected time, $O(n)$ space

Note: The WOTD algorithm is well suited for a lazy construction.
Construction of suffix trees

Theorem [Weiner, 1973]: $T(S)$ can be constructed in $\mathcal{O}(n)$ time.


A simpler algorithm: Write-Only, Top-Down (WOTD).

\[
\begin{array}{c}
\text{T A T A T}\$
\text{A T A T}\$
\text{T A T}\$
\text{T}\$
\text{A}\$
\text{T}\$
\text{1}
\text{2}
\text{3}
\text{4}
\text{5}
\text{6}
\Rightarrow
\end{array}
\]

Analysis: $\mathcal{O}(n^2)$ worst-case, $\mathcal{O}(n \log n)$ expected time, $\mathcal{O}(n)$ space.

Note: The WOTD algorithm is well suited for a lazy construction.
Construction of suffix trees

Theorem [Weiner, 1973]: $T(S)$ can be constructed in $\mathcal{O}(n)$ time.


A simpler algorithm: Write-Only, Top-Down (WOTD).

```
TAATT\$
ATATT\$6
TATT\$5
AT\$4
T\$3
\$2
\$1
```

⇒

```
A\$4
T\$2
```

⇒

```
A\$
T\$
```

⇒

```
A\$
T\$
```

⇒

```
A\$
T\$
```

```
A\$
T\$
```

Note:
The WOTD algorithm is well suited for a lazy construction.
Construction of suffix trees

Theorem [Weiner, 1973]: \( T(S) \) can be constructed in \( O(n) \) time.


A simpler algorithm: Write-Only, Top-Down (WOTD).

Analysis: \( O(n^2) \) worst-case, \( O(n \log n) \) expected time, \( O(n) \) space
Construction of suffix trees

Theorem [Weiner, 1973]: $T(S)$ can be constructed in $\mathcal{O}(n)$ time.


A simpler algorithm: **Write-Only, Top-Down (WOTD)**.

Analysis: $\mathcal{O}(n^2)$ worst-case, $\mathcal{O}(n \log n)$ expected time, $\mathcal{O}(n)$ space

**Note:** The WOTD algorithm is well suited for a lazy construction.
Lazy construction of suffix trees

Experimental results:
index construction plus $\rho n$ pattern searches for $\rho \in [0, 1]$

\[\text{mcch = suffix tree (McCreight’s algorithm with hash tables)}\]
\[\text{bmh = online search (Boyer-Moore-Horspool algorithm)}\]
Lazy construction of suffix trees

Experimental results:
index construction plus $\rho n$ pattern searches for $\rho \in [0, 1]$

mcch = suffix tree (McCreight’s algorithm with hash tables)
bmh = online search (Boyer-Moore-Horspool algorithm)
wotdlazy = suffix tree write-only top-down construction (lazy version)
Index structures in biological sequence analysis

1. Introduction
2. Suffix trees
3. Affix trees
4. Suffix arrays
5. The $q$-Gram index
6. Summary and Conclusion
More complicated search: Inside-out

Define an RNA hairpin (in **HyPaL** syntax):

```plaintext
stem = .4
hairpin = @stem CGGA complement(@stem)
```

```
C
G
G
A
```

Search strategy:

1. Find all exact matches of CGGA (left-to-right)
2. Extend by an arbitrary character a to the right, complementary character to the left, an arbitrary character b to the right, complementary character to the left, ...
More complicated search: Inside-out

Define an RNA hairpin (in \textbf{HyPaL} syntax):

\begin{align*}
\text{stem} &= .4 \\
\text{hairpin} &= \@\text{stem} \text{ CGGA complement(}@\text{stem})
\end{align*}

Search strategy:

1. Find all exact matches of \textbf{CGGA} (left-to-right)
More complicated search: Inside-out

Define an RNA hairpin (in HyPaL syntax):

stem = .4

hairpin = @stem CGGA complement(@stem)

Search strategy:

1. Find all exact matches of CGGA (left-to-right)
More complicated search: Inside-out

Define an RNA hairpin (in **HyPaL** syntax):

```
stem = .4
hairpin = @stem CGGA complement(@stem)
```

