Indexing Compressed Text: a Tale of Time and Space

Nicola Prezza, LUISS Guido Carli, Rome
18th Symposium on Experimental Algorithms, Catania, Italy, June 16-18, 2020
Introduction
In this talk I will present a brief history and state-of-the-art of the problem of computing over compressed data.
In this talk I will present a brief history and state-of-the-art of the problem of computing over compressed data.

We will look at solutions for a specific problem (text indexing). In general, the question of the field is:
In this talk I will present a brief history and state-of-the-art of the problem of computing over compressed data.

We will look at solutions for a specific problem (text indexing). In general, the question of the field is:

"I have a really good compressor that compresses my data $X$ into an archive $C$, with $\text{size}(C) \ll \text{size}(X)$.

Can I perform computation directly over $C$, without decompressing it?"
In general, the solution depends on the compressor $C$ and on the problem (i.e. input and queries).
Compressed text indexing

In general, the solution depends on the compressor $C$ and on the problem (i.e. input and queries).

In this talk, we will see solutions for different $Cs$ and one particular problem:

Definition (text indexing)

Given a string $S \in \Sigma^n$, build a data structure $D(S)$ that answers the following queries:

- Count the number of occurrences of a string $P \in \Sigma^m$, $m \leq n$ in $S$
- Locate the occurrences of $P$ in $S$
- Extract a text substring $S[i, \ldots, i + \ell - 1]$

Additional constraint: $D(S)$ should take space proportional to $C$ (compressed).
In general, the solution depends on the compressor $C$ and on the problem (i.e. input and queries).

In this talk, we will see solutions for different $C$s and one particular problem:

**Definition (text indexing)** Given a string $S \in \Sigma^n$, build a data structure $D(S)$ that answers the following queries:
Compressed text indexing

In general, the solution depends on the compressor $C$ and on the problem (i.e. input and queries).

In this talk, we will see solutions for different $Cs$ and one particular problem:

**Definition (text indexing)** Given a string $S \in \Sigma^n$, build a data structure $D(S)$ that answers the following queries:

- **Count** the number $occ$ of occurrences of a string $P \in \Sigma^m$, $m \leq n$ in $S$
Compressed text indexing

In general, the solution depends on the compressor $C$ and on the problem (i.e. input and queries).

In this talk, we will see solutions for different $C$s and one particular problem:

**Definition (text indexing)** Given a string $S \in \Sigma^n$, build a data structure $D(S)$ that answers the following queries:

- **Count** the number $occ$ of occurrences of a string $P \in \Sigma^m$, $m \leq n$ in $S$
- **Locate** the $occ$ of occurrences of $P$ in $S$
In general, the solution depends on the compressor $C$ and on the problem (i.e. input and queries).

In this talk, we will see solutions for different $C$s and one particular problem:

**Definition (text indexing)** Given a string $S \in \Sigma^n$, build a data structure $D(S)$ that answers the following queries:

- **Count** the number $occ$ of occurrences of a string $P \in \Sigma^m$, $m \leq n$ in $S$
- **Locate** the $occ$ of occurrences of $P$ in $S$
- **Extract** a text substring $S[i, \ldots, i + \ell - 1]$
Compressed text indexing

In general, the solution depends on the compressor $C$ and on the problem (i.e. input and queries).

In this talk, we will see solutions for different $Cs$ and one particular problem:

**Definition (text indexing)** Given a string $S \in \Sigma^n$, build a data structure $D(S)$ that answers the following queries:

- **Count** the number $occ$ of occurrences of a string $P \in \Sigma^m$, $m \leq n$ in $S$
- **Locate** the $occ$ of occurrences of $P$ in $S$
- **Extract** a text substring $S[i, \ldots, i + \ell - 1]$

Additional constraint: $D(S)$ should take space proportional to $C$ (compressed).
Example

\[ S = \text{ATAATAAGA} \]

\begin{align*}
1 & \quad 2 & \quad 3 & \quad 4 & \quad 5 & \quad 6 & \quad 7 & \quad 8 & \quad 9 \\
\end{align*}

- Count(ATA) = 3
- Locate(ATA) = \{1, 3, 7\}
- Extract(4,7) = "TAGA"

Note: because of the extract query, \( D(S) \) replaces \( S \) (we call it a self-index).
Entropy Compression
At first, research focused on Shannon’s measure of text entropy.

