Full-text indexes for genomic data

Gregory Kucherov

LIGM/CNRS Marne-la-Vallée
Data deluge in genomics

- Data flow in DNA sequencing surpassed the flow of advances in hardware and storage
- Moore law (chip performance doubling every two years) vs volume of sequencing data (x1.8 every year)
- Next-generation sequencing technologies, appeared in mid 00s, generate of order $10^{10}$ B (=10GB) per run
- For comparison: the sequencing of human genome (2001) used ~24GB which took several years
- Nowadays, a typical sequencing project may use 150-200 GB of sequence data:
  - *In silico* assembly of Panda genome\(^1\) (Nature 2010) was done with $56 \times 2.4$GB$\approx$135GB of sequence
  - assembly of Gorilla genome (Nature 2012) used 167Gbp of reads
• NGS produces millions (1M-1G) of short reads (35-400bp) of DNA in days
What is a full-text index?

- Text index = a data structure built from a given text (string, sequence) that supports certain type of (typically pattern look-up) queries.

- Genomic sequences don’t have word structure (cf inverted indexes) and query size is arbitrary (cf hash tables).

- **Full-text indexes** allow a search for patterns of **any length** occurring at **any position** of the text.
Plan

- “Classical” indexes
  - Suffix trees
  - DAWG and Position heaps
  - Suffix arrays

- Succinct (compressed) indexes
  - Burrows-Wheeler transform and FM-index
“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 $O(|P| + \text{occ})$
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=acatacagatg$

acatacagatg$ 1
   catacagatg$ 2
      atacagatg$ 3
          tacagatg$ 4
              acagatg$ 5
                 cagatg$ 6
                    agatg$ 7
                       gatg$ 8
                          atg$ 9
                             tg$ 10
                                g$ 11
                                   $ 12
Suffix tree

$T = acataca$ g$

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

$suf-link(\overline{au})=\overline{u}$
suf-link(au)=u
Suffix tree: applications

T = acatacagatg$

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

T = acatacagatg$

- longest repeated substring ⇒ 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]
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 and position heap
DAWG: Directed Acyclic Word Graph

- $T[1..n]$, sub($T$) set of substrings of $T$

- For $u, v \in \text{sub}(T)$, define $u \approx v$ iff $u$ and $v$ have the same end positions in $T$
  - Example: for $T=acatacagatg\$, $aca \approx ca$, $at \approx t$, $ga \approx caga$

- DAWG $\mathcal{A}$ is a directed graph $(V,E)$, where $V=\{[u]_\approx | u \in \text{sub}(T)\}$, $E=\{([u]_\approx, [ua]_\approx) | ua \in \text{sub}(T)\}$
**DAWG: Directed Acyclic Word Graph**

- $T[1..n]$, $\text{sub}(T)$ set of substrings of $T$

- for $u,v \in \text{sub}(T)$, define $u \equiv v$ iff $u$ and $v$ have the same end positions in $T$
  - $Ex$: for $T=acatacagatg$, $aca \equiv ca$, $at \equiv t$, $ga \equiv caga$

- DAWG $\mathcal{A}$ is a directed graph $(V,E)$, where $V=\{[u]_\equiv | u \in \text{sub}(T)\}$, $E=\{([u]_\equiv,[ua]_\equiv) | ua \in \text{sub}(T)\}$

$T=acatacagatg$
DAWG has remarkable properties

[Blumer et al 85] [Crochemore 86]
- DAWG is the \textit{minimal automaton} recognizing all suffixes of $T$

$T=\text{acatacagatg}\$

T=acatacagatg$
DAWG has remarkable properties

[Blumer et al 85] [Crochemore 86]

- DAWG is the **minimal automaton** recognizing all suffixes of $T$
- DAWG has at most $2n$ nodes and $3n$ edges

$T = \text{acatacagatg}$

![Diagram of DAWG]
DAWG has remarkable properties

[Blumer et al 85] [Crochemore 86]

- DAWG is the *minimal automaton* recognizing all suffixes of $T$
- DAWG has at most $2n$ nodes and $3n$ edges
- DAWG can be seen as resulting from a different compression of suffix trie

$$T=\text{acatacagatg}\$$
DAWG has remarkable properties

[Blumer et al 85] [Crochemore 86]

- DAWG is the **minimal automaton** recognizing all suffixes of $T$
- DAWG has at most $2n$ nodes and $3n$ edges
- DAWG can be seen as resulting from a different compression of suffix trie
- DAWG can be constructed in time $O(n)$ **online**

$T = \text{acatacagatg}$$^\$
DAWG has remarkable properties

[Blumer et al 85] [Crochemore 86]

- DAWG is the \textit{minimal automaton} recognizing all suffixes of \( T \)
- DAWG has at most \( 2n \) nodes and \( 3n \) edges
- DAWG can be seen as resulting from a different compression of suffix trie
- DAWG can be constructed in time \( O(n) \) \textit{online}
- DAWG can be used as a text index to answer matching queries in time \( O(|P|) \)

\[ T = \text{acatacagatg$} \]
DAWG has remarkable properties

[Blumer et al 85] [Crochemore 86]

- DAWG is the **minimal automaton** recognizing all suffixes of $T$
- DAWG has at most $2n$ nodes and $3n$ edges
- DAWG can be seen as resulting from a different compression of suffix trie
- DAWG can be constructed in time $O(n)$ *online*
- DAWG can be used as a text index to answer matching queries in time $O(|P|)$
- DAWG can be augmented with suffix links and can be used as a matching automaton

$T=$acatacagatg$\$$
DAWG has remarkable properties

- DAWG is the **minimal automaton** recognizing all suffixes of $T$
- DAWG has at most $2n$ nodes and $3n$ edges
- DAWG can be seen as resulting from a different compression of suffix trie
- DAWG can be constructed in time $O(n)$ **online**
- DAWG can be used as a text index to answer matching queries in time $O(|P|)$
- DAWG can be augmented with suffix links and can be used as a matching automaton
- The structure of suffix links of DAWG for $T$ is the same as the structure of suffix tree for the inverse of $T$

$T = \text{acatacagatg}\$
DAWG has remarkable properties

[Blumer et al 85] [Crochemore 86]

