Full-text indexes
Indexing sequence data

- Large sequence data should be stored into data structures (indexes) that support efficient and fast query and retrieval

  - **Example 1**: to look up for specific patterns, we can use an inverted index

  - **Example 2**: to look up for patterns (strings) of fixed length $k$ ($k$-mers) we can use a hash table

- What if we need to search for patterns of arbitrary length?

  - **full-text indexes** allow a search for patterns of any length occurring at any position of the text
“Ideal” index structure for $T[1..n]$

- Takes space $O(n)$
- Can be constructed in time $O(n)$
- All occurrences of a query pattern $P$ can be reported in time $O(|P| + occ)$
Suffix tree
Trie (aka digital tree)

trie for \{agca,aacaa,agcc,aa\}

- every string of the set is “spelled” starting from root
- edges outgoing from a node are labeled by different characters
- trie can be viewed as an automaton recognizing the given set of strings (or all their prefixes)
Suffix trie

$T=\text{acatacagatg}$

\begin{itemize}
  \item acatacagatg\$
  \item catacagatg\$
  \item atacagatg\$
  \item tacagatg\$
  \item acagatg\$
  \item cagatg\$
  \item agatg\$
  \item gatg\$
  \item atg\$
  \item tg\$
  \item g\$
  \item \$
\end{itemize}
Suffix trie

T = acatacagatg$

$O(n^2)$ nodes $a^n b^n$
Suffix tree

$T = \text{acatacagatg}$

- explicit vs implicit nodes
- an edge label is start and end positions of corresponding substring (rather than substring itself)
- takes space $O(n)$
Exercise

- Construct the suffix tree for string MISSISSIPPI$
- How many internal nodes does it have?
Suffix tree: applications

$T=\text{acatacagatg}\$

- check if a pattern $P$ occurs in $T$ in time $O(|P|)$. Ex: $P_1=\text{atac}$  $P_2=\text{c}$
Suffix tree: applications

$T=acatacagatg$

- check if a pattern $P$ occurs in $T$ in time $O(|P|)$. Ex: $P_1=atac$ $P_2=c$
- report the number of occurrences in $O(|P|)$ ⇒ preprocess nb of leaves
Suffix tree: applications

T = acatacagatg$

• check if a pattern $P$ occurs in $T$ in time $O(|P|)$. Ex: $P_1 = \text{atac}$  $P_2 = \text{c}$
• report the number of occurrences in $O(|P|) \Rightarrow$ preprocess nb of leaves
• report all occurrences in $O(|P| + \text{occ}) \Rightarrow$ chain leaves and preprocess leftmost and rightmost leaves
Suffix tree: applications

\( T = \text{acatacagatg}\$

- check if a pattern \( P \) occurs in \( T \) in time \( O(|P|) \). Ex: \( P_1 = \text{atac} \quad P_2 = \text{c} \)
- report the number of occurrences in \( O(|P|) \Rightarrow \) preprocess nb of leaves
- report all occurrences in \( O(|P| + \text{occ}) \Rightarrow \) chain leaves and preprocess leftmost and rightmost leaves
- report the first (leftmost) occurrence in \( O(|P|) \Rightarrow \) preprocess minimal leaf label
Suffix tree: applications

$T = \text{acatacagatg}$

- longest repeated substring $\Rightarrow$ deepest (w.r.t. string depth) internal node in the suffix tree (aca)
Suffix tree: applications

T = acatacagatg$

- longest extension queries: given two positions $i,j$, output the length of the longest common substring starting at $i,j$
- reduces to lowest common ancestor (lca) queries
- lca queries can be answered in $O(1)$ time after linear-time preprocessing of the tree [Harel, Tarjan 84], [Bender, Farach-Colton 00]
LCA and RMQ

- Lowest Common Ancestor (LCA) queries can be easily simulated by 1-dimensional Range Minimum Queries (RMQ)

Euler tour (DF traversal)  depth array

aeabfbdgdhdcica
01012123434323210
LCA and RMQ

- Lowest Common Ancestor (LCA) queries can be easily simulated by 1-dimensional Range Minimum Queries (RMQ)

LCA\((f, h) = b\)

Euler tour (DF traversal)

depth array

```
aeabfbcdgdhdcicba
01012123434323210
```
RMQ: solution in < $O(n \log n), O(1)$ >

- **Trivial solution in** < $O(n^2), O(1)$ >
  - compute and store min for all intervals

- **Improved solution in** < $O(n \log n), O(1)$ >:
  - compute and store min for all intervals of length $2^k$, $k \leq \lfloor \log n \rfloor$
  - there are $O(n \log n)$ such intervals
  - any interval is a union of two (overlapping) such intervals ⇒
    take min of the two

- pre-processing time
- query time

\[
2^k
\]
RMQ: solution in \(< O(n), O(\log n) >\)

- **Interval tree**

```
[1,16]
[1,8] [9,16]
[1,4] [5,8] [9,12] [13,16]
[1,2] [3,4] [5,6] [7,8] [9,10] [11,12] [13,14] [15,16]
[1,1] [2,2] [3,3] [4,4] [5,5] [6,6] [7,7] [8,8] [9,9] [10,10] [11,11] [12,12] [13,13] [14,14] [15,15] [16,16]
```
RMQ: solution in $< O(n), O(\log n) >$