**Search strategy:**

1. Find all exact matches of **CGGA** (left-to-right)
More complicated search: Inside-out

Define an RNA hairpin (in **HyPaL** syntax):

\[
\text{stem} = .4 \\
\text{hairpin} = \@stem \text{ CGGA complement(@stem)}
\]

**Search strategy:**

1. Find all exact matches of **CGGA** (left-to-right)

![](search_tree.png)
More complicated search: Inside-out

Define an RNA hairpin (in **HyPaL** syntax):

\[
\begin{align*}
\text{stem} & = .4 \\
\text{hairpin} & = @\text{stem} \text{ CGGA complement}(@\text{stem})
\end{align*}
\]

Search strategy:

1. Find all exact matches of **CGGA** *(left-to-right)*
More complicated search: Inside-out

Define an RNA hairpin (in HyPaL syntax):

stem = .4

hairpin = @stem CGGA complement(@stem)

Search strategy:

1. Find all exact matches of CGGA (left-to-right)
2. Extend by
   - an arbitrary character \textit{a} to the right
More complicated search: Inside-out

Define an RNA hairpin (in **HyPaL** syntax):

\[
\text{stem} = .4 \\
\text{hairpin} = \@\text{stem} \text{ CGGA complement}(\@\text{stem})
\]

Search strategy:

1. Find all exact matches of **CGGA** (left-to-right)
2. Extend by
   - an arbitrary character *a* to the right
   - *complementary* character to the left
More complicated search: Inside-out

Define an RNA hairpin (in HyPaL syntax):

```latex
\begin{align*}
\text{stem} &= .4 \\
\text{hairpin} &= \@\text{stem} \text{ CGGA complement(}@\text{stem})
\end{align*}
```

Search strategy:

1. Find all exact matches of CGGA (left-to-right)
2. Extend by
   - an arbitrary character \textit{a} to the right
   - \textit{complementary} character to the left
   - an arbitrary character \textit{b} to the right
Define an RNA hairpin (in **HyPaL** syntax):

```
stem = .4
hairpin = @stem CGGA complement(@stem)
```

**Search strategy:**

1. Find all exact matches of **CGGA** (*left-to-right*)
2. Extend by
   - an arbitrary character *a* to the right
   - *complementary* character to the left
   - an arbitrary character *b* to the right
   - *complementary* character to the left
More complicated search: Inside-out

Define an RNA hairpin (in HyPaL syntax):

stem = .4

hairpin = @stem CGGA complement(@stem)

Search strategy:

1. Find all exact matches of CGGA (left-to-right)
2. Extend by
   - an arbitrary character \( a \) to the right
   - complementary character to the left
   - an arbitrary character \( b \) to the right
   - complementary character to the left
   - \ldots
Towards a bi-directional data structure

- **Suffix tree** is **asymmetric**: left-to-right matching only
- Similar data structure for right-to-left matching:
  reverse prefix tree
Towards a bi-directional data structure

- **Suffix tree** is asymmetric: left-to-right matching only
- Similar data structure for right-to-left matching: reverse prefix tree
- **Idea:** Create the **atomic suffix tree** and reverse prefix tree.

![Suffix Tree and Reverse Prefix Tree Diagram]

$$S = \text{A A T A T A}$$

$$S^{-1} = \text{A T A T A}$$
Towards a bi-directional data structure

- **Suffix tree** is asymmetric: left-to-right matching only
- Similar data structure for right-to-left matching: reverse prefix tree
- **Idea**: Create the atomic suffix tree and reverse prefix tree.
- Create bi-directional links between corresponding nodes.

\[ S = A A T A T A \]
\[ S^{-1} = A T A T A A \]
Towards a bi-directional data structure

- **Suffix tree** is asymmetric: left-to-right matching only
- Similar data structure for right-to-left matching: reverse prefix tree
- **Idea:** Create the **atomic suffix tree** and reverse prefix tree.
- Create **bi-directional links** between corresponding nodes.
Towards a bi-directional data structure

- **Suffix tree** is asymmetric: left-to-right matching only
- Similar data structure for right-to-left matching: reverse prefix tree
- **Idea:** Create the atomic suffix tree and reverse prefix tree.
- Create bi-directional links between corresponding nodes.