- DAWG is the **minimal automaton** recognizing all suffixes of T
- DAWG has at most 2n nodes and 3n edges
- DAWG can be seen as resulting from a different compression of suffix trie
- DAWG can be constructed in time $O(n)$ online
- DAWG can be used as a text index to answer matching queries in time $O(|P|)$
- DAWG can be augmented with suffix links and can be used as a matching automaton
- The structure of suffix links of DAWG for T is the same as the structure of suffix tree for the inverse of T
- DAWG can be extended to (dynamic) dictionaries

T=acatacagatg$
DAWG has remarkable properties

[Blumer et al 85] [Crochemore 86]

- DAWG is the **minimal automaton** recognizing all suffixes of $T$
- DAWG has at most $2n$ nodes and $3n$ edges
- DAWG can be seen as resulting from a different compression of suffix trie
- DAWG can be constructed in time $O(n)$ *online*
- DAWG can be used as a text index to answer matching queries in time $O(|P|)$
- DAWG can be augmented with suffix links and can be used as a matching automaton
- The structure of suffix links of DAWG for $T$ is the same as the structure of suffix tree for the inverse of $T$
- DAWG can be extended to (dynamic) dictionaries
- DAWG can be further compressed by compacting non-branching paths (like suffix tree)

$T=acatacagatg$
Sequence hash tree [Coffman&Eve 70]

\[ W = \{\text{caccaac, ccac, ac, caaa, aacac, cacaaca}\} \]

There are 6 sequences, labeled \( 1 \) to \( 6 \).

How to insert \( w_i \):

Find the longest prefix \( v \) of \( w_i \) which is represented in the trie. Let \( v \) is followed by char \( a \) in \( w_i \) (i.e. \( w_i=vat \)). Then create a new child node of \( v \) labeled by \( i \) and connected to \( v \) by an edge labeled by \( a \).
Position heap

$T \xrightarrow{\text{suffixes}(T)} \text{compacted trie of suffixes}(T) = \text{suffix tree of } T$

$\xrightarrow{\text{sequence hash tree of suffixes}(T) = \text{position heap of } T}$

Sequence hash tree [Coffman&Eve 70]

$W = \{\text{caccaac, ccac, ac, caaa, aacac, cacaaca}\}$

1 2 3 4 5 6

How to insert $w_i$:
Find the longest prefix $v$ of $w_i$ which is represented in the trie. Let $v$ be followed by char $a$ in $w_i$ (i.e. $w_i=\text{vat}$). Then create a new child node of $v$ labeled by $i$ and connected to $v$ by an edge labeled by $a$. 
**Position heap**

\[ T \rightarrow \text{suffixes}(T) \rightarrow \text{compacted trie of suffixes}(T) = \text{suffix tree of } T \]

\[ \text{sequence hash tree of suffixes}(T) = \text{position heap of } T \]

Sequence hash tree [Coffman&Eve 70]

\[ W = \{\text{caccaac, ccac, ac, caaa, aacac, cacaaca}\} \]

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & 6 \\
\end{array}
\]

**How to insert \( w_i \):**

Find the longest prefix \( v \) of \( w_i \) which is represented in the trie. Let \( v \) is followed by char a in \( w_i \) (i.e. \( w_i=\text{vat} \)). Then create a new child node of \( v \) labeled by \( i \) and connected to \( v \) by an edge labeled by a.
Position heap

T $\rightarrow$ suffixes(T) $\rightarrow$ compacted trie of suffixes(T) = suffix tree of T

T $\rightarrow$ suffixes(T) $\rightarrow$ sequence hash tree of suffixes(T) = position heap of T

Sequence hash tree [Coffman&Eve 70]

$W = \{\text{caccaac, ccac, ac, caaa, aacac, cacaaca}\}$

1 2 3 4 5 6

How to insert $w_i$:
Find the longest prefix $v$ of $w_i$ which is represented in the trie. Let $v$ is followed by char a in $w_i$ (i.e. $w_i=vat$). Then create a new child node of $v$ labeled by $i$ and connected to $v$ by an edge labeled by a.
Position heap

T $\rightarrow$ suffixes(T)  

compacted trie of suffixes(T) = suffix tree of T

sequence hash tree of suffixes(T) = position heap of T

Sequence hash tree [Coffman&Eve 70]

$W = \{ \text{caccaac, ccac, ac, caaa, aacac, cacaaca} \}$

How to insert $w_i$:
Find the longest prefix $v$ of $w_i$ which is represented in the trie. Let $v$ is followed by char a in $w_i$ (i.e. $w_i=vat$). Then create a new child node of $v$ labeled by $i$ and connected to $v$ by an edge labeled by a.
Position heap

\[ T \rightarrow \text{suffixes}(T) \]

\[ \text{compacted trie of suffixes}(T) = \text{suffix tree of } T \]

\[ \text{sequence hash tree of suffixes}(T) = \text{position heap of } T \]

Sequence hash tree [Coffman&Eve 70]

\[ W = \{ \text{cacaac, ccac, ac, caaa, aacac, cacaaca} \} \]

How to insert \( w_i \):

Find the longest prefix \( v \) of \( w_i \) which is represented in the trie. Let \( v \) is followed by char a in \( w_i \) (i.e. \( w_i=\text{vat} \)). Then create a new child node of \( v \) labeled by \( i \) and connected to \( v \) by an edge labeled by a.
Position heap

\[ T \rightarrow \text{suffixes}(T) \rightarrow \text{compacted trie of suffixes}(T) = \text{suffix tree of } T \]

\[ \text{sequence hash tree of suffixes}(T) = \text{position heap of } T \]

Sequence hash tree [Coffman&Eve 70]

\[ W = \{\text{caccaac}, \text{ccac}, \text{ac}, \text{caaa}, \text{aacac}, \text{cacaaca}\} \]

1 2 3 4 5 6

How to insert \( w_i \):
Find the longest prefix \( \nu \) of \( w_i \) which is represented in the trie. Let \( \nu \) is followed by char a in \( w_i \) (i.e. \( w_i = \nu \text{a} \)). Then create a new child node of \( \nu \) labeled by \( i \) and connected to \( \nu \) by an edge labeled by a.
Position heap

\[ T \xrightarrow{\text{suffixes}(T)} \text{compacted trie of suffixes}(T) = \text{suffix tree of } T \]

\[ \text{sequence hash tree of suffixes}(T) = \text{position heap of } T \]

Sequence hash tree [Coffman&Eve 70]

\( W = \{ \text{caccaac, ccac, ac, caaa, aacac, cacaaca} \} \)

1 2 3 4 5 6

How to insert \( w_i \):
Find the longest prefix \( v \) of \( w_i \) which is represented in the trie. Let \( v \) is followed by char a in \( w_i \) (i.e. \( w_i = \text{vat} \)). Then create a new child node of \( v \) labeled by \( i \) and connected to \( v \) by an edge labeled by a.