In order to do so, we first need to adapt the definition to the empirical character frequencies (we work on texts, not on character sources):
At first, research focused on Shannon’s measure of text entropy.

In order to do so, we first need to adapt the definition to the empirical character frequencies (we work on texts, not on character sources):

**Definition (Zero-Order Empirical Entropy)**

\[
H_0(S) = \sum_{c \in \Sigma} \frac{\text{occ}_c}{n} \log_2 \frac{n}{\text{occ}_c}
\]

where \(\text{occ}_c\) = number of occurrences of character \(c\) in \(S\).
At first, research focused on Shannon’s measure of text entropy. In order to do so, we first need to adapt the definition to the empirical character frequencies (we work on texts, not on character sources):

Definition (Zero-Order Empirical Entropy)

\[
H_0(S) = \sum_{c \in \Sigma} \frac{\text{occ}_c}{n} \log_2 \frac{n}{\text{occ}_c}
\]

where \( \text{occ}_c \) = number of occurrences of character \( c \) in \( S \).

Thm. \( nH_0(S) \) bits are needed to represent a text using any encoding of the alphabet’s characters into binary codes that only depend on the character’s frequency.
A more powerful notion clusters symbols by context.
A more powerful notion clusters symbols by context.

Let $S_C$ = sting obtained by concatenating all characters that follow substring $C$ in $S$.

Example: in $S = AAAT AAG CT$, $S_{AA} = "ATG"$
High-Order Empirical Entropy

A more powerful notion clusters symbols by context.

Let $S_C$ = sting obtained by concatenating all characters that follow substring $C$ in $S$.

Example: in $S = AAAT AAG CT$, $S_{AA} = "ATG"$

Definition (High-Order Empirical Entropy*)

$$H_k = \sum_{C \in \Sigma^k} \frac{|S_C|}{n} \cdot H_0(S_C)$$

Intuition: weighted average of the contexts’ zero-order entropies.

*From now on we will simply write $H_k$ instead of $H_k(S)$
Entropy compressors (e.g. Huffman, arithmetic) compress $S$ into $nH_k + o(n \log \sigma)$ bits, for some $k \leq \log \sigma n$ * ($\sigma = |\Sigma| = \text{alphabet size}$)

On typical context-predictable texts, e.g. XML:

- $nH_0$ is about 65% of $n \log \sigma$.
- $nH_5$ is about 10% of $n \log \sigma$.

* We cannot do much better than that: Gagie [Inf. Proc. Letters, 2016] showed that for $k \geq \log \sigma n$, no compressed representation can achieve a worst-case space bound of $\Theta(nH_k) + o(n \log \sigma)$
Goal: build a text index taking $O(nH_k) + o(n \log \sigma)$ bits of space and supporting fast queries.
Goal: build a text index taking $O(nH_k) + o(n \log \sigma)$ bits of space and supporting fast queries.

Classic solutions: **suffix trees, suffix arrays**. Fast, but use $O(n \log n)$ bits of space, which could be two orders of magnitude larger than $nH_k$. 
Goal: build a text index taking $O(nH_k) + o(n \log \sigma)$ bits of space and supporting fast queries.

Classic solutions: suffix trees, suffix arrays. Fast, but use $O(n \log n)$ bits of space, which could be two orders of magnitude larger than $nH_k$.

Let’s see (in 1 slide!) what is and how to compress a suffix array
Input $\$\text{-terminated text}$ ($\preceq_{\text{lex}} \ c$ for all $c \in \Sigma$)

$$S = \text{A T A T A G A T } \$ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9$$

Note: $\psi$ is increasing by letter (color).
Why? applying $\psi$ = removing the first char from a suffix. Preserves relative ordering of suffixes starting with same letter.