- **Interval tree**

  - [1,16]
    - [1,8]
      - [1,4]
        - [1,2]
          - [1,1]
        - [3,4]
      - [5,6]
      - [7,8]
    - [5,8]
  - [9,16]
    - [9,12]
      - [9,10]
      - [11,12]
    - [13,16]
     - [13,14]
     - [15,16]

- **Claim:** Any interval is a union of $O(\log n)$ (disjoint) tree intervals
RMQ: solution in $< O(n), O(log n) >$

- **Interval tree**

```
[1,16]
```

```
[1,8]
```

```
[1,4]
```

```
[3,4]
```

```
[5,6]
```

```
[7,8]
```

```
[1,2]
```

```
[1,1]
```

```
[2,2]
```

```
[3,3]
```

```
[4,4]
```

```
[5,5]
```

```
[6,6]
```

```
[7,7]
```

```
[8,8]
```

```
[9,9]
```

```
[10,10]
```

```
[11,11]
```

```
[12,12]
```

```
[13,13]
```

```
[14,14]
```

```
[15,15]
```

```
[16,16]
```

- **Claim**: Any interval is a union of $O(log n)$ (disjoint) tree intervals

- **Example**: [6,13]
RMQ: solution in $< O(n), O(1) >$

- Possible!
- a little more technical, we skip it
- see [M.Bender, M.Farach-Colton, The LCA problem revisited, LATIN 2000]
Coding project

- see https://wikimpri.dptinfo.ens-cachan.fr/doku.php?id=cours:c-1-32&#exams

- deadline Nov 21
Suffix tree: some history

- Suffix tree can be constructed in time $O(n)$
- Weiner 1973: right-to-left construction
Suffix tree: some history

- Suffix tree can be constructed in time $O(n)$
- Weiner 1973: right-to-left construction
- McCreight 1976: left-to-right
- Ukkonen 1995: left-to-right and online
- Farach 1997: for integer alphabets
Suffix tree augmented with suffix links

McCreight and Ukkonen use *suffix links*

$suf\text{-}link(\overline{au})=\overline{u}$ for explicit nodes $\overline{au}$
Suffix tree augmented with suffix links

McCreight and Ukkonen use suffix links

$suf\text{-}link(\overline{au}) = \overline{u}$ for explicit nodes $\overline{au}$
Lempel-Ziv encoding

- used in compression algorithms (compress, gzip, gif, …)

abaabaaabababaabb...

a b a abaa aba baba ab b...

a b (1,1) (1,4) (1,3) (9,4) (1,2) b

Definition: $T = f_1 \ldots f_{i-1} f_i \ldots$, where $f_i$ is defined by

- if the letter $a$ following $f_1 \ldots f_{i-1}$ does not occur earlier, then $f_i = a$
- otherwise $f_i$ is the longest substring that occurs earlier (with possible overlap)
Lempel-Ziv encoding (cont)

- used in compression algorithms (compress, gzip, gif, …)

```plaintext
abaabaaabababaabb...
 a b a abaa aba baba ab b...
 a b (1,1) (1,4) (1,3) (9,4) (1,2) b
```

- compute LZ-encoding (offline) with suffix tree:
  - build suffix tree for T
  - annotate internal nodes by smallest position of a descendant leaf
  - to compute a new phrase \((i,l)\) starting at position \(p\), match \(T[p..]\) against the suffix tree as long as the min position is smaller than \(p\)
Modifications and extensions of suffix tree

- *Generalized suffix tree*: suffix tree for several strings (dictionaries)
- *Sparse suffix tree*: suffix tree for a fraction of suffixes
- *Affix trees*: suffix tree for a string and its inverse
- *Order-preserving suffix trees*: suffix trees for integer sequences allowing “order-matching search”

- *Suffix-tree-like data structures*
  - DAWG (Directed Acyclic Word Graph)
  - Position heap
Algorithmic beauty facing rude reality of big data

- pointer-based structures take too much space for genomic data ($O(n)$ computer words but $O(n \log(n))$ bits)
- naïve ST implementation requires $5 \times 32 = 160$ bits per internal node and at least 32 bits per leaf (for strings not exceeding $2^{32} \approx 4 \times 10^9$ chars)
- in practice, STs require between 70 and 200 bits/char; this results to $27 - 80$ Gb for human genome
- note that the DNA sequence itself takes 2 bits/char
- still, STs are used in practical bioinformatics programs (MUMmer)
Suffix array
Suffix array: definition

$T=\text{acatacagatg}$

\begin{align*}
\text{acatacagatg} & : 1 \\
\text{catacagatg} & : 2 \\
\text{atacagatg} & : 3 \\
\text{tacagatg} & : 4 \\
\text{acagatg} & : 5 \\
\text{cagatg} & : 6 \\
\text{agatg} & : 7 \\
\text{gatg} & : 8 \\
\text{atg} & : 9 \\
\text{tg} & : 10 \\
\text{g} & : 11 \\
\$ & : 12
\end{align*}
### Suffix array: definition