Assumption: no \( w_i \) is a prefix of another \( w_j \)
position heap of $T = sequence hash tree$ of suffixes$(T)$

suffixes are inserted right to left [Ehrenfeucht et al, 09]
suffixes are inserted left to right [Kucherov 11]
Position heap

position heap of $T = \text{sequence hash tree of suffixes}(T)$

suffixes are inserted right to left [Ehrenfeucht et al, 09]
suffixes are inserted left to right [Kucherov 11]

$T = \text{acatacagatg}$

```
\begin{array}{c}
\text{acatacagatg}\$ & 1 \\
catacagatg\$ & 2 \\
atacagatg\$ & 3 \\
tacagatg\$ & 4 \\
acagatg\$ & 5 \\
cagatg\$ & 6 \\
agatg\$ & 7 \\
gatg\$ & 8 \\
agat\$ & 9 \\
tg\$ & 10 \\
g\$ & 11 \\
\$ & 12 \\
\end{array}
```
Augmented position heaps

For string matching, we will need the following operations performed in O(1) time:

- given number $i$, retrieve node $i$
- given two nodes $i$ and $j$, check if $i$ is an ancestor of $j$
- for a node $i$, retrieve the node (its descendant) corresponding to the longest substring of $T$ starting at position $i$ $\Rightarrow$ maximal-reach pointers
Two useful properties:

i. Given a node \( \overline{u} \) and a position (node) \( i \)
   - if node \( i \) is inside the subtree rooted at \( \overline{u} \), then \( u \) occurs at position \( i \)
   - if node \( i \) is an ancestor of node \( \overline{u} \), then \( u \) occurs at \( i \) iff \( \text{max-reach-pointer}(i) \) is into the subtree rooted at \( \overline{u} \)

ii. Assume \( u \) is represented in the heap and \( u a \) is not. Then \( ua \) has at most \( |u| \) occurrences in \( T \) at positions \( i \) stored on the path from root to \( \overline{u} \)
How to search for a pattern \( P \). Two cases:

1. If \( P \) is represented in the heap, then \( P \) occurs at all positions stored in the subtree of \( \overline{P} \) and those positions stored on the path from root to \( \overline{P} \) with max-reach-pointer into the subtree of \( \overline{P} \). All positions can be reported in time \( O(|P| + \text{occ}) \).

2. Otherwise process \( P \) by chunks:
   1. find the longest proper prefix \( P[1..m] \) represented in the heap
   2. build the list of those positions on the path to \( \overline{P[1..m]} \) at which \( P[1..m] \) actually occurs (at most \( m \) such positions)
   3. iteratively proceed to \( P[m+1..] \) to check if it occurs at positions \( (i+m) \) where \( i \) is a position from the current list.
String matching on position heaps

T=acatacagatg$

*Example:* P=acata

---

How to search for a pattern $P$. Two cases:

1. If $P$ is represented in the heap, then $P$ occurs at all positions stored in the subtree of $\overline{P}$ and those positions stored on the path from root to $\overline{P}$ with **max-reach-pointer** into the subtree of $\overline{P}$. All positions can be reported in time $O(|P|+occ)$

2. Otherwise process $P$ by chunks:
   1. find the longest proper prefix $P[1..m]$ represented in the heap
   2. build the list of those positions on the path to $\overline{P}[1..m]$ at which $P[1..m]$ actually occurs (at most $m$ such positions)
   3. iteratively proceed to $P[m+1..]$ to check if it occurs at positions $(i+m)$ where $i$ is a position from the current list
String matching on position heaps

T=acatacagatg$

**Example:** P=acata
{1,5}

How to search for a pattern P. Two cases:

1. If P is represented in the heap, then P occurs at all positions stored in the subtree of $P$ and those positions stored on the path from root to $P$ with max-reach-pointer into the subtree of $P$. All positions can be reported in time $O(|P| + \text{occ})$

2. Otherwise process P by chunks:
   1. find the longest proper prefix $P[1..m]$ represented in the heap
   2. build the list of those positions on the path to $P[1..m]$ at which $P[1..m]$ actually occurs (at most $m$ such positions)
   3. iteratively proceed to $P[m+1..]$ to check if it occurs at positions $(i+m)$ where $i$ is a position from the current list
String matching on position heaps

T=acatacagatg$

**Example:** P=acata

$\{1,5\} \rightarrow \{3,7\}$

How to search for a pattern P. Two cases:

1. If P is represented in the heap, then P occurs at all positions stored in the subtree of $\overline{P}$ and those positions stored on the path from root to $\overline{P}$ with **max-reach-pointer** into the subtree of $\overline{P}$. All positions can be reported in time $O(|P|+\text{occ})$

2. Otherwise process P by chunks:
   1. find the longest proper prefix $P[1..m]$ represented in the heap
   2. build the list of those positions on the path to $\overline{P[1..m]}$ at which $P[1..m]$ actually occurs (at most $m$ such positions)
   3. iteratively proceed to $P[m+1..]$ to check if it occurs at positions $(i+m)$ where $i$ is a position from the current list
String matching on position heaps

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

**Example:** \( P = \text{acata} \)

\{1,5\} \(\Rightarrow\) \{3,7\} \(\Rightarrow\) \{5\}

How to search for a pattern \( P \). Two cases:

1. If \( P \) is represented in the heap, then \( P \) occurs at all positions stored in the subtree of \( \overline{P} \) and those positions stored on the path from root to \( \overline{P} \) with **max-reach-pointer** into the subtree of \( \overline{P} \). All positions can be reported in time \( O(|P|+\text{occ}) \).

