Space-Efficient Data Structures for Strings and Sequences

Roberto Grossi
Università di Pisa, Italy

Joint works with
P. Ferragina, A. Gupta, G. Ottaviano,
K. Sadakane, R. Shah, J.S. Vitter, B. Xu
Why string data?

Ubiquitous and massive data:
- Digital libraries and product catalogues
- Electronic white and yellow pages
- Specialized information sources (e.g., genome or patents)
- Web pages repositories
- XML format for structured data
- Newsgroups
- ...
What is a String Dictionary?

- Persistent and **space-efficient** data structure.
- **Quickly** answer to lots of search queries.
- Accessing a **small** portion of the string collection.

- Basic building block of any IR system (along with modeling, ranking, query languages, security and access control)

For example...
Text indexing (TI): world-level vs full-text

“This rare goods [the information] will be prepared under malleable or eatable form, will be delivered to more and more customers; it will be sold, exported, duplicated and reproduced everywhere…”

Paul Valéry, Poet, 1871-1945
Text indexing (TI): world-level vs full-text

“This rare goods [the information]
will be prepared under malleable or eatable form,
will be delivered to more and more customers;
it will be sold, exported, duplicated and reproduced
everywhere…”

Paul Valéry, Poet, 1871-1945
Word-level TI: Inverted lists/files

- Split the text into words
- Collect all distinct words in a string dictionary: for each word \( w \), store the (inverted or position) list of its locations in the text
- Support Boolean/ranked queries
“This rare goods [the information]

will be prepared under malleable or eatable form,

will be delivered to more and more customers;

it will be sold, exported, duplicated and reproduced

everywhere, and …”

Paul Valéry, Poet, 1871-1945
Basic property of full-text indexing: **Prefix searching** in a string dictionary

Pattern $P$ occurs at position $i$ of text $T$ iff

$P$ is prefix of a suffix of $T$

**mississippi**
Basic property of full-text indexing: **Prefix searching** in a string dictionary

Pattern $P$ occurs at position $i$ of text $T$ iff $P$ is prefix of a suffix of $T$
Example of **string dictionaries** supporting **prefix search**, you can make **succinct** using the techniques seen in previous talks...
Trie T for a set S of strings

1. S empty $\rightarrow$ T empty
2. $S = S_A \cup S_B \ldots \cup S_Z$ (partition):

   T is the root with children
   $T_A, T_B, \ldots, T_Z$

   with $T_x = \text{trie}(S_x)$ with initial x removed
Compacted tries

Branching character

Node label

three
trial
triangle
trie
triple
triply
Compact trie / Patricia storing the suffixes of $bababa#$

Space is roughly $16n$ bytes [Manber, Myers]

Online construction in linear time [Ukkonen]
Full text index: Automaton and DAWG

- Collapse isomorphic nodes to reduce their number
- Average size is about 1.3 nodes per character
- Nodes are of larger size
- Number of bytes does not reduce much
String B-trees

P = AT
**SA suffix array:** *mississippi*

<table>
<thead>
<tr>
<th></th>
<th>mississippi</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>mississippi</td>
</tr>
<tr>
<td>2</td>
<td>ississippi</td>
</tr>
<tr>
<td>3</td>
<td>ssissippi</td>
</tr>
<tr>
<td>4</td>
<td>sississippi</td>
</tr>
<tr>
<td>5</td>
<td>ississippi</td>
</tr>
<tr>
<td>6</td>
<td>ssissippi</td>
</tr>
<tr>
<td>7</td>
<td>sippi</td>
</tr>
<tr>
<td>8</td>
<td>ippi</td>
</tr>
<tr>
<td>9</td>
<td>ppi</td>
</tr>
<tr>
<td>10</td>
<td>pi</td>
</tr>
<tr>
<td>11</td>
<td>i</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>mississippi</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>mississippi</td>
</tr>
<tr>
<td>2</td>
<td>ississippi</td>
</tr>
<tr>
<td>3</td>
<td>ssissippi</td>
</tr>
<tr>
<td>4</td>
<td>sississippi</td>
</tr>
<tr>
<td>5</td>
<td>ississippi</td>
</tr>
<tr>
<td>6</td>
<td>ssissippi</td>
</tr>
<tr>
<td>7</td>
<td>sippi</td>
</tr>
<tr>
<td>8</td>
<td>ippi</td>
</tr>
<tr>
<td>9</td>
<td>ppi</td>
</tr>
<tr>
<td>10</td>
<td>pi</td>
</tr>
<tr>
<td>11</td>
<td>i</td>
</tr>
</tbody>
</table>
Let us focus on suffix arrays...
text $T = \text{SENSELESSNESS}$

$\text{i} \quad \text{SA}[i] \quad T[\text{SA}[i] \ldots n]$
Storing the SA permutation...