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

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>$$</td>
<td>12</td>
</tr>
<tr>
<td>acagatg$</td>
<td>5</td>
</tr>
<tr>
<td>acatacagatg$</td>
<td>1</td>
</tr>
<tr>
<td>agatg$</td>
<td>7</td>
</tr>
<tr>
<td>atacagatg$</td>
<td>3</td>
</tr>
<tr>
<td>atg$</td>
<td>9</td>
</tr>
<tr>
<td>cagatg$</td>
<td>6</td>
</tr>
<tr>
<td>catacagatg$</td>
<td>2</td>
</tr>
<tr>
<td>g$</td>
<td>11</td>
</tr>
<tr>
<td>gatg$</td>
<td>8</td>
</tr>
<tr>
<td>tacagatg$</td>
<td>4</td>
</tr>
<tr>
<td>tg$</td>
<td>10</td>
</tr>
</tbody>
</table>
Suffix array: definition

Suffix array (SA) is a permutation of the positions of $T$ taken in the lexicographical order of corresponding suffixes

$[T[SA[i]..n]]_{\text{lex}} < [T[SA[i+1]..n]}$

[Manber, Myers 90]
[Gonnet, Baeza-Yates, Snider 92]
String matching with suffix arrays

<p>| | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>acagatg$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>acatacagatg$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>agatg$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>atacagatg$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>atg$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>cagatg$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>catacagatg$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>g$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>gatg$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>tacagatg$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>tg$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\( T = \text{acatacagatg$} \)

**SA**

- For any substring, its occurrences form an interval of SA
- Interval \([L_p, R_p]\) for \(P\) can be found by two binary searches
- Search for \(L_p\):

\[
\text{while } l < r \text{ do} \\
\quad m = [(l + r)/2] \\
\quad \text{if } P >_{\text{lex}} T[SA[m]..n] \\
\quad \quad \text{then } l \leftarrow m + 1 \\
\quad \quad \text{else } r \leftarrow m
\]

- the whole search for \(P\) takes time \(O(m \cdot \log(n))\)
String matching with suffix arrays

$T = \text{acatacagatg}\$

<table>
<thead>
<tr>
<th>SA</th>
<th>LCP</th>
</tr>
</thead>
<tbody>
<tr>
<td>$$</td>
<td>12</td>
</tr>
<tr>
<td>acagatg$</td>
<td>5</td>
</tr>
<tr>
<td>acatacagatg$</td>
<td>1</td>
</tr>
<tr>
<td>agatg$</td>
<td>7</td>
</tr>
<tr>
<td>atagcagatg$</td>
<td>3</td>
</tr>
<tr>
<td>atag$</td>
<td>9</td>
</tr>
<tr>
<td>cagatg$</td>
<td>6</td>
</tr>
<tr>
<td>catacagatg$</td>
<td>2</td>
</tr>
<tr>
<td>g$</td>
<td>11</td>
</tr>
<tr>
<td>gatag$</td>
<td>8</td>
</tr>
<tr>
<td>tacagatg$</td>
<td>4</td>
</tr>
<tr>
<td>tg$</td>
<td>10</td>
</tr>
</tbody>
</table>

- **LCP[i]** = length of the longest common prefix between suffixes starting at **SA[i]** and **SA[i-1]**
- **Note**: the longest common prefix between any two suffixes **SA[i]** and **SA[j]** can be computed in constant time after a linear-time preprocessing of LCP array. (Use range minimum queries)
- using LCP array, interval search time $O(m \cdot \log(n))$ can be turned into $O(m + \log(n))$
String matching with suffix arrays

$P=\text{aacccaa}$

$suf_i=\text{aaaga...}$

$suf_r=\text{aacccac...}$

$\text{SA}$

- assume we know $lcp_l=LCP(suf_i,P)$, $lcp_r=LCP(suf_r,P)$ and assume $lcp_l<lcp_r$

notation: $suf_i=T[i..n]$
String matching with suffix arrays

$P=\text{aacccaa}$

$S=\text{aaaga...}$

$suf_l=\text{aaaga...}$

$suf_m=\text{aaccaca...}$

$suf_r=\text{aacccac...}$

$\text{lcp}_l$

$\text{lcp}_m$

$\text{lcp}_r$

$\text{SA}$

• assume we know $\text{lcp}_l=\text{LCP}(suf_l,P)$,
  $\text{lcp}_r=\text{LCP}(suf_r,P)$ and assume $\text{lcp}_l<\text{lcp}_r$

• consider $suf_m$ and consider $\text{LCP}(suf_m,suf_r)$
String matching with suffix arrays

\[ P = \text{aacccaa} \]

\[ S = \text{aaga...} \]

\[ \text{suf}_i = \text{aaga...} \]

\[ \text{suf}_m = \text{aaccaca...} \]

\[ \text{suf}_r = \text{aaccca...} \]

\[ SA \]

- assume we know \( \text{lcp}_i = \text{LCP(suf}_i, P) \), \( \text{lcp}_r = \text{LCP(suf}_r, P) \) and assume \( \text{lcp}_i < \text{lcp}_r \)
- consider \( \text{suf}_m \) and consider \( \text{LCP(suf}_m, \text{suf}_r) \)
- Case 1: \( \text{LCP(suf}_m, \text{suf}_r) < \text{lcp}_r \)
String matching with suffix arrays

- assume we know \( lcp_l = \text{LCP}(suf_l, P) \), \( lcp_r = \text{LCP}(suf_r, P) \) and assume \( lcp_l < lcp_r \)
- consider \( suf_m \) and consider \( \text{LCP}(suf_m, suf_r) \)

**Case 1:** \( \text{LCP}(suf_m, suf_r) < lcp_r \)

Then \( P >_{\text{lex}} suf_m \), \( \text{LCP}(suf_m, P) = \text{LCP}(suf_m, suf_r) \) and we set \( l := m \) for the next iteration
String matching with suffix arrays