2. Otherwise process \( P \) by chunks:
   1. find the longest proper prefix \( P[1..m] \) represented in the heap
   2. build the list of those positions on the path to \( \overline{P[1..m]} \) at which \( P[1..m] \) actually occurs (at most \( m \) such positions)
   3. iteratively proceed to \( P[m+1..] \) to check if it occurs at positions \( (i+m) \) where \( i \) is a position from the current list
String matching on position heaps

T=acatacagatg$

*Example:* $P=acata$

$\{1,5\} \rightarrow \{3,7\} \rightarrow \{5\}$

$\Rightarrow P$ occurs at position

$5+1-|P|=1$

How to search for a pattern $P$. Two cases:

1. If $P$ is represented in the heap, then $P$ occurs at all positions stored in the subtree of $P$ and those positions stored on the path from root to $P$ with max-reach-pointer into the subtree of $P$. All positions can be reported in time $O(|P|+occ)$

2. Otherwise process $P$ by chunks:
   1. find the longest proper prefix $P[1..m]$ represented in the heap
   2. build the list of those positions on the path to $P[1..m]$ at which $P[1..m]$ actually occurs (at most $m$ such positions)
   3. iteratively proceed to $P[m+1..]$ to check if it occurs at positions $(i+m)$ where $i$ is a position from the current list
T=acatacagatg$

Example: $P=acata$

$\{1,5\} \rightarrow \{3,7\} \rightarrow \{5\}$

$\Rightarrow P$ occurs at position $5+1-|P|=1$

Complexity:

**Case 1:** $O(|P|+occ)$

**Case 2:** assume the algorithm partitions $P$ into $v_1v_2\ldots v_k$. Processing each $v_i$ takes time $O(|v_{i-1}|+|v_i|)$. Then the whole algorithm takes time $O(|P|)$

**Conclusion:** all occurrences of $P$ are reported in time $O(|P|+occ)$
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 non exceeding $2^{32} \approx 4 \times 10^9$ chars)

in practice, STs require between 70 and 200 bits/char; this results to 27-80Gb for human genome

note that the sequence itself takes 2bits/char

still, STs are used in practical programs (MUMmer)
Suffix array
### Suffix array: definition

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

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

T=acatacagatg$

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Index</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 (SA) is a permutation of the positions of $T$ taken in the lexicographical order of corresponding suffixes:

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

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

$T = \text{acatacagatg}$

| 1 | $    $ |
| 2 | acagatg$ |
| 3 | acatacagatg$ |
| 4 | agatg$ |
| 5 | atacagatg$ |
| 6 | atg$ |
| 7 | cagatg$ |
| 8 | catacagatg$ |
| 9 | g$ |
| 10 | gatg$ |
| 11 | tacagatg$ |
| 12 | tg$ |

$\text{SA}$

12 5 1 7 3 9 6 2 11 8 4 10

- For any substring, its occurrences form an interval of $\text{SA}$
- Interval $[L_P, R_P]$ for $P$ can be found by two binary searches
- Search for $L_P$:
  