\[
\begin{align*}
S &= T A T A T A \\
P &= A T A T \\
S^{-1} &= A T A T A A
\end{align*}
\]
Towards a bi-directional data structure

- **Suffix tree** is asymmetric: left-to-right matching only
- Similar data structure for right-to-left matching: reverse prefix tree
- **Idea:** Create the **atomic suffix tree** and reverse prefix tree.
- Create bi-directional links between corresponding nodes.

\[ S = A A T A T A \]
\[ P = A T A T \]

\[ S^{-1} = A T A T A A \]
Towards a bi-directional data structure

- **Suffix tree** is asymmetric: left-to-right matching only
- Similar data structure for right-to-left matching: reverse prefix tree
- **Idea:** Create the **atomic suffix tree** and reverse prefix tree.
- Create **bi-directional links** between corresponding nodes.

\[ S = A A T A T A \]
\[ S^{-1} = A T A T A A \]
\[ P = A T A T \]
Towards a bi-directional data structure

- **Suffix tree** is asymmetric: left-to-right matching only
- Similar data structure for right-to-left matching: reverse prefix tree
- **Idea:** Create the **atomic suffix tree** and reverse prefix tree.
- Create **bi-directional links** between corresponding nodes.

![Diagram of bi-directional links between suffix tree and reverse prefix tree]
Towards a bi-directional data structure

- **Suffix tree** is asymmetric: left-to-right matching only
- Similar data structure for right-to-left matching: reverse prefix tree
- **Idea:** Create the atomic suffix tree and reverse prefix tree.
- Create bi-directional links between corresponding nodes.

**Problem:** quadratic space
The affix tree

- To save space apply same idea to compact suffix tree and reverse prefix tree.
- Problem: Corresponding node might be missing $\Rightarrow$ create the missing nodes.
- The joined data structure is called the affix tree of $S$. 

$$S = A A T A T A$$

Diagram of the affix tree

$$S^{-1} = A A A T T A$$
The affix tree

- To save space apply same idea to **compact suffix tree** and reverse prefix tree.
- **Problem:** Corresponding node might be missing ⇒ create the missing nodes.
- The joined data structure is called the **affix tree** of $S$. 

\[ S = A A T A T A \]

\[ S^{-1} = A T A T A A \]
The affix tree

- To save space apply same idea to **compact suffix tree** and reverse prefix tree.
- **Problem:** Corresponding node might be missing ⇒ create the missing nodes.
- The joined data structure is called the **affix tree** of $S$. 
The affix tree

- To save space apply same idea to **compact suffix tree** and reverse prefix tree.
- **Problem:** Corresponding node might be missing ⇒ create the missing nodes.
- The joined data structure is called the **affix tree** of $S$. 

$S = A A T A T A$

$S^{-1} = A T A T A A$
The affix tree

- To save space apply same idea to **compact suffix tree** and reverse prefix tree.
- **Problem:** Corresponding node might be missing $\Rightarrow$ create the missing nodes.
- The joined data structure is called the **affix tree** of $S$. 

![Diagram of the affix tree]
The affix tree

- To save space apply same idea to **compact suffix tree** and reverse prefix tree.
- **Problem:** Corresponding node might be missing ⇒ create the missing nodes.
- The joined data structure is called the **affix tree** of $S$. 

$$S = A A T A T A$$

$$S^{-1} = A T A T A A$$
The affix tree

- To save space apply same idea to **compact suffix tree** and reverse prefix tree.
- **Problem:** Corresponding node might be missing ⇒ create the missing nodes.
- The joined data structure is called the **affix tree** of $S$. 

$$S = A A T A T A$$

$$S^{-1} = A T A T A A$$
The affix tree

- To save space apply same idea to compact suffix tree and reverse prefix tree.
- **Problem:** Corresponding node might be missing ⇒ create the missing nodes.
- The joined data structure is called the **affix tree** of \( S \).
The affix tree

- To save space apply same idea to **compact suffix tree** and reverse prefix tree.
- **Problem:** Corresponding node might be missing ⇒ create the missing nodes.
- The joined data structure is called the **affix tree** of \( S \).
The affix tree

- To save space apply same idea to **compact suffix tree** and reverse prefix tree.
- **Problem:** Corresponding node might be missing ⇒ create the missing nodes.
- The joined data structure is called the **affix tree** of $S$. 

$$S = A A T A T A$$

$$S^{-1} = A T A T A A$$
The affix tree

- To save space apply same idea to *compact suffix tree* and reverse prefix tree.