P = aacagac
$s$

$\text{sa}_l$

$suf_i = \text{aaaga...}$

$suf_m = \text{aaccaca...}$

$suf_r = \text{aaccacac...}$

$\text{sa}_r$

$\text{sa}_m$

$\text{sa}_r$

$\text{sa}_l$

\[ \text{String matching with suffix arrays} \]

- assume we know $\text{lcp}_l = \text{LCP}(suf_i, P)$, $\text{lcp}_r = \text{LCP}(suf_r, P)$ and assume $\text{lcp}_l < \text{lcp}_r$

- consider $suf_m$ and consider $\text{LCP}(suf_m, suf_r)$

- **Case 2**: $\text{LCP}(suf_m, suf_r) > \text{lcp}_r$

Then $P <_{\text{lex}} suf_m$, $\text{LCP}(suf_m, P) = \text{lcp}_r$ and we set $r := m$ for the next iteration.
String matching with suffix arrays

\[ P = \text{aaccagc} \]

\[ \$ \]

\[ \text{suf}_l = \text{aaaga} \ldots \]

\[ \text{suf}_m = \text{aaccaca} \ldots \]

\[ \text{suf}_r = \text{aaccac}c \ldots \]

\[ \text{SA} \]

\[ \text{lcp}_l \]

\[ \text{lcp}_r \]

\[ \text{lcp}_m \]

- assume we know \( \text{lcp}_l = \text{LCP}(\text{suf}_l,P) \), \( \text{lcp}_r = \text{LCP}(\text{suf}_r,P) \) and assume \( \text{lcp}_l < \text{lcp}_r \)

- consider \( \text{suf}_m \) and consider \( \text{LCP}(\text{suf}_m,\text{suf}_r) \)

- **Case 3**: \( \text{LCP}(\text{suf}_m,\text{suf}_r) = \text{lcp}_r \)

Then we keep comparing chars of \( P \) with those of \( \text{suf}_m \) until \( P[j] \neq \text{suf}_m[j] \) which determines if \( P \leq_{\text{lex}} \text{suf}_m \) or \( P >_{\text{lex}} \text{suf}_m \), and also the value \( \text{lcp}_m = \text{LCP}(\text{suf}_m,P) \)
String matching with suffix arrays

\[ P = \text{aaccagc} \]

\[ \text{suf}_l = \text{aaaga...} \]

\[ \text{suf}_m = \text{aaccaca...} \]

\[ \text{suf}_r = \text{aaccacac...} \]

\[ \text{lcp}_l \]

\[ \text{lcp}_m \]

\[ \text{lcp}_r \]

\[ \text{SA} \]

- Assume we know \( \text{lcp}_l = \text{LCP(suf}_l, P) \), \( \text{lcp}_r = \text{LCP(suf}_r, P) \) and assume \( \text{lcp}_l < \text{lcp}_r \)
- Consider \( \text{suf}_m \) and consider \( \text{LCP(suf}_m, \text{suf}_r) \)
- **Case 3**: \( \text{LCP(suf}_m, \text{suf}_r) = \text{lcp}_r \)

Then we keep comparing chars of \( P \) with those of \( \text{suf}_m \) until \( P[j] \neq \text{suf}_m[j] \) which determines if \( P \leq_{\text{lex}} \text{suf}_m \) or \( P >_{\text{lex}} \text{suf}_m \), and also the value \( \text{lcp}_m = \text{LCP(suf}_m, P) \)
String matching with suffix arrays

$P=\text{aaccagc}$

$\$…$
suf$_i$= $\text{aaaga…}$ …

$suf$_m$=\text{aaccaca…}$ …

$suf$_r$=\text{aaccacac…}$ …

$\text{SA}$

- assume we know $\text{lcp}$_l=\text{LCP}(\text{suf}_i,P)$,
  $\text{lcp}$_r=\text{LCP}(\text{suf}_r,P)$ and assume $\text{lcp}$_l<$\text{lcp}$_r$
- consider $\text{suf}$_m and consider $\text{LCP}(\text{suf}$_m,\text{suf}$_r)$
- **Case 3:** $\text{LCP}(\text{suf}$_m,\text{suf}$_r)$=$\text{lcp}$_r

Then we keep comparing chars of $P$ with those of $\text{suf}$_m until $P[j]$$\neq$$\text{suf}$_m[j]$ which determines if $P$$\leq_{\text{lex}}$$\text{suf}$_m or $P$$>_{\text{lex}}$$\text{suf}$_m, and also the value $\text{lcp}$_m=\text{LCP}(\text{suf}$_m,P)$
String matching with suffix arrays

\[ P = \text{aaccac} \]

\[ \text{Suf}_l = \text{aaaga} \ldots \]

\[ \text{Suf}_m = \text{aaccac} \ldots \]

\[ \text{Suf}_r = \text{aaccac} \ldots \]

\[ \text{Lcp}_l = \text{LCP(suf}_l,P) \]

\[ \text{Lcp}_m = \text{LCP(suf}_m,P) \]

\[ \text{Lcp}_r = \text{LCP(suf}_r,P) \]

- Assume we know \( \text{lcp}_l = \text{LCP(suf}_l,P) \), \( \text{lcp}_r = \text{LCP(suf}_r,P) \) and assume \( \text{lcp}_l < \text{lcp}_r \).
- Consider \( \text{suf}_m \) and consider \( \text{LCP(suf}_m,suf}_r \).