  ```
  while l<r do
    m=(l+r)/2
    if $P <_{\text{lex}} T[\text{SA}[m]..n]$ then
      l=m
    else
      r=m
  
  the whole search for $P$ takes time $O(m \cdot \log(n) + \text{occ})$```
### String matching with suffix arrays

- **T=acatacagatg$**
- **SA**
- **LCP**

<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>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$</td>
<td>12</td>
<td>-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>acagatg$</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>acatacagatg$</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>agatg$</td>
<td>7</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>atacagatg$</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>atg$</td>
<td>9</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>cagatg$</td>
<td>6</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>catacagatg$</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>g$</td>
<td>11</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>gatg$</td>
<td>8</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>tacagatg$</td>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>tg$</td>
<td>10</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **SA**: array of suffix array indices
- **LCP**: array of longest common prefix lengths

- 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 $O(m + \log(n))$
Computing LCP from SA (and T)

- compute $\text{Rank}[i] = \text{SA}^{-1}[i]$
- compute $\text{LCP}[i]$ for $i=\text{Rank}[1], \ldots, \text{Rank}[n]$
- to compute $\text{LCP}[	ext{Rank}[i]]$, look at $\text{LCP}[	ext{Rank}[i-1]]$
- let $q=\text{Rank}[i]$, $p=\text{Rank}[i-1]$
- *key observation*: if $h=\text{LCP}[p]>1$, then $\text{LCP}[q] \geq h-1 \Rightarrow h-1$ comparisons can be saved!

$$j = \text{SA}[p-1] + 1$$
Computing LCP from SA (and T)

- compute \( \text{Rank}[i] = \text{SA}^{-1}[i] \)
- compute \( \text{LCP}[i] \) for \( i=\text{Rank}[1],\ldots,\text{Rank}[n] \)
- to compute \( \text{LCP}[\text{Rank}[i]] \), look at \( \text{LCP}[\text{Rank}[i-1]] \)
- let \( q=\text{Rank}[i], \ p=\text{Rank}[i-1] \)
- *key observation*: if \( h=\text{LCP}[p]>1 \), then \( \text{LCP}[q]\geq h-1 \Rightarrow h-1 \) comparisons can be saved!
- compare successively \( T[i+h], T[i+h+1], \ldots \) with respectively \( T[k+h], T[k+h+1], \ldots \)
Computing LCP from SA (and T)

- compute $\text{Rank}[i] = \text{SA}^{-1}[i]$
- compute $\text{LCP}[i]$ for $i=\text{Rank}[1], \ldots, \text{Rank}[n]$
- to compute $\text{LCP}[	ext{Rank}[i]]$, look at $\text{LCP}[	ext{Rank}[i-1]]$
- let $q=\text{Rank}[i]$, $p=\text{Rank}[i-1]$
- *key observation*: if $h=\text{LCP}[p]>0$, then $\text{LCP}[q] \geq h-1 \Rightarrow h-1$ comparisons can be saved!
- compare successively $T[i+h]$, $T[i+h+1]$, … with respectively $T[k+h]$, $T[k+h+1]$, …
- compute new $h=\text{LCP}[q]$
- iterate to $i+1$
- running time: $O(n)$
String matching with suffix arrays

P = aacccaa

\$ \cdots \text{lcp}_l \cdots \text{suf}_l = \underline{aa}aga\cdots \cdots \cdots \cdots \cdots \cdots \cdots \cdots \cdots \cdots \text{suf}_r = \underline{aaccc}a\cdots \cdots \cdots \cdots \cdots \cdots \cdots \cdots \cdots \cdots \text{lcp}_r \cdots\

\text{notation: suf}_i = T[i..n]

\cdot \text{assume we know lcp}_l = \text{LCP}(\text{suf}_l, P), \text{lcp}_r = \text{LCP}(\text{suf}_r, P) \text{ and assume lcp}_l < \text{lcp}_r
String matching with suffix arrays

$\text{P}=aacc\text{caaa}$

$\text{suf}_l=a\text{aaga...}$

$\text{suf}_m=a\text{accaca...}$

$\text{suf}_r=a\text{accaca...}$

$lcp_i$

$lcp_r$

$\text{SA}$

$\cdot$ assume we know $lcp_i=LCP(suf_l,P)$, $lcp_r=LCP(suf_r,P)$ and assume $lcp_i<lcp_r$

$\cdot$ consider $suf_m$ and consider $LCP(suf_m,suf_r)$
String matching with suffix arrays

- assume we know $lcp_l=LCP(suf_l,P)$, $lcp_r=LCP(suf_r,P)$ and assume $lcp_l<lcp_r$
- consider $suf_m$ and consider $LCP(suf_m,suf_r)$
- **Case 1**: $LCP(suf_m,suf_r)<lcp_r$

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

assume we know lcp_l=LCP(suf_l,P), lcp_r=LCP(suf_r,P) and assume lcp_l<lcp_r

consider suf_m and consider LCP(suf_m,suf_r)

Case 2: LCP(suf_m,suf_r)>lcp_r

Then P <_{lex}suf_m, LCP(suf_m,P)=lcp_r and we set r=m for the next iteration
String matching with suffix arrays

$\text{P} =$ aaccagc

\begin{align*}
\text{suf}_l &= \text{aa} \quad \text{sufl} = \\
\text{suf}_m &= \text{aacc} \quad \text{sufm} = \\
\text{suf}_r &= \text{aacc} \quad \text{sufr} = 
\end{align*}

\begin{align*}
\text{lcp}_l &= \text{lcp}_r = LCP(\text{suf}_l, \text{P}) \\
\text{lcp}_r &= LCP(\text{suf}_r, \text{P}) \quad \text{and assume} \quad \text{lcp}_l < \text{lcp}_r
\end{align*}

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

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

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

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

**Case 3**: $LCP(suf_m,suf_r)=lcp_r$

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

$P = \text{aaccagc}$

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

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

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

$SA$

- 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 3:** $\text{LCP}(suf_m, suf_r) = lcp_r$

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

$P=aaccagc$

$SA$

- assume we know $lcp_{i}=LCP(suf_{i},P)$, $lcp_{r}=LCP(suf_{r},P)$ and assume $lcp_{i}<lcp_{r}$
- consider $suf_{m}$ and consider $LCP(suf_{m},suf_{r})$
  - **Case 3**: $LCP(suf_{m},suf_{r})=lcp_{r}$

Then we keep comparing chars of $P$ with those of $suf_{m}$ until $P[j] \neq suf_{m}[j]$ which determines if $P \leq_{lex} suf_{m}$ or $P >_{lex} suf_{m}$, and also the value $lcp_{m}=LCP(suf_{m},P)$

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

- construction in $O(n \cdot \log(n))$ [Manber, Myers 90]
- 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>Suffix</th>
<th>SA</th>