- **Problem:** Corresponding node might be missing ⇒ create the missing nodes.

- The joined data structure is called the **affix tree** of $S$.  

\[ S = A A T A T A \]  
\[ S^{-1} = A T A T A A \]
The affix tree

- To save space apply same idea to **compact suffix tree** and reverse prefix tree.
- **Problem:** Corresponding node might be missing $\Rightarrow$ create the missing nodes.
- The joined data structure is called the **affix tree** of $S$. 

$$S = A A T A T A$$

$$S^{-1} = A T A T A A$$
The affix tree

- To save space apply same idea to **compact suffix tree** and reverse prefix tree.
- **Problem:** Corresponding node might be missing ⇒ create the missing nodes.
- The joined data structure is called the **affix tree** of \( S \).
The affix tree

- To save space apply same idea to compact suffix tree and reverse prefix tree.
- **Problem:** Corresponding node might be missing \(\Rightarrow\) create the missing nodes.
- The joined data structure is called the **affix tree** of \(S\).
The affix tree properties

- The affix tree of $S$ requires $O(n)$ space:
  - at most $2n - 2$ nodes
  - at most $2n - 4$ edges
  - as for suffix trees, edge labels can be represented by pairs of pointers into $S$

The affix tree can be constructed in $O(n)$ time and space (Maaß, CPM 2000).

Supports all applications of suffix tree, and some more.
Index structures in biological sequence analysis

1. Introduction
2. Suffix trees
3. Affix trees
4. Suffix arrays
5. The $q$-Gram index
6. Summary and Conclusion
The suffix array

Suffix tree, affix tree:
- very flexible data structures
- support a ‘myriad’ of applications
- But: require considerable space in practice
  - Suffix tree: 10-20 bytes per text character
  - Affix tree: roughly twice as much

Alternative using less space: **suffix array** (Manber & Myers, 1993)
The suffix array

$S = \text{Mississippi }$ $
\begin{array}{cccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
\end{array}
$

- Array containing suffix numbers, lexicographically sorted by their suffixes
- Space usage: $4n$ bytes
- Query time: $O(|P| \log n)$ time (or $O(|P| + \log n)$ with tricks)
- Technique to simulate all suffix tree operations: enhanced suffix array (Abouelhoda et al., JDA 2004)
Construction of suffix arrays

(a) Read leaf numbers of suffix tree
   \[\mathcal{O}(n)\] time

(b) Direct construction of suffix arrays:
   simple algorithms use \[\mathcal{O}(n^2)\] or \[\mathcal{O}(n \log n)\] time
   - Kim et al., CPM 2003
   - Ko & Aluru, CPM 2003
   - Kärkkäinen & Sanders, ICALP 2003
   \[\mathcal{O}(n)\] time

(c) Practical algorithms have worse time complexities
   - Bucket-pointer refinement (Schürmann & JS, *SP&E* 2006)
   \[\mathcal{O}(n^2)\] time in worst case, **much better in practice**
### The bucket-pointer refinement algorithm