**Case 3**: \( \text{LCP(suf}_m,suf}_r = \text{lcp}_r \)

Then we keep comparing chars of \( P \) with those of \( \text{suf}_m \) until \( P[j] \neq \text{suf}_m[j] \) which determines if \( P \leq_{\text{lex}} \text{suf}_m \) or \( P >_{\text{lex}} \text{suf}_m \), and also the value \( \text{lcp}_m = \text{LCP(suf}_m,P) \).

**Conclusion**: at each step we either do binary division, or move one char forward in the pattern \( \Rightarrow \) time \( O(|P| + \log(n)) \).
Construction of suffix array

- construction in $O(n \log n)$ [Manber, Myers 90]
- *bad option*: construct the suffix tree, then extract the suffix array
- construction in $O(n)$ [Kärkkäinen, Sanders 03] [Ko, Aluru 03] [Kim et al 03]
- works on practical linear-time construction of suffix array: [Nong et al 09]
Suffix array and suffix tree

T = acatacagatg$

<table>
<thead>
<tr>
<th>SA</th>
<th>LCP</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>-1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>

SA ~ leaf labels left-to-right provided that children of any node are ordered according to the order of characters

LCP ~ string length of lowest common ancestor (lca)
Enhanced suffix arrays

- LCP array encodes the tree topology of ST: internal node $\approx$ interval $\text{LCP}[i..j]$ s.t. $\min\{\text{LCP}[i+1..j]\}=q$ and $\text{LCP}[i]<q$, $\text{LCP}[j+1]<q$
- on LCP and SA arrays, one can simulate bottom-up and top-down navigation in ST, as well as suffix links [Abouelhoda et al 04]
Enhanced suffix arrays

- LCP array encodes the tree topology of ST: internal node \( \approx \) interval \( \text{LCP}[i..j] \) s.t. \( \min\{\text{LCP}[i+1..j]\}=q \) and \( \text{LCP}[i]<q, \text{LCP}[j+1]<q \)
- on LCP and SA arrays, one can simulate bottom-up and top-down navigation in ST, as well as suffix links [Abouelhoda et al 04]
Suffix arrays: practical issues

- SA is a popular data structure that provides a space-efficient alternative to ST
- Enhanced suffix arrays take 40-72 bits/char, this results to 15-27Gb for human genome
- Enhanced SAs are used in practical bioinformatics software (Vmatch, segemehl, Fiona)
- Efficient SA construction is an active area of research (external memory, parallelization, …)
- Compressed suffix arrays [Grossi, Vitter 00] (different structures surveyed in [Navarro, Makinen 07])
Burrows-Wheeler transform and BWT-index
Succinct and compressed indexes

- **succinct** index takes space *in bits* proportional to that of the text itself.
- previous indexes are not succinct as they take $O(n)$ computer words but $O(n \cdot \log(n))$ bits.

- **compressed** index takes space *in bits* proportional to that of the *compressed text*.
- **self-index** does not require storing the text.
Burrows-Wheeler transform

$T = acatacagatg$

$acatacagatg$
acagatg$acat$
acatacagatg$
agatg$acatac$
atacagatg$ac$
atg$acatacag$
cagatg$acata$
catacagatg$a$
g$acatacagat$
gatg$acataca$
tacagatg$aca$
tg$acatacaga
## Burrows-Wheeler transform

$$T = \text{acatacagatg}$${

<table>
<thead>
<tr>
<th>String</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{acatacagatg}$</td>
<td>12</td>
</tr>
<tr>
<td>$\text{acagatg}\text{acat}$</td>
<td>5</td>
</tr>
<tr>
<td>$\text{acatacagatg}$</td>
<td>1</td>
</tr>
<tr>
<td>$\text{agatg}\text{acatac}$</td>
<td>7</td>
</tr>
<tr>
<td>$\text{atacagatg}\text{ac}$</td>
<td>3</td>
</tr>
<tr>
<td>$\text{atg}\text{acatacag}$</td>
<td>9</td>
</tr>
<tr>
<td>$\text{cagatg}\text{acata}$</td>
<td>6</td>
</tr>
<tr>
<td>$\text{catacagatg}\text{a}$</td>
<td>2</td>
</tr>
<tr>
<td>$\text{g}\text{acatacagat}$</td>
<td>11</td>
</tr>
<tr>
<td>$\text{gatg}\text{acataca}$</td>
<td>8</td>
</tr>
<tr>
<td>$\text{tacagatg}\text{aca}$</td>
<td>4</td>
</tr>
<tr>
<td>$\text{tg}\text{acatacaga}$</td>
<td>10</td>
</tr>
</tbody>
</table>
Burrows-Wheeler transform

\[ T = \acata\catagagatg\$ \]

\[
\begin{align*}
& \acata\catagagatg \\
& \acagatg\$\acat \\
& \acata\catagagatg\$ \\
& \acata\catagagatg\$ \\
& \acata\catagagatg\$ \\
& \acata\catagagatg\$ \\
& \acata\catagagatg\$ \\
& \acata\catagagatg\$ \\
& \acata\catagagatg\$ \\
& \acata\catagagatg\$ \\
& \acata\catagagatg\$ \\
& \acata\catagagatg\$ \\
& \acata\catagagatg\$ \\
& \acata\catagagatg\$ \\
& \acata\catagagatg\$ \\
& \acata\catagagatg\$ \\
& \acata\catagagatg\$ \\
\end{align*}
\]
Burrows-Wheeler transform