“suffix link” for SA: 
\[ \Phi \text{ function } [G.\& Vitter '00] \]

\[ \Phi(i) = k \text{ iff } SA[k] = SA[i] + 1 \]

\[ \Phi(i) = 0 \text{ when } SA[i] = n \text{ or } \]
\[ \Phi(i) = z \text{ where } SA[z] = 1 \]
Using $\Phi$ and a bitvector $BV$ we can avoid storing $SA$.
Using $\Phi$ and a bitvector $BV$ we can avoid storing $SA$.
Using $\Phi$ and a bitvector $BV$ we can avoid storing $SA$.
Using $\Phi$ and a bitvector $BV$ we can avoid storing $SA$.
Using $\Phi$ and a bitvector $BV$ we can avoid storing $SA$. 

$$
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c|c|}
\text{BV}[i] & \text{SA}[i] & \Phi(i) \\
\hline
1 & 14 & 0/11 \\
2 & 5 & 6 \\
3 & 2 & 8 \\
4 & 11 & 13 \\
5 & 7 & 14 \\
6 & 6 & 5 \\
7 & 10 & 4 \\
8 & 3 & 10 \\
9 & 13 & 1 \\
10 & 4 & 2 \\
11 & 1 & 3 \\
12 & 9 & 7 \\
13 & 12 & 9 \\
14 & 8 & 12 \\
\end{array}
$$
Using $\Phi$ and a bitvector $BV$ we can avoid storing $SA$.
Using $\Phi$ and a bitvector $BV$ we can avoid storing $SA$.
Replacing SA by $\Phi$ and BV

- Record which $SA[j] = 1$
- $T[i] = c$ iff $\Phi^{(i)}(j) \in \text{Interval}_c$
This is what we need to store:
- bitvector BV
- $\Phi$ function

Also $\Phi$ is represented by bitvectors!
Exploit $\Phi$’s properties

- Observation made on SA perms:
  \[ T[SA[i]] = T[SA[i+1]] \Rightarrow \Phi(i) < \Phi(i+1) \]

- Monotonicity: a maximal run of equal symbols
  \[ T[SA[i]] \mathbin{=} \cdots \mathbin{=} T[SA[j]] \Rightarrow 1 \leq \Phi(i) < \cdots < \Phi(j) \leq n \]

- At most $\sigma$ runs, called $\Sigma$ lists
Wavelet tree on BWT
Entering compression stuff...

A first simple approach
• Encode bitvector BV in $B(\sigma, n) + o(n)$ bits
• Encode $\Phi$ function in $B(n, n\sigma) + o(n \log \sigma)$ bits

Using Wavelet Trees, compressed representation in $n \log \sigma + \text{lower order bits}$ becomes high-order entropy $n \, H_k$ by a refined analysis
Let us focus on wavelet trees...
membership: $k$-th symbol
rank(c,k): # c's in the first k symbols
select$(c,k)$: position of the $k$-th symbol $c$
2D-range(c..d, j..k): like in Comp.Geom.
Wavelet tree represents…

- a sequence of symbols as we saw
- a reordering stored in the root of the multiset given by the elements in the leaves
- a set of grid points, where the x-coordinates are the positions and and the y-coordinates are the values
A list of applications [Navarro ’12]

- entropy encoding with direct access
- range queries
- positional inverted indexes
- graph representation
- permutations
- numeric sequences
- document retrieval
- binary relations
- data mining

... not always the best bounds, but easy to get them
Some theory
Quick Analysis of Space Occupancy

• $B^j = j$-th raw bit string in preorder traversal of the wavelet tree

• Fact 1. $\sum_j |B^j| H_0(B^j) = |\text{string}| \times H_0(\text{string})$

• Fact 2 [Makinen-Navarro]. Wavelet on $\text{BWT(string)}$

$$\sum_j |B^j| H_0(B^j) = |\text{string}| \times H_k(\text{string})$$
(order-$k$ empirical entropy)
The rationale behind wavelet trees...