#### DNA sequences

<table>
<thead>
<tr>
<th>DNA sequences</th>
<th>bpr</th>
<th>deep cache</th>
<th>copy</th>
<th>qsufsort</th>
<th>difference cover</th>
<th>divide &amp; conquer</th>
<th>skew</th>
</tr>
</thead>
<tbody>
<tr>
<td>E. coli genome</td>
<td>1.46</td>
<td>1.71</td>
<td>3.69</td>
<td>2.89</td>
<td>2.87</td>
<td>4.32</td>
<td>13.48</td>
</tr>
<tr>
<td>A. thaliana chr. 4</td>
<td>5.24</td>
<td>5.01</td>
<td>12.24</td>
<td>9.94</td>
<td>8.42</td>
<td>13.29</td>
<td>16.94</td>
</tr>
<tr>
<td>H.sapiens chr. 22</td>
<td>15.92</td>
<td>16.64</td>
<td>40.08</td>
<td>30.04</td>
<td>26.52</td>
<td>44.93</td>
<td>51.31</td>
</tr>
<tr>
<td>C. elegans chr. 1</td>
<td>5.70</td>
<td>6.03</td>
<td>20.79</td>
<td>17.48</td>
<td>13.09</td>
<td>16.94</td>
<td>18.64</td>
</tr>
<tr>
<td>6 Streptococci</td>
<td>5.27</td>
<td>5.97</td>
<td>14.43</td>
<td>10.38</td>
<td>13.16</td>
<td>14.50</td>
<td>16.40</td>
</tr>
<tr>
<td>4 Chlamydompha</td>
<td>2.31</td>
<td>3.43</td>
<td>17.46</td>
<td>14.45</td>
<td>7.49</td>
<td>5.59</td>
<td>6.13</td>
</tr>
<tr>
<td>3 E. coli</td>
<td>8.01</td>
<td>13.75</td>
<td>437.18</td>
<td>1,294.30</td>
<td>32.72</td>
<td>20.57</td>
<td>21.58</td>
</tr>
</tbody>
</table>

#### Text

<table>
<thead>
<tr>
<th>Text</th>
<th>bpr</th>
<th>deep cache</th>
<th>copy</th>
<th>qsufsort</th>
<th>difference cover</th>
<th>divide &amp; conquer</th>
<th>skew</th>
</tr>
</thead>
<tbody>
<tr>
<td>bible</td>
<td>1.90</td>
<td>1.41</td>
<td>2.91</td>
<td>2.24</td>
<td>3.17</td>
<td>3.74</td>
<td>6.39</td>
</tr>
<tr>
<td>world192</td>
<td>1.05</td>
<td>0.73</td>
<td>1.47</td>
<td>1.24</td>
<td>1.91</td>
<td>2.28</td>
<td>3.57</td>
</tr>
<tr>
<td>rfc</td>
<td>31.16</td>
<td>26.37</td>
<td>57.97</td>
<td>55.21</td>
<td>58.10</td>
<td>71.10</td>
<td>101.57</td>
</tr>
<tr>
<td>sprot34</td>
<td>35.75</td>
<td>29.77</td>
<td>71.95</td>
<td>71.96</td>
<td>60.24</td>
<td>81.76</td>
<td>104.71</td>
</tr>
<tr>
<td>howto</td>
<td>22.10</td>
<td>19.63</td>
<td>39.92</td>
<td>47.27</td>
<td>41.14</td>
<td>48.45</td>
<td>83.32</td>
</tr>
<tr>
<td>reuters</td>
<td>47.32</td>
<td>52.74</td>
<td>111.80</td>
<td>157.63</td>
<td>73.19</td>
<td>108.85</td>
<td>108.84</td>
</tr>
<tr>
<td>w3c2</td>
<td>41.04</td>
<td>61.37</td>
<td>82.46</td>
<td>167.76</td>
<td>69.40</td>
<td>96.02</td>
<td>105.89</td>
</tr>
<tr>
<td>jdk13</td>
<td>40.35</td>
<td>47.23</td>
<td>101.58</td>
<td>263.86</td>
<td>73.75</td>
<td>97.12</td>
<td>98.13</td>
</tr>
<tr>
<td>linux</td>
<td>23.72</td>
<td>23.95</td>
<td>50.93</td>
<td>99.47</td>
<td>61.01</td>
<td>65.66</td>
<td>98.06</td>
</tr>
<tr>
<td>etext99</td>
<td>32.60</td>
<td>33.25</td>
<td>68.84</td>
<td>267.48</td>
<td>61.19</td>
<td>65.31</td>
<td>110.95</td>
</tr>
<tr>
<td>gcc</td>
<td>33.19</td>
<td>76.23</td>
<td>2,894.81</td>
<td>21,836.56</td>
<td>59.44</td>
<td>73.54</td>
<td>83.96</td>
</tr>
</tbody>
</table>

Construction time (sec.)
Application to approximate DNA matching

QUASAR: Q-Gram based database search Using A Suffix ARray
(Burkhardt et al., RECOMB 1999)

Idea: Search **dot-plot** for regions with many *q*-Grams.
Application to approximate DNA matching

**QUASAR:** Q-Gram based database search Using A Suffix ARray

(Burkhardt et al., RECOMB 1999)

Idea: Search **dot-plot** for regions with many q-Grams.