T=acatacagatg$

$ acatacagat g
a cagatg$aca t
a catacagatg $
$a gatg$acata c$
$a tacagatg$a c$
a tg$acataca g$
c agatg$acat a$
c atacagatg$ a$
g $acatacaga t$
g atg$acatac a$
t acagatg$ac a$
t g$acatacag a
**Burrows-Wheeler transform**

$T=\text{acatacagatg}\$

<table>
<thead>
<tr>
<th>$T[SA[i]]$</th>
<th><strong>BWT</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>1 $ \text{acatacagat}$</td>
<td>$\text{g}$</td>
</tr>
<tr>
<td>2 a cagatg$\text{aca}$</td>
<td>$\text{t}$</td>
</tr>
<tr>
<td>3 a catacagatg</td>
<td>$\text{g}$</td>
</tr>
<tr>
<td>4 a gatg$\text{acata}$</td>
<td>$\text{c}$</td>
</tr>
<tr>
<td>5 a tacagatg$\text{a}$</td>
<td>$\text{c}$</td>
</tr>
<tr>
<td>6 a tg$\text{acataca}$</td>
<td>$\text{g}$</td>
</tr>
<tr>
<td>7 c agatg$\text{acat}$</td>
<td>$\text{a}$</td>
</tr>
<tr>
<td>8 c atacagatg$$</td>
<td>$\text{a}$</td>
</tr>
<tr>
<td>9 g $\text{acatacaga}$</td>
<td>$\text{t}$</td>
</tr>
<tr>
<td>10 g atg$\text{acatac}$</td>
<td>$\text{a}$</td>
</tr>
<tr>
<td>11 t acagatg$\text{ac}$</td>
<td>$\text{a}$</td>
</tr>
<tr>
<td>12 t g$\text{acatacag}$</td>
<td>$\text{a}$</td>
</tr>
</tbody>
</table>

- $\text{BWT}[i]=T[SA[i]-1]$ if $\text{SA}[i] \neq 1$, otherwise $\$
- $\text{BWT}$ has been defined for the purpose of compression, as $\text{BWT}$ compresses better than the input text
- $\text{BWT}$ is reversible!
# Burrows-Wheeler transform

\( T = \text{acatacagatg} $ \)

## \( T[SA[i]] \) vs. BWT

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>BWT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$</td>
<td>g</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>t</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
<td>$</td>
</tr>
<tr>
<td>4</td>
<td>a</td>
<td>c</td>
</tr>
<tr>
<td>5</td>
<td>a</td>
<td>c</td>
</tr>
<tr>
<td>6</td>
<td>a</td>
<td>g</td>
</tr>
<tr>
<td>7</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>8</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>9</td>
<td>g</td>
<td>t</td>
</tr>
<tr>
<td>10</td>
<td>g</td>
<td>a</td>
</tr>
<tr>
<td>11</td>
<td>t</td>
<td>a</td>
</tr>
<tr>
<td>12</td>
<td>t</td>
<td>a</td>
</tr>
</tbody>
</table>

- \( \text{BWT}[i] = T[SA[i]-1] \) if \( SA[i] \neq 1 \), otherwise $ $

- **Obs 1**: the first column (F) is easy to reconstruct, it can be represented by an array \( C[x] = \sum_{y<x} |\text{occ}(y, T)| \) for each letter \( x \)
  - Ex: \( C = [0, 1, 6, 8, 10] \)

\[
\begin{array}{ccccccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12
\end{array}
\]
**Burrows-Wheeler transform**

\[ T = \]

<table>
<thead>
<tr>
<th>T[SA[i]]</th>
<th>BWT</th>
<th>BWT[i]=T[SA[i]-1] if SA[i]≠1, otherwise $</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  $</td>
<td>g</td>
<td></td>
</tr>
<tr>
<td>2  a</td>
<td>t</td>
<td></td>
</tr>
<tr>
<td>3  a</td>
<td>g</td>
<td></td>
</tr>
<tr>
<td>4  $</td>
<td>t</td>
<td></td>
</tr>
<tr>
<td>5  c</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>6  c</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>7  g</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>8  g</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>9  a</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>10 a</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>11 t</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>12 t</td>
<td>a</td>
<td></td>
</tr>
</tbody>
</table>

**Obs 1:** the first column (F) is easy to reconstruct, it can be represented by an array \( C[x]=\sum_{y<x}|\text{occ}(y,T)| \) for each letter \( x \)

- Ex: \( C=[0,1,6,8,10] \)
Burrows-Wheeler transform

\[
T = \$
\]

<table>
<thead>
<tr>
<th>T[SA[i]]</th>
<th>BWT</th>
<th>( \text{BWT[i]} = T[\text{SA[i]}-1] ) if ( \text{SA[i]} \neq 1 ), otherwise $</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$</td>
<td>(*)</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>g</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
<td>t</td>
</tr>
<tr>
<td>4</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>g</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>g</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>a</td>
<td></td>
</tr>
</tbody>
</table>

- **Obs 1**: the first column (F) is easy to reconstruct, it can be represented by an array \( C[x] = \sum_{y<x} |\text{occ}(y, T)| \) for each letter \( x \)
  - Ex: \( C = [0, 1, 6, 8, 10] \)
Burrows-Wheeler transform

\[ T = g$ \]