FOCUS on bit string $B \equiv B^j$
Fully indexable dictionaries (FID)  
[Brodnik, Munro ‘99, Raman et al ’02]

- Bitvector $V$ with $n$ 1s and $m-n$ 0s:
  - $\text{rank}_1(i) = \#1s$ in $V[1...i]$  
  - $\text{select}_1(j) =$ position in $V$ of the $j$th 1  
  - same operations for 0s

- Can also solve the predecessor problem

- Space: $B(n,m) + o(m) \sim n H_0 + o(m)$ bits
- Time: $O(1)$
Run Length Encoding (RLE) of a Bit String

- Bit string \( B = B[1 \ldots n] = b_1^{l_1} b_2^{l_2} \ldots b_m^{l_m} \)
  - \( b_i \neq b_j \), if \( i \neq j \)
  - \( l_i > 0 \)

- Prefix-free positive integer coding [Elias’ 75]
  - \( |\gamma(x)| = 2\lfloor \log x \rfloor + 1 \) bits
  - \( |\delta(x)| = \lceil \log x \rceil + 2\lfloor \log (\lceil \log x \rceil + 1) \rfloor + 1 \) bits

- RLE-\( \gamma \) of \( B \):
  \( b_1 \gamma(l_1) \gamma(l_2) \ldots \gamma(l_m) \)
  - Uniquely decodable

- RLE-\( \delta \) of \( B \):
  \( b_1 \delta(l_1) \delta(l_2) \ldots \delta(l_m) \)
RLE of a Wavelet Tree

- $T$: an arbitrary text
  - Drawn from an alphabet of size $\sigma$

- $T_{rle,\gamma}$ = wavelet tree of $T$ with each bit array RLE-$\gamma$ encoded

- $T_{rle,\delta}$ = wavelet tree of $T$ with each bit array RLE-$\delta$ encoded
Size of the RLE of a Wavelet Tree

- Total #bits in all the RLE encoded bit arrays

- $|B_{\text{rle},\gamma}| \leq 2 \ nH_0(B) + 2 \log n + 2$
  
- $|T_{\text{rle},\gamma}| \leq 2 \ nH_0(T) + (2\log n + 2)(\sigma - 1)$

- $|B_{\text{rle},\delta}| \leq 3 \ nH_0(B) + 2\log(\log n + 1) + \log n + 2$

- $|T_{\text{rle},\delta}| \leq 3 \ nH_0(T) + (2\log(\log n + 1) + \log n + 2)(\sigma - 1)$

RLE-$\delta$ is less space efficient for an arbitrary text!!
Why is RLE-$\delta$ Less Space Efficient?

- Although:
  - $\gamma(x)$ uses $2\lfloor \log x \rfloor + 1$ bits
  - $\delta(x)$ uses $\lfloor \log x \rfloor + 2\lfloor \log (\lfloor \log x \rfloor + 1) \rfloor + 1$ bits

- Actually:
  $\delta(x)$ uses more bits if $x$ is small, which is often the case in the wavelet tree bit arrays.
  [well-known fact]
Why is RLE-$\delta$ Less Space Efficient?

(Cont’d)

A comparison of $|\gamma(x)|$ and $|\delta(x)|$

(a) Over a narrow range of $x$

$|\delta(x)| < |\gamma(x)|$ for $x > 31$; $|\delta(x)| = |\gamma(x)|$ for $x \in \{1, 4-7, 16-31\}$; $|\delta(x)| > |\gamma(x)|$ for $x \in \{2-3, 8-15\}$. 

(b) Over a wide range of $x$
Can RLE-\( \delta \) become provably better?