Use **suffix array** to locate all occurrences of a q-Gram in the database $S$: 
QUASAR algorithm

Algorithm: Filter database for relevant blocks.

1. Divide database into overlapping blocks.
2. Shift window of certain size over query; for each database block find the number of matching $q$-Grams from the current window; if this number is at least $T(n, q, e) := n + 1 - q(e + 1)$, consider this block relevant.
3. Test the relevant blocks with a more complex method.
QUASAR algorithm

**Algorithm:** Filter database for relevant blocks.

1. Divide database into overlapping blocks.
2. Shift window of certain size over query; for each database block find the number of matching $q$-Grams from the current window; if this number is at least $T(n, q, e) := n + 1 - q(e + 1)$, consider this block *relevant*.
3. Test the relevant blocks with a more complex method.
QUASAR algorithm

**Algorithm:** Filter database for relevant blocks.

1. Divide database into overlapping blocks.
2. Shift window of certain size over query; for each database block find the number of matching $q$-Grams from the current window; if this number is at least $T(n, q, e) := n + 1 - q(e + 1)$, consider this block *relevant*.
3. Test the relevant blocks with a more complex method.
**QUASAR algorithm**

**Algorithm:** Filter database for relevant blocks.

1. Divide database into overlapping blocks.
2. Shift window of certain size over query; for each database block find the number of matching $q$-Grams from the current window; if this number is at least $T(n, q, e) := n + 1 - q(e + 1)$, consider this block *relevant*.
3. Test the relevant blocks with a more complex method.
**QUASAR algorithm**

**Algorithm:** Filter database for relevant blocks.

1. Divide database into overlapping blocks.
2. Shift window of certain size over query; for each database block find the number of matching \( q \)-Grams from the current window; if this number is at least \( T(n, q, e) := n + 1 - q(e + 1) \), consider this block *relevant*.
3. Test the relevant blocks with a more complex method.
**QUASAR algorithm**

**Algorithm:** Filter database for relevant blocks.

1. Divide database into overlapping blocks.
2. Shift window of certain size over query; for each database block find the number of matching \( q \)-Grams from the current window; if this number is at least \( T(n, q, e) := n + 1 - q(e + 1) \), consider this block *relevant*.
3. Test the relevant blocks with a more complex method.

---

*Diagram showing database and relevant blocks.*
**QUASAR algorithm**

**Algorithm:** Filter database for relevant blocks.

1. Divide database into overlapping blocks.

2. Shift window of certain size over query; for each database block find the number of matching $q$-Grams from the current window; if this number is at least $T(n, q, e) := n + 1 - q(e + 1)$, consider this block relevant.

3. Test the relevant blocks with a more complex method.
Algorithm: Filter database for relevant blocks.

1. Divide database into overlapping blocks.
2. Shift window of certain size over query; for each database block find the number of matching $q$-Grams from the current window; if this number is at least $T(n, q, e) := n + 1 - q(e + 1)$, consider this block relevant.
3. Test the relevant blocks with a more complex method.
**QUASAR algorithm**

**Algorithm:** Filter database for relevant blocks.

1. Divide database into overlapping blocks.
2. Shift window of certain size over query; for each database block find the number of matching $q$-Grams from the current window; if this number is at least $T(n, q, e) := n + 1 - q(e + 1)$, consider this block *relevant*.
3. Test the relevant blocks with a more complex method.
**QUASAR algorithm**

**Algorithm:** Filter database for relevant blocks.

1. Divide database into overlapping blocks.

2. Shift window of certain size over query; for each database block find the number of matching $q$-Grams from the current window; if this number is at least $T(n, q, e) := n + 1 - q(e + 1)$, consider this block relevant.

3. Test the relevant blocks with a more complex method.
QUASAR algorithm

**Algorithm:** Filter database for relevant blocks.

1. Divide database into overlapping blocks.

2. Shift window of certain size over query; for each database block find the number of matching $q$-Grams from the current window; if this number is at least $T(n, q, e) := n + 1 - q(e + 1)$, consider this block *relevant*.