<th>LCP</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>12</td>
<td>-1</td>
</tr>
<tr>
<td>acagatg$</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>acatacagatg$</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>agatg$</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>atacagatg$</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>atg$</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>cagatg$</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>catacagatg$</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>g$</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>gatg$</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>tacagatg$</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>tg$</td>
<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 ≈ interval \(LCP[i..j]\) s.t. \(\min\{LCP[i+1..j]\}=q\) and \(LCP[i]<q, LCP[j+1]<q\)

- 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 $≈$ interval LCP[i..j] s.t. min{LCP[i+1..j]}=q and LCP[i]<q, 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)
- 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 FM-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

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

$\text{acatacagatg}$
$\text{acagatg}$
$\text{acat}$
$\text{acatacagatg}$
$\text{agatg}$
$\text{acatac}$
$\text{atacagatg}$
$\text{ac}$
$\text{agatg}$
$\text{acatac}$
$\text{cagatg}$
$\text{acata}$
$\text{catacagatg}$
$\text{a}$
$\text{g}$
$\text{acatacagat}$
$\text{gatg}$
$\text{acataca}$
$\text{tacagatg}$
$\text{aca}$
$\text{tg}$
$\text{acatacaga}$
### Burrows-Wheeler Transform

**T** = `acatacagatg$`

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

$T = \text{acatacagatg}$$

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

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

\[ T = \text{acatacagatg} \$

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

F L

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

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

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

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

T=

<table>
<thead>
<tr>
<th>T[SA[i]]</th>
<th>BWT</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>$</td>
</tr>
<tr>
<td>a cagatg$aca</td>
<td>g</td>
</tr>
<tr>
<td>a catacagatg</td>
<td>t</td>
</tr>
<tr>
<td>a gatg$acata</td>
<td>c</td>
</tr>
<tr>
<td>a tacagatg$a</td>
<td>c</td>
</tr>
<tr>
<td>a tg$acataca</td>
<td>g</td>
</tr>
<tr>
<td>c agatg$acat</td>
<td>a</td>
</tr>
<tr>
<td>c atacagatg$</td>
<td>a</td>
</tr>
<tr>
<td>g $acatacaga</td>
<td>t</td>
</tr>
<tr>
<td>g atg$acatac</td>
<td>a</td>
</tr>
<tr>
<td>t acagatg$ac</td>
<td>a</td>
</tr>
<tr>
<td>t g$acatacag</td>
<td>a</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:** $C=[0,1,6,8,10]$

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

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

\[
\begin{array}{c|c|c}
T[SA[i]] & BWT \\
\hline
1 & $ & \text{acatacagat} \\
2 & a & \text{cagatg} \text{$aca} \\
3 & a & \text{catacagatg} \\
4 & a & \text{gatg} \text{$acata} \\
5 & a & \text{tacagatg} \text{$a} \\
6 & a & \text{tg} \text{$acataca} \\
7 & c & \text{agatg} \text{$acat} \\
8 & c & \text{atacagatg} \text{$} \\
9 & g & \text{$acatacag} \\
10 & g & \text{atg} \text{$acatac} \\
11 & t & \text{acagatg} \text{$ac} \\
12 & t & \text{g} \text{$acatacag} \\
\end{array}
\]

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

$T = g$

$T[SA[i]]$  

<table>
<thead>
<tr>
<th>1</th>
<th>$$</th>
<th>acatacagat</th>
<th>$g$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>a</td>
<td>cagatg$aca$</td>
<td>t</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
<td>catacagat$$</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>a</td>
<td>gatg$acata$</td>
<td>c</td>
</tr>
<tr>
<td>5</td>
<td>a</td>
<td>tacagatg$$a$</td>
<td>c</td>
</tr>
<tr>
<td>6</td>
<td>a</td>
<td>tg$acataca$</td>
<td>g</td>
</tr>
<tr>
<td>7</td>
<td>c</td>
<td>agatg$acat$</td>
<td>a</td>
</tr>
<tr>
<td>8</td>
<td>c</td>
<td>atacagatg$$</td>
<td>a</td>
</tr>
<tr>
<td>9</td>
<td>g</td>
<td>$acatacaga$</td>
<td>t</td>
</tr>
<tr>
<td>10</td>
<td>g</td>
<td>atg$acatac$</td>
<td>a</td>
</tr>
<tr>
<td>11</td>
<td>t</td>
<td>acagatg$ac$</td>
<td>a</td>
</tr>
<tr>
<td>12</td>
<td>t</td>
<td>g$acatacag$</td>
<td>a</td>
</tr>
</tbody>
</table>

### BWT

$BWT[i] = T[SA[i]-1]$ if $SA[i] \neq 1$, otherwise $\$$. 

### Observations

1. **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: $C=[0,1,6,8,10]$
2. **Obs 2**: for identical chars, their relative order in F and L is the same.

### Examples

- $LF[i] = C[BWT[i]] + \text{rank}[BWT[i], i]$
  - Ex: $LF[1] = 8 + 1$
Burrows-Wheeler transform

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

\begin{align*}
T[SA[i]] & \\
1 & \$ \text{acatacagat} \quad \text{g} \\
2 & a \text{cagatg$} \text{aca} \quad \text{t} \\
3 & a \text{catacagatg} \quad \$ \\
4 & a \text{gatg$} \text{acata} \quad \text{c} \\
5 & a \text{tacagatg$} \text{a} \quad \text{c} \\
6 & a \text{tg$} \text{acataca} \quad \text{g} \\
7 & c \text{agatg$} \text{acat} \quad \text{a} \\
8 & c \text{atacagatg$} \quad \text{a} \\
9 & \text{g} \$ \text{acatacaga} \quad \text{t} \\
10 & \text{g} \text{atg$} \text{acatac} \quad \text{a} \\
11 & \text{t} \text{acagatg$} \quad \text{a} \\
12 & \text{t} \text{g$} \text{acatacacag} \quad \text{a} \\
\end{align*}

- 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

\[
\begin{array}{cccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
\text{F} & \text{L} \\
\end{array}
\]

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

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

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

\[
\text{BWT}[\text{SA}[i]] = \begin{array}{c}
1 & \text{acatacagat} & g \\
2 & \text{cagatg}\text{aca} & t \\
3 & \text{catacagatg} & $ \\
4 & \text{gatg}\text{acata} & c \\
5 & \text{tacagatg}\text{a} & c \\