- **Yes, when the data is highly skewed.**

- **For a bit string** \( B \)
  - If \( H_0(B) = o(1) \):
    \[
    |B_{rle,\delta}| \leq nH_0(B) + o(nH_0(B)) + 2\log(\log n + 1) + \log n + 2 \quad \text{(optimal)}
    \]
    (while \( |B_{rle,\gamma}| \leq 2nH_0(B) + 2\log n + 2 \))

- **For an arbitrary text** \( T \) from alphabet of size \( \sigma \)
  - If \( \sigma = O(1) \) and \( H_0(T) = o(\log \sigma) \):
    \[
    |T_{rle,\delta}| \leq nH_0(T) + o(nH_0(T)) + (2\log(\log n + 1) + \log n + 2)(\sigma - 1) \quad \text{(optimal)}
    \]
    (while \( |T_{rle,\gamma}| \leq 2nH_0(T) + (2\log n + 2)(\sigma - 1) \))
Experiments
Software tool for wavelet trees

3 tree shapes (balanced, Huffman, Hu-Tucker)

11 coding schemes for bit strings $B^i$

• Can experiment 33 possible incarnations...

• URL:
  http://penguin.ewu.edu/~bojianxu/publications
Experimental results

1. RLE+$\gamma$ is good for low-entropy text
2. Hu-Tucker shape as good as Huffman shape, except for low-entropy data
3. Queries are faster in Hu-Tucker shape
Part II

Fast Compressed Tries through Path Decompositions
Compacted tries

Branching character

Node label

three
trial
triangle
trie
triple
triply
Applications

• String dictionaries
  – With prefix lookup, predecessor, ...
  – Exploit prefix compression

• Monotone perfect hash functions
  – “Hollow” or “Blind” tries [ALENEX 09]
  – Binary tree (no need store branching chars)
  – No need to store node labels, just lengths (skips)
Height vs. performance

• Tries can be deep – no guarantee on height
• Bad with pointer-based trees
  – ~1 cache miss per *child* operation
• Worse with succinct tree encodings
  – Need to access several directories
  – *Many* cache misses per *child* operation
  – Large constants hidden in the O(1)
Problem

- Given a sorted set $S$ of $K$ prefix-free strings
  - $\{s_1, s_2, s_3, \ldots, s_K\}$
  - With total length $N = \sum |s_i|$
  - With each character drawn from $\Sigma = \{1, 2, \ldots, \sigma\}$

- Create a data structure that answers the following queries for a pattern $p$
  - $\text{Lookup}(p)$ returns -1 if $p$ is not in $S$ or a unique ID in the interval $[K]$
  - $\text{Access}(i)$ retrieves the string with ID = $i$
Linearized Trie Lower Bound: $LT(S)$

- Let $T$ be the compacted trie of $S$
- Let $E$ be the number of characters on the edges of $T$
- Let $t$ be the total number of nodes of $T$

Theorem. Any succinct encoding of $S$ requires at least

$$LT(S) = E \log \sigma + B(t-1,E)$$

bits in the worst case.

$$B(n,m) = \lceil \log \binom{m}{n} \rceil$$
Upper Bounds

Encoding for $S$ that

- supports $\text{Lookup}(p)$ and $\text{Access}(i) = p$ in $O(p)$ time
- uses $(1 + o(1)) \cdot \text{LT}(S) + O(K)$ bits
- easy to implement and fast
Path decomposition

Recorse here with suffix le

Query: triple
Centroid path decomposition

• Decompose along the **heavy paths**
  – choose the edge that has most descendants
• Height of the decomposed tree: $O(\log n)$
  – Usually lower
• Average height

<table>
<thead>
<tr>
<th></th>
<th>Web Queries</th>
<th>URLs</th>
<th>Synthetic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compacted trie</td>
<td>11.0</td>
<td>18.1</td>
<td>504.4</td>
</tr>
<tr>
<td>Centroid trie</td>
<td>5.2</td>
<td>6.2</td>
<td>2.8</td>
</tr>
<tr>
<td>Hollow trie</td>
<td>50.8</td>
<td>67.3</td>
<td>1005.3</td>
</tr>
<tr>
<td>Centroid hollow trie</td>
<td>8.0</td>
<td>9.2</td>
<td>2.8</td>
</tr>
</tbody>
</table>
Succinct encoding

- [PODS 08] presents a succinct data structure for centroid path-decomposed tries
- Not practical: need complex operations on succinct trees
- Simpler and practical encoding
- This encoding enables also simple compression of the labels
Succinct encoding

\[ \text{triangle} \]