3. Test the relevant blocks with a more complex method.
Two ideas for improving QUASAR

(Rasmussen, JS & Myers, JCB 2006)

1. Use a simpler index structure: \textbf{q-Gram index}

\begin{center}
\begin{tabular}{|c|c|c|c|c|c|c|c|c|}
\hline
A A A & 3 & 4 & 30 & 46 & 47 & 48 & 91 & 23 \tabularnewline
A A C & 21 & 49 & 73 & 22 \tabularnewline
A A G & 5 & 73 \tabularnewline
A A T & 39 & 92 & 115 & 59 & 203 \tabularnewline
A C A & 1 & 22 & 73 \tabularnewline
A C C & 27 & 55 & 72 & 79 & 40 & 113 & 74 & 183 \tabularnewline
\vdots & \vdots \tabularnewline
T T T & 11 & 32 & 35 & 34 & 60 & 94 & 131 \tabularnewline
\hline
\end{tabular}
\end{center}

Analysis: simple $\mathcal{O}(n)$ construction time
Two ideas for improving QUASAR

(Rasmussen, JS & Myers, *JCB* 2006)

1. Use a simpler index structure: **q-Gram index**

   \[
   \begin{array}{c|cccccccccc}
   & 5 & 4 & 36 & 46 & 47 & 48 & 91 & 21 & 12 & 26 \\
   A & 31 & 49 & 72 & 22 & & & & & & \\
   A & & & & & & & & & & \\
   C & & & & & & & & & & \\
   & & & & & & & & & & \\
   A & 5 & 73 & & & & & & & & \\
   A & & & & & & & & & & \\
   G & & & & & & & & & & \\
   A & & & & & & & & & & \\
   A & 30 & 92 & 110 & 59 & 201 & & & & & \\
   T & 1 & 22 & 72 & & & & & & & \\
   A & & & & & & & & & & \\
   C & & & & & & & & & & \\
   A & & & & & & & & & & \\
   C & & & & & & & & & & \\
   & & & & & & & & & & \\
   & & & & & & & & & & \\
   T & 13 & 31 & 35 & 36 & 63 & 94 & & & & \\
   T & & & & & & & & & & \\
   T & & & & & & & & & & \\
   T & & & & & & & & & & \\
   \end{array}
   \]

   **Analysis:** simple $O(n)$ construction time

2. Reduce size of the relevant regions: parallelograms

→ need proper filter criteria
Filter criteria for parallelograms

Define $\epsilon$-match: match of database substring $\alpha$ and query substring $\beta$ of length $n \geq n_0$ with at most $e := \lfloor \epsilon n \rfloor$ errors.

Given: $q$, $n_0$, $\epsilon$

Compute: threshold $\tau$ such that for every $\epsilon$-match there exists a $w \times e$ parallelogram containing at least $\tau$ $q$-Gram hits.
The SWIFT algorithm

**Algorithm:** Filter database for relevant parallelograms.

1. Divide database into overlapping diagonal blocks of width $e$.
2. Shift window of size $w$ over query;
3. Find $w \times e$ parallelograms with more than $\tau$ $q$-grams.
4. Test these with a more complex method, e.g. $X$-drop extension.
Algorithm: Filter database for relevant parallelograms.

1. Divide database into overlapping diagonal blocks of width $e$.
2. Shift window of size $w$ over query;
3. Find $w \times e$ parallelograms with more than $\tau$ $q$-grams.
4. Test these with a more complex method, e.g. $X$-drop extension.
Algorithm: Filter database for relevant parallelograms.

1. Divide database into overlapping diagonal blocks of width $e$.
2. Shift window of size $w$ over query;
3. Find $w \times e$ parallelograms with more than $\tau$ q-grams.
4. Test these with a more complex method, e.g. $X$-drop extension.
Algorithm: Filter database for relevant parallelograms.

1. Divide database into overlapping diagonal blocks of width $e$.
2. Shift window of size $w$ over query;
3. Find $w \times e$ parallelograms with more than $\tau$ $q$-grams.
4. Test these with a more complex method, e.g. $X$-drop extension.
The SWIFT algorithm

**Algorithm:** Filter database for relevant parallelograms.

1. Divide database into overlapping diagonal blocks of width $e$.
2. Shift window of size $w$ over query;
3. Find $w \times e$ parallelograms with more than $\tau$ $q$-grams.
4. Test these with a more complex method, e.g. $X$-drop extension.
The SWIFT algorithm

Algorithm: Filter database for relevant parallelograms.