6 & \text{tg}\text{acataca} & g \\
7 & \text{agatg}\text{acat} & a \\
8 & \text{atacagatg}\$ & a \\
9 & \text{g}\text{acatacaga} & t \\
10 & \text{atg}\text{acatac} & a \\
11 & \text{tacagatg}\text{ac} & a \\
12 & \text{g}\text{acatacag} & a \\
\end{array}
\]

- 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

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

Ex: LF[1] = 8 + 1
The Burrows-Wheeler Transform (BWT) is a data transformation algorithm that is often used in data compression and bioinformatics. Given a string $T$, the BWT is defined as:

$$T = \text{atg}$\text{g}$$

and

$$T[SA[i]] = \text{acatacagat} \text{g}$$

The BWT is defined as:

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

**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.

For example:

- $\text{LF}[1] = C[\text{BWT}[1]] + \text{rank}[\text{BWT}[1], i]$

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

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

\[
\begin{array}{cccc}
T[SA[i]] & \text{BWT} \\
1 & $ & \text{acatacagat} & g \\
2 & a & cagatg$aca & t \\
3 & a & catacagatg & $ \\
4 & a & gatg$acata & c \\
5 & a & tacagatg$a & c \\
6 & a & tg$acataca & g \\
7 & c & agatg$acat & a \\
8 & c & atacagatg$a & a \\
9 & g & $acatacaga & t \\
10 & g & atg$acatac & a \\
11 & t & acagatg$ac & a \\
12 & t & g$acatacag & a \\
\end{array}
\]

- \( \text{BWT}[i]=T[SA[i]-1] \) 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 \( x \)
  
  \[ \text{Ex: } C=[0,1,6,8,10] \]

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

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

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

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

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

- **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

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

Ex: \( \text{LF}[1]=8+1 \)
LF function

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

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

\[
\begin{array}{cccc}
1 & \$ & \text{acatacagat} & g \\
2 & a & \text{cagatg}\$\text{aca} & t \\
3 & a & \text{catacagatg} & \$ \\
4 & a & \text{gatg}\$\text{acata} & c \\
5 & a & \text{tacagatg}\$\text{a} & c \\
6 & a & \text{tg}\$\text{acataca} & g \\
7 & c & \text{agatg}\$\text{acat} & a \\
8 & c & \text{atacagatg}\$ & a \\
9 & g & \$\text{acatacaga} & t \\
10 & g & \text{atg}\$\text{acatac} & a \\
11 & t & \text{acagatg}\$\text{ac} & a \\
12 & t & g\$\text{acatacag} & a \\
\end{array}
\]

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

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

<table>
<thead>
<tr>
<th>$T[SA[i]]$</th>
<th>BWT</th>
</tr>
</thead>
</table>
| $\$ acatacagat g  
a cagatg$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  
F   | L |

$LF[i]=C[BWT[i]]+rank[BWT[i],i]$
### rank function

<table>
<thead>
<tr>
<th>$T[SA[i]]$</th>
<th>BWT</th>
<th>$\text{rank}[\text{BWT}[i],i]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$$$</td>
<td>acatacagat</td>
<td>g</td>
</tr>
<tr>
<td>a</td>
<td>cagatg$</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>catacagatg</td>
<td>$</td>
</tr>
<tr>
<td>a</td>
<td>gatg$</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>tacagatg$</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>tg$</td>
<td>a</td>
</tr>
<tr>
<td>c</td>
<td>agatg$</td>
<td>a</td>
</tr>
<tr>
<td>c</td>
<td>atacagatg$</td>
<td>a</td>
</tr>
<tr>
<td>g</td>
<td>$</td>
<td>a</td>
</tr>
<tr>
<td>g</td>
<td>atg$</td>
<td>acatacag</td>
</tr>
<tr>
<td>t</td>
<td>acagatg$</td>
<td>ac</td>
</tr>
<tr>
<td>t</td>
<td>g$</td>
<td>acatacag</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>T[SA[i]]</th>
<th>BWT</th>
<th>rank[BWT[i],i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>acatacagat</td>
<td>g 0</td>
</tr>
<tr>
<td>a</td>
<td>cagatg$aca</td>
<td>t 0</td>
</tr>
<tr>
<td>a</td>
<td>catacagatg</td>
<td>$ 0</td>
</tr>
<tr>
<td>a</td>
<td>gatg$acata</td>
<td>c 0</td>
</tr>
<tr>
<td>a</td>
<td>tacagatg$a</td>
<td>c 1</td>
</tr>
<tr>
<td>a</td>
<td>tg$acataca</td>
<td>g 1</td>
</tr>
<tr>
<td>c</td>
<td>agatg$acata</td>
<td>a 0</td>
</tr>
<tr>
<td>c</td>
<td>atacagatg$</td>
<td>a 1</td>
</tr>
<tr>
<td>g</td>
<td>$acatacaga</td>
<td>t 1</td>
</tr>
<tr>
<td>g</td>
<td>atg$acatac</td>
<td>a 2</td>
</tr>
<tr>
<td>t</td>
<td>acagatg$ac</td>
<td>a 3</td>
</tr>
<tr>
<td>t</td>
<td>g$acatacag</td>
<td>a 4</td>
</tr>
</tbody>
</table>

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

How about general queries \( rank[a, i] \) 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]
- $\text{rank}$ 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
Implementing rank on bitmaps

- consider a bitmap B of size n
- tabulate rank within all blocks of size \((\log n)/2\)
  there are \(2^{(\log n)/2} = \sqrt{n}\) different blocks, and \((\log n)/2\) possible queries, with the result taking \((\log \log n)\) bits.
  Overall space: \(O(\sqrt{n} \cdot \log n \cdot \log \log n) = o(n)\)

- *idea*: compute “cumulative rank” for block borders

- takes \(2n/(\log n) \cdot (\log n) = 2n\) bits
  too much! trick: introduce two levels of blocks

\[
\begin{array}{cccccc}
1001 & 1001 & 1101 & 1111 & 0101 & \ldots \\
2 & 4 & 7 & 11 & 13 & \\
\end{array}
\]
Implementing *rank* on bitmaps

- consider a bitmap $B$ of size $n$
- tabulate *rank* within all blocks of size $(\log n)/2$
  there are $2^{(\log n)/2} = \sqrt{n}$ different blocks, and $(\log n)/2$ possible queries, with the result taking $(\log \log n)$ bits.
  Overall space: $O(\sqrt{n} \cdot \log n \cdot \log \log n) = o(n)$
- split $B$ into $n/(\log^2 n)$ superblocks of size $(\log^2 n)$; compute cumulative rank. This takes $n/(\log^2 n) \cdot (\log n) = n/(\log n) = o(n)$ bits
- split each superblock into blocks of size $(\log n)/2$; compute cumulative rank *inside superblock*; the result takes $O(\log \log n)$ bits.
  Therefore we only need