- Node label written literally, interleaved with number of other branching characters at that point in array \( L \)
- Corresponding branching characters in array \( B \)
- Tree encoded with DFUDS in bitvector \( BP \)
  - Variant of Range Min-Max tree [ALENEX 10] to support Find\{Close,Open\}, more space-efficient (Range Min tree)
Compression of \textbf{L}

\textcolor{red}{\texttt{index.html}}\$...\textcolor{blue}{\texttt{.html}}\$...\textcolor{red}{\texttt{.html}}\$...
\textcolor{blue}{\texttt{index.html}}$

\begin{itemize}
  \item Dictionary codewords shared among labels
  \item Codewords do not cross label boundaries ($\$$)
  \item Use vbyte to compress the codeword ids
\end{itemize}
Compression of L

• Node labels (triangle, lie, ...):
  – each label is suffix of a string in the set
  – interleaved with few “special characters” 1, 2, 3,...

• Compressible if strings are compressible

• Dictionary and parsing computed with modified Re-Pair
  – Domain-specific compression can be used instead

• Decompression overhead negligible
Experimental results (time)

• Experiments show gains in time comparable to the gains in height
• Confirm that bottleneck is traversal operations

<table>
<thead>
<tr>
<th></th>
<th>Web Queries</th>
<th>URLs</th>
<th>Synthetic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trie</td>
<td>3.5</td>
<td>7.0</td>
<td>119.8</td>
</tr>
<tr>
<td>Centroid trie</td>
<td>2.4</td>
<td>4.3</td>
<td>5.1</td>
</tr>
<tr>
<td>Hollow trie [ALENEX 09]</td>
<td>16.6</td>
<td>22.4</td>
<td>462.7</td>
</tr>
<tr>
<td>Hollow trie</td>
<td>7.2</td>
<td>13.9</td>
<td>137.1</td>
</tr>
<tr>
<td>Centroid hollow trie</td>
<td>2.8</td>
<td>4.4</td>
<td>11.1</td>
</tr>
</tbody>
</table>

(microseconds, lower is better)

Code available at https://github.com/ot/path_decomposed_tries
Experimental results (space)

• For strings with many common prefixes, even non-compressed trie is space-efficient
• Labels compression considerably increases space-efficiency
• Decompression time overhead: ~10%

<table>
<thead>
<tr>
<th></th>
<th>Web Queries</th>
<th>URLs</th>
<th>Synthetic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hu-Tucker Front Coding</td>
<td>40.9%</td>
<td>24.4%</td>
<td>19.1%</td>
</tr>
<tr>
<td>Centroid trie</td>
<td>55.6%</td>
<td>22.4%</td>
<td>17.9%</td>
</tr>
<tr>
<td>Centroid trie + compression</td>
<td>31.5%</td>
<td>13.6%</td>
<td>0.4%</td>
</tr>
</tbody>
</table>

(compression ratio, lower is better)

Code available at [https://github.com/ot/path_decomposed_tries](https://github.com/ot/path_decomposed_tries)
Part III:

Wavelet Trie for a Sequence of Strings
Sequence of strings ≠ Set of strings

- **Time order** e.g. in data analytics:

  *What has been the most accessed domain during winter vacation?*

- **Binary relation** e.g. in web graphs and social networks:

  *How did friendship links change during winter vacation?*
Indexed String Sequences

- Queries
  - **Access(i):** access the $i$-th element
    - Access(2) = foobar
  - **Rank(s, pos):** count occurrences of $s$ before $pos$
    - Rank(bar, 5) = 2
  - **Select(s, i):** find the $i$-th occurrence of $s$
    - Select(foo, 2) = 6
Prefix operations

(\texttt{foo, bar, foobar, foo, bar, bar, foo})

- **Queries**
  - **RankPrefix**($p$, $pos$): count strings prefixed by $p$ before $pos$
    - RankPrefix(\texttt{foo}, 5) = 3
  - **SelectPrefix**($p$, $i$): find the $i$-th string prefixed by $p$
    - SelectPrefix(\texttt{foo}, 2) = 3
Example: storing relations

- Write the **columns** as string sequences
  - Store them separately
  - Reduce relational operations to sequence queries

<table>
<thead>
<tr>
<th>User</th>
<th>Likes URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Leonard</td>
</tr>
<tr>
<td>1</td>
<td>Penny</td>
</tr>
<tr>
<td>2</td>
<td>Sheldon</td>
</tr>
<tr>
<td>3</td>
<td>Penny</td>
</tr>
<tr>
<td>4</td>
<td>Leonard</td>
</tr>
<tr>
<td>5</td>
<td>Sheldon</td>
</tr>
<tr>
<td>6</td>
<td>Sheldon</td>
</tr>
</tbody>
</table>

What does **Sheldon** like?