- \( T[SA[i]] \)

**Obs 1**: the first column (F) is easy to reconstruct, it can be represented by an array
\[ C[x] = \sum_{y<x} |occ(y,T)| \]
for each letter \( x \)

- Ex: \( C = [0, 1, 6, 8, 10] \)

- \( BWT[i] = T[SA[i] - 1] \) if \( SA[i] \neq 1 \), otherwise $
**Burrows-Wheeler transform**

\[
T = g$

\[
T[SA[i]] \quad BWT
\]

- \( BWT[i] = T[SA[i]-1] \) if \( SA[i] \neq 1 \), otherwise $g$

- **Obs 1**: the first column (F) is easy to reconstruct, it can be represented by an array \( C[x] = \sum_{y\leq x} |occ(y, T)| \) for each letter \( x \)
  - Ex: \( C = [0, 1, 6, 8, 10] \)

- **Obs 2**: for identical chars, their relative order in F and L is the same

\[
LF[i] = C[BWT[i]] + rank[BWT[i], i]
\]

Ex: \( LF[1] = 8 + 1 \)
**Burrows-Wheeler transform**

$$T = \text{tg}$$

| T[SA[i]] | BWT | Obs 1: the first column (F) is easy to reconstruct, it can be represented by an array $C[x] = \sum_{y<x} |\text{occ}(y, T)|$ for each letter $x$ | Obs 2: for identical chars, their relative order in F and L is the same |
|---|---|---|---|
| $|$ | $g$ | BWT[i] = T[SA[i]-1] if SA[i] ≠ 1, otherwise $|$ |
| a | t | | |
| a | c | | |
| a | c | | |
| a | g | | |
| a | c | | |
| a | g | | |
| a | t | | |
| g | t | | |
| a | a | | |
| a | a | | |
| a | a | | |

**Obs 1**: $F$ is easy to reconstruct, it can be represented by an array $C[x] = \sum_{y<x} |\text{occ}(y, T)|$ for each letter $x$.

- Example: $C = [0, 1, 6, 8, 10]$

**Obs 2**: For identical chars, their relative order in $F$ and $L$ is the same.

$LF[i] = C[BWT[i]] + \text{rank}[BWT[i], i]$

Example: $LF[1] = 8 + 1$
**Burrows-Wheeler transform**

\[ T = \text{tg$\_\$} \]

- \[ T[SA[i]] = T[SA[i]-1] \text{ if } SA[i] \neq 1, \text{ otherwise } \$

- **Obs 1**: the first column (F) is easy to reconstruct, it can be represented by an array \( C[x] = \sum_{y<x} |\text{occ}(y,T)| \) for each letter \( x \)
  - **Ex**: \( C = [0, 1, 6, 8, 10] \)

- **Obs 2**: for identical chars, their relative order in F and L is the same

\[
\begin{align*}
\text{LF}[i] &= C[BWT[i]] + \text{rank}[BWT[i], i] \\
\text{Ex}: \text{LF}[1] &= 8 + 1
\end{align*}
\]
Burrows-Wheeler transform

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

\[ T[SA[i]] \]

\[ \text{BWT} \]

- \[ \text{BWT}[i] = T[SA[i]-1] \] if \( SA[i] \neq 1 \), otherwise \$

- \textbf{Obs 1}: the first column (F) is easy to reconstruct, it can be represented by an array \( C[x] = \sum_{y<x} |\text{occ}(y,T)| \) for each letter \( x \)
  - \textit{Ex}: \( C = [0,1,6,8,10] \)
- \textbf{Obs 2}: for identical chars, their relative order in F and L is the same
  \[ LF[i] = C[\text{BWT}[i]] + \text{rank}[\text{BWT}[i],i] \]
  \textit{Ex}: \( LF[1] = 8 + 1 \)
Burrows-Wheeler transform

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

\[ T[SA[i]] \]

BWT

- \[ BWT[i] = T[SA[i]-1] \text{ if } SA[i] \neq 1, \text{ otherwise } \$

- \text{Obs 1: the first column (F) is easy to reconstruct, it can be represented by an array } C[x] = \sum_{y<x} \text{occ}(y,T) \text{ for each letter } x
  
  - \text{Ex: } C = [0,1,6,8,10]

- \text{Obs 2: for identical chars, their relative order in F and L is the same}

\[ LF[i] = C[BWT[i]] + \text{rank}[BWT[i],i] \]

\text{Ex: } LF[1] = 8 + 1
Burrows-Wheeler transform

T = gatg$

T[SA[i]]

1 2 3 4 5 6 7 8 9 10 11 12
F a a a a c c a g a t g t a
L t a a t

BWT

$ g t s c c g a a c g a c g a t$

- BWT[i] = T[SA[i] - 1] if SA[i] ≠ 1, otherwise $

- **Obs 1**: the first column (F) is easy to reconstruct, it can be represented by an array C[x] = \( \sum_{y<x} |\text{occ}(y,T)| \) for each letter x
  - Ex: C = [0, 1, 6, 8, 10]
- **Obs 2**: for identical chars, their relative order in F and L is the same

\[ LF[i] = C[BWT[i]] + \text{rank}[BWT[i], i] \]