1. Divide database into overlapping diagonal blocks of width $e$.
2. Shift window of size $w$ over query;
3. Find $w \times e$ parallelograms with more than $\tau$ $q$-grams.
4. Test these with a more complex method, e.g. $X$-drop extension.
The SWIFT algorithm

**Algorithm**: Filter database for relevant parallelograms.

1. Divide database into overlapping diagonal blocks of width $e$.
2. Shift window of size $w$ over query;
3. Find $w \times e$ parallelograms with more than $\tau$ $q$-grams.
4. Test these with a more complex method, e.g. $X$-drop extension.
The SWIFT algorithm

Algorithm: Filter database for relevant parallelograms.

1. Divide database into overlapping diagonal blocks of width $e$.
2. Shift window of size $w$ over query;
3. Find $w \times e$ parallelograms with more than $\tau$ $q$-grams.
4. Test these with a more complex method, e.g. $X$-drop extension.
Algorithm: Filter database for relevant parallelograms.

1. Divide database into overlapping diagonal blocks of width $e$.
2. Shift window of size $w$ over query;
3. Find $w \times e$ parallelograms with more than $\tau$ $q$-grams.
4. Test these with a more complex method, e.g. $X$-drop extension.
The SWIFT algorithm

**Algorithm:** Filter database for relevant parallelograms.

1. Divide database into overlapping diagonal blocks of width $e$.
2. Shift window of size $w$ over query;
3. Find $w \times e$ parallelograms with more than $\tau q$-grams.
4. Test these with a more complex method, e.g. $X$-drop extension.
The SWIFT algorithm

**Algorithm:** Filter database for relevant parallelograms.

1. Divide database into overlapping diagonal blocks of width $e$.
2. Shift window of size $w$ over query;
3. Find $w \times e$ parallelograms with more than $\tau$ $q$-grams.
4. Test these with a more complex method, e.g. $X$-drop extension.
The SWIFT algorithm

**Algorithm:** Filter database for relevant parallelograms.

1. Divide database into overlapping diagonal blocks of width $e$.
2. Shift window of size $w$ over query;
3. Find $w \times e$ parallelograms with more than $\tau$ $q$-grams.
4. Test these with a more complex method, e.g. $X$-drop extension.
Experimental results

<table>
<thead>
<tr>
<th>parameters $(\epsilon, n_0)$</th>
<th>SSEARCH</th>
<th>BLASTN $(5%, 50)$</th>
<th>SWIFT $(4%, 30)$</th>
<th>SWIFT $(5%, 30)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>running time</td>
<td>8h</td>
<td>773 s</td>
<td>18 s</td>
<td>29 s</td>
</tr>
<tr>
<td>filtration ratio</td>
<td>—</td>
<td>—</td>
<td>6.5e-6</td>
<td>4.5e-6</td>
</tr>
</tbody>
</table>
Index structures in biological sequence analysis

1. Introduction
2. Suffix trees
3. Affix trees
4. Suffix arrays
5. The $q$-Gram index
6. Summary and Conclusion
Handling large amounts of DNA sequence data is challenging. Standard methods do not apply in bioinformatics. Data structures discussed: suffix tree, affix tree, suffix array, $q$-Gram index. Popular approach: filtration. Tools that use index structures: MUMmer, Genalyzer, SSAHA, BLAT, SWIFT, PatternHunter, ... Simplicity is often a key element of practical algorithms!
Acknowledgments

Suffix trees
- Robert Giegerich (Bielefeld)
- Stefan Kurtz (Hamburg)

Repeats
- Dan Gusfield (Davis)
- Enno Ohlebusch (Ulm)
- Chris Schleiermacher (Köln)
- Jomuna Choudhuri (Mannheim)

Affix trees
- Moritz Maaß (München)

Suffix arrays
- Klaus-Bernd Schürmann (Bielefeld)

SWIFT
- Kim Roland Rasmussen (Bielefeld)
- Gene Myers (HHMI, Janelia Farm)
The end.