  $2n/(\log n) \cdot (\log \log n) = o(n)$ bits

DONE!
### String matching with BWT

{T=acatacagatg$}

<table>
<thead>
<tr>
<th>T[SA[i]]</th>
<th>BWT</th>
<th>P=taca</th>
</tr>
</thead>
<tbody>
<tr>
<td>$  acatacagat</td>
<td>g</td>
<td></td>
</tr>
<tr>
<td>a  cagatg$aca</td>
<td>t</td>
<td></td>
</tr>
<tr>
<td>a  catacagatg</td>
<td>$</td>
<td></td>
</tr>
<tr>
<td>a  gatg$acata</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>a  tacagatg$a</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>a  tg$acataca</td>
<td>g</td>
<td></td>
</tr>
<tr>
<td>c  agatg$acata</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>c  atacagatg$</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>g  $acatacaga</td>
<td>t</td>
<td></td>
</tr>
<tr>
<td>g  atg$acatac</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>t  acagatg$ac</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>t  g$acatacag</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>L</td>
<td></td>
</tr>
</tbody>
</table>
String matching with BWT

T = acatacagatg$

T[SA[i]]

P = taca

BWT

F

L
String matching with BWT

T = acatacagatg$

T[SA[i]] BWT

$ acatacagat g
a cagatg$aca t
a catacagatg s
a gatg$acata c
a tacagatg$a c
a tg$acataca g
b agatg$acat a
b atacagatg$a a
g $acatacaga t
b atg$acatac a
t acagatg$ac a
t g$acatacag a

F L

P = ta c a
String matching with BWT

$T=acatacagatg$

$T[SA[i]]$  $BWT$  $P=tac\ a$

\$  acatacagat  \ g$
\a  cagatg$aca  \ t  \ e$
\a  catacagatg  \$  \ c  \ e$
\a  gatg$acata  \ c  \ e$
\a  tacagatg$a  \ c  \ f$
\a  tg$acataca  \ g  \ f$
\c  agatg$acat  \ a$
\c  atacagatg$  \ a$
\g  $acatacaga  \ t$
\g  atg$acatac  \ a$
\t  acagatg$ac  \ a$
\t  g$acatacag  \ a$

$F  L$

$e:= C[x]+rank[x,e]+1$
$f:= C[x]+rank[x,f]$
String matching with BWT

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

\[ P = \text{taca} \]

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

T=acatacagatg$

$ acatacagat g
a cagatg$aca t
a catacagatg $

P=taca

[e,f] : current interval
x : letter

\[ e := C[x] + \text{rank}[x,e] + 1 \]
\[ f := C[x] + \text{rank}[x,f] \]
String matching with BWT

$T = \text{acatacagatg}\$

$T[\text{SA}[i]]$  \hspace{1cm} \text{BWT}$

P=taca

$[e,f]$ : current interval

$e := C[x] + \text{rank}[x,e] + 1$

$f := C[x] + \text{rank}[x,f]$
String matching with BWT

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

\[ T[SA[i]] \quad \text{BWT} \quad P = \text{taca} \]

| \$ | $\text{acatacagat} | g | e |
| a | $\text{cagatg}\$\text{aca} | t | f |
| a | $\text{catacagatg} | $ |
| a | $\text{gatg}\$\text{acata} | c |
| a | $\text{tacagatg}\$a | c |
| a | $\text{tg}\$\text{acataca} | g |
| c | $\text{agatg}\$\text{acat} | a |
| c | $\text{atacagatg}\$ | a |
| g | $\text{acatacaga} | t |
| g | $\text{atg}\$\text{acatac} | a |
| t | $\text{acagatg}\$\text{ac} | a |
| t | $\text{g}\$\text{acatacag} | a |
| F | L |

\[ x: \text{letter} \]
\[ e := C[x]+\text{rank}[x,e]+1 \]
\[ f := C[x]+\text{rank}[x,f] \]

[e,f] : current interval
String matching with BWT

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

\[ P = \text{taca} \]

**T[SA[i]]**

- $ \text{acatacagat}$
- a $\text{cagatg}$
- a $\text{catacagat}$
- a $\text{gatg}$$\text{acata}$
- a $\text{tacagatg}$$\text{a}$
- a $\text{tg}$
- c $\text{agatg}$$\text{acata}$
- c $\text{atacagatg}$
- g $\text{acatacaga}$
- g $\text{atg}$
- t $\text{acagatg}$
- t g$\text{acatacag}$

**BWT**

- $\text{g}$
- t
- $\text{e}$
- $\text{f}$

\[ [e,f] : \text{current interval} \]
\[ x : \text{letter} \]
\[ e := C[x] + \text{rank}[x,e] + 1 \]
\[ f := C[x] + \text{rank}[x,f] \]
String matching with BWT

T=acatacagatg$

BWT

\[ \begin{array}{c|c}
\text{SA[i]} & \text{BWT} \\
\hline
$ & \text{acatacagat} \\
a & \text{cagatg}$aca \\
a & \text{catacagatg} \\
a & \text{gatg}$acata \\
a & \text{tacagatg}$a \\
a & \text{tg}$acataca \\
c & \text{agatg}$acata \\
c & \text{acatacagatg}$ \\
g & \text{$acatacaga} \\
g & \text{atg}$acatac \\
t & \text{acagatg}$ac \\
t & \text{g}$acatacag \\
F & \text{L} \\
\end{array} \]

P=taca

\[ e, f : \text{current interval} \]
\[ x : \text{letter} \]
\[ e := C[x] + \text{rank}[x,e]+1 \]
\[ f := C[x] + \text{rank}[x,f] \]
String matching with BWT

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

$\text{P} = \text{taca}$

What position is it??
String matching with BWT

P = taca

T = acatacagatg$

T[SA[i]]

BWT

F L
### FM-index

T = `acatacagatg$

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

**Solution:** store only a fraction of values of SA.

Storing one value over \( \log(n) \) leads to \( O(n \cdot \log(n) / \log(n)) = O(n) \) bits.
FM-index

$T = \text{acatacagatg}$

<table>
<thead>
<tr>
<th>SA</th>
<th>$T[SA[i]]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>$\text{acatacagat}$ $g$</td>
</tr>
<tr>
<td>6</td>
<td>$\text{cagatg$aca}$ $t$</td>
</tr>
<tr>
<td>6</td>
<td>$\text{acatacagatg}$ $s$</td>
</tr>
<tr>
<td>8</td>
<td>$\text{gatg$acata}$ $c$</td>
</tr>
<tr>
<td>2</td>
<td>$\text{atcacagatg}$ $a$</td>
</tr>
<tr>
<td>4</td>
<td>$\text{acagatg$ac}$ $c$</td>
</tr>
<tr>
<td>10</td>
<td>$\text{g$acatacag}$ $a$</td>
</tr>
</tbody>
</table>

Solution: store only a fraction of values of SA.

Storing one value over $\log(n)$ leads to $O(n \cdot \log(n)/\log(n)) = O(n)$ bits

Search time becomes $O(|P| \cdot \log(n))$

[Ferragina, Manzini 00]

FM-index includes:
- BWT
- selection of SA values
- auxiliary structures: array C, rank, position marking …
FM-index: practical issues

- FM-index can be implemented using \(~3\) bits/char (!!)
- FM-index is now \textit{widely} used in practical bioinformatics software: BWA, bowtie, SOAP2 (mapping), CGA (assembly)
- other succinct data structure exist (including \textit{compact suffix array}) and continue to appear
- construction may require much more space than the resulting structure
- external memory algorithms are important
- algorithms specialized to multi-core or GPU processor architectures
- dynamic indexes