Ex: LF[1] = 8 + 1
LF function

\[ T = \text{acatacagatg}\$

\[
\begin{array}{c|c|c}
\text{T[SA[i]]} & \text{BWT} & \text{SA} \\
\hline
1 & $ & g \\
2 & a & t \\
3 & a & $ \\
4 & a & c \\
5 & a & c \\
6 & a & g \\
7 & c & a \\
8 & c & a \\
9 & g & t \\
10 & g & a \\
11 & t & a \\
12 & t & a \\
\end{array}
\]

\[ \text{LF}[i] = C[\text{BWT}[i]] + \text{rank}[\text{BWT}[i], i] \]

- \( \text{LF}[i] \) yields the index (in SA) of the suffix immediately preceding (in \( T \)) the \( i \)-th suffix (in SA). Formally, \( \text{SA}[\text{LF}[i]] = \text{SA}[i] - 1 \).

\[ \text{SA}[\text{LF}[7]] = \text{SA}[7] - 1 = 6 - 1 = 5 \]
**rank function**

\[
T[SA[i]] \quad \text{BWT}
\]

\[
\begin{align*}
\$ & \text{ acatacagat } & g \\
a & \text{ cagatg$aca } & t \\
a & \text{ catacagatg } & $ \\
a & \text{ gatg$acata } & c \\
a & \text{ tacagatg$a } & c \\
a & \text{ tg$acataca } & g \\
c & \text{ agatg$acat } & a \\
c & \text{ atacagatg$ } & a \\
g & \text{ $acatacaga } & t \\
g & \text{ atg$acatac } & a \\
t & \text{ acagatg$ac } & a \\
t & \text{ g$acatacag } & a \\
F & L
\end{align*}
\]

LF\[i]=C[BWT[i]]+\text{rank}[BWT[i],i]
### rank function

<table>
<thead>
<tr>
<th>$T[SA[i]]$</th>
<th>BWT</th>
<th>$rank[BWT[i],i]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$$</td>
<td>acatacagat</td>
<td>g</td>
</tr>
<tr>
<td>a</td>
<td>cagatg$</td>
<td>aca</td>
</tr>
<tr>
<td>a</td>
<td>catacagatg</td>
<td>$$</td>
</tr>
<tr>
<td>a</td>
<td>gatg$</td>
<td>acata</td>
</tr>
<tr>
<td>a</td>
<td>tacagatg$</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>tg$</td>
<td>acataca</td>
</tr>
<tr>
<td>c</td>
<td>agatg$</td>
<td>acat</td>
</tr>
<tr>
<td>c</td>
<td>atacagatg$</td>
<td>a</td>
</tr>
<tr>
<td>g</td>
<td>$</td>
<td>acatacaga</td>
</tr>
<tr>
<td>g</td>
<td>atg$</td>
<td>acatca</td>
</tr>
<tr>
<td>t</td>
<td>acagatg$</td>
<td>ac</td>
</tr>
<tr>
<td>t</td>
<td>g$</td>
<td>acatacag</td>
</tr>
<tr>
<td>F</td>
<td>L</td>
<td></td>
</tr>
</tbody>
</table>
**rank function**

<table>
<thead>
<tr>
<th>T[SA[i]]</th>
<th>BWT</th>
<th>rank[BWT[i],i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>g</td>
<td>0</td>
</tr>
<tr>
<td>a</td>
<td>cagatg$aca</td>
<td>t</td>
</tr>
<tr>
<td>a</td>
<td>catacagatg</td>
<td>$</td>
</tr>
<tr>
<td>a</td>
<td>gatg$acata</td>
<td>c</td>
</tr>
<tr>
<td>a</td>
<td>tacagatg$a</td>
<td>c</td>
</tr>
<tr>
<td>a</td>
<td>tg$acataca</td>
<td>g</td>
</tr>
<tr>
<td>c</td>
<td>agatg$acat</td>
<td>a</td>
</tr>
<tr>
<td>c</td>
<td>atacagatg$</td>
<td>a</td>
</tr>
<tr>
<td>g</td>
<td>$acatacaga</td>
<td>t</td>
</tr>
<tr>
<td>g</td>
<td>atg$acatac</td>
<td>a</td>
</tr>
<tr>
<td>t</td>
<td>acagatg$ac</td>
<td>a</td>
</tr>
<tr>
<td>t</td>
<td>g$acatacag</td>
<td>a</td>
</tr>
<tr>
<td>F</td>
<td>L</td>
<td></td>
</tr>
</tbody>
</table>

$$\text{LF}[i] = C[BWT[i]] + \text{rank}[BWT[i],i]$$

how about general queries

$$\text{rank}[a,i]$$ for any letter $$a$$ and any position $$i$$?
rank/select functions

- given a string $T$, efficiently answer queries $\text{rank}(a,i)$ on the number of $a$’s in $T[1..i]$
- $\text{rank}$ function (on bit vectors) turns out to be a fundamental algorithmic block for building succinct data structures [Jacobson 89]
- on bit vectors $\text{rank}$ can be supported in time $O(1)$ using $o(n)$ additional $bits$ of memory
- complementary function $\text{select}(a,j)$: output the position of the $j$-th occurrence of $a$ in $T$. $\text{select}$ can also be supported in $O(1)$ time
- on large alphabets, $\text{rank/select}$ can be supported in $O(\log|A|)$ time using wavelet trees with $O(n \log|A|)$ additional bits