Who likes pages from domain **wikipedia.org**?

Other operations: range counting, ...
Some notation

- Sequence $S$, $|S| = n$
  - In the example $n = 7$
- String set $S_{set}$ is *unordered set of distinct* strings appearing in $S$
  - In the example, $\{\text{foo, bar, foobar}\}$, $|S_{set}| = 3$
  - Also called *alphabet* but is LARGE and VARIABLE
- Sequence symbols can also be integers, characters, ...
  - As long as they are binarized to strings
Dynamic sequences

We want to support the following operations:

- **Insert(s, pos)**: insert the string \( s \) immediately before position \( pos \)
- **Append(s)**: append the string \( s \) at end of the sequence (special case of **Insert**)
- **Delete(pos)**: delete the string at position \( pos \)

If data structure only supports **Append**, we call it **append-only**, otherwise **dynamic** (or **fully dynamic**)

All the operations can change \( S_{set} \)
The Wavelet Trie

- The Wavelet Trie is a Wavelet Tree on sequences of binary strings ($S_{set} \subset \{0, 1\}^*$)
- Supports Access/Rank(Prefix)/Select(Prefix)
- Fully dynamic...
- ... or append only (with better bounds)
- The string set $S_{set}$ need not be known in advance
Wavelet Trie: Construction

Sequence of binary strings

Branching bit: $\beta$

Common prefix: $\alpha$

$\alpha$: 010
$\beta$: 1001011
Wavelet Trie: Construction

α: 010
β: 1001011

α: ε
β: 101

α: 01

α: 10

α: 10

α: ε
β: 1011

α: ε
β: 110

α: ε

α: ε

010111
0100110
0100001
0101010
0100110
010111
010110
Space analysis

• Information-theoretic lower bound

\[ \text{LB}(S) = \text{LT}(S_{\text{set}}) + nH_0(S) \]

  \( \text{LT} \) is the information-theoretic lower bound for storing a set of strings (see previous slides)

  \( nH_0(S) \) is the 0-th order entropy of a sequence \( S \) of \( n \) atomic items

• \( \hat{h} = \text{avg height of the Wavele Trie (WT)} \)
Space analysis

• Static WT:
  \[ \text{LB}(S) + o(\hat{hn}) \]

• Append-only WT:
  \[ \text{LB}(S) + \text{PT}(S_{\text{set}}) + o(\hat{hn}) \]
  \[ \text{PT}(S_{\text{set}}): \text{space taken by the Patricia Trie} \]

• Fully dynamic WT:
  \[ \text{LB}(S) + \text{PT}(S_{\text{set}}) + O(nH_0(S)) \]
Operations time complexity

• Need new dynamic bitvectors to support \textit{initialization} (create a bitvector $0^n$ or $1^n$)

• Static and Append-only Wavelet Trie
  – All supported operations on $s$ in $O(|s| + h_s)$
  – $h_s$ is number of nodes traversed by string $s$

• Fully dynamic Wavelet Trie
  – All supported operations in $O(|s| + h_s \log n)$
  – Deletion may take $O(|\hat{s}| + h_s \log n)$ where $\hat{s}$ is longest string in the trie
Wavelet Trie: Access

Access(5) = 010 1 1 1

Rank is similar
Wavelet Trie: Select

Select(0100110, 2) = 4
Wavelet Trie: Append

Insert/Delete are similar
Summary

• Compressed suffix arrays for a \texttt{string} = sequence of symbols
• Wavelet trees for a string
• Succinct tries for a \texttt{set of strings}
• Wavelet tries for a sequence of strings

• Industrial applications: json processing, query log analysis, auto-completion
• Code being developed is available at https://github.com/ot/succinct
Thanks for your attention!

Questions?