Store $\Delta[i] = \psi[i] - \psi[i - 1]$ (delta-encoding):

$nH_0 + O(n)$ bits, $O(1)$ random access.
Input $\$\text{-terminated text}$ ($\preceq_{\text{lex}} c$ for all $c \in \Sigma$)

$$S = \begin{array}{cccccccccc}
  & A & T & A & T & A & G & A & T & \$ \\
 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & \\
\end{array}$$

**Suffix Array**: sort positions by lexicographic order of suffixes:

$$SA = \begin{array}{cccccccccc}
  & 9 & 5 & 7 & 3 & 1 & 6 & 8 & 4 & 2 \\
\$ & A & A & A & A & G & T & T & T & \\
G & T & T & T & A & $ & A & A & \\
A & $ & A & A & T & G & T & \\
T & G & T & $ & A & A & \\
$ & A & A & T & G & \\
T & G & $ & A & \\
$ & A & T & \\
T & $ & \\
$ & \\
\end{array}$$

Note: occurrences of a pattern form a range: count/locate = binary search.
Input $\$\$-terminated text ($\preceq_{\text{lex}} c$ for all $c \in \Sigma$)

\[
S = \begin{array}{cccccc}
A & T & A & T & A & G & A & T & \$ \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\end{array}
\]

\[\psi \text{ Array: } \psi[i] = SA^{-1}[SA[i] + 1] \]

\[
\begin{array}{cccccc}
SA & = & 9 & 5 & 7 & 3 & 1 & 6 & 8 & 4 & 2 \\
\psi & = & 5 & 6 & 7 & 8 & 9 & 3 & 1 & 2 & 4 \\
& & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\end{array}
\]

* except $\psi[1] = SA^{-1}[1]$
Input $\$\text{-terminated text}$ ($\preceq_{\text{lex}} c$ for all $c \in \Sigma$)

\[
S = \text{A T A T A G A T } \ \$ \\
1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9
\]

$\psi$ Array: $\psi[i] = SA^{-1}[SA[i] + 1] \ *

\[
SA = 9 \ 5 \ 7 \ 3 \ 1 \ 6 \ 8 \ 4 \ 2 \\
\psi = 5 \ 6 \ 7 \ 8 \ 9 \ 3 \ 1 \ 2 \ 4 \\
1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9
\]

- Note: $\psi$ is increasing by letter (color).
Input \$-terminated text (\$ \prec_{\text{lex}} c \text{ for all } c \in \Sigma)

\[
S = A \ T \ A \ T \ A \ G \ A \ T \ \$ \\
1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9
\]

\(\psi\) Array: \(\psi[i] = SA^{-1}[SA[i] + 1]\) *

\[
SA = 9 \ 5 \ 7 \ 3 \ 1 \ 6 \ 8 \ 4 \ 2 \\
\psi = 5 \ 6 \ 7 \ 8 \ 9 \ 3 \ 1 \ 2 \ 4 \\
1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9
\]

- Note: \(\psi\) is increasing by letter (color).
- Why? applying \(\psi\) = removing the first char from a suffix. Preserves relative ordering of suffixes starting with same letter
Input $\$-terminated text ($\preccurlyeq_{\text{lex}} c$ for all $c \in \Sigma$)

\[ S = \text{ATAGAT} \]
\[ 1 2 3 4 5 6 7 8 9 \]

\[ \psi \text{ Array: } \psi[i] = SA^{-1}[SA[i] + 1] \]

\[ SA = \begin{array}{cccccccc}
9 & 5 & 7 & 3 & 1 & 6 & 8 & 4 & 2 \\
\end{array} \]

\[ \psi = \begin{array}{cccccccc}
5 & 6 & 7 & 8 & 9 & 3 & 1 & 2 & 4 \\
\end{array} \]

- Note: $\psi$ is increasing by letter (color).
- Why? applying $\psi = \text{removing the first char from a suffix}. \text{Preserves relative ordering of suffixes starting with same letter}$
- Store $\Delta[i] = \psi[i] - \psi[i - 1]$ (delta-encoding): $nH_0 + O(n)$ bits, $O(1)$ random access.
Let’s see how to extract the suffix starting in position $SA[5]$. We store: $\psi$ and first letters (underlined). Space: $nH_0 + O(n)$ bits.

\[
\begin{array}{cccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\psi &=& 5 & 6 & 7 & 8 & 9 & 3 & 1 & 2 & 4 \\
G & T & T & T & T & A & $ & A & A \\
A & $ & A & A & T & G & T \\
T & G & T & $ & A & A \\
$ & A & A & T & G \\
T & G & $ & A \\
$ & A & T \\
T & $ \\
$ \\
\end{array}
\]

Extracted: A
Let’s see how to extract the suffix starting in position \( SA[5] \).
We store: \( \psi \) and first letters (underlined). Space: \( nH_0 + O(n) \) bits.

\[
\begin{array}{cccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\psi &=& 5 & 6 & 7 & 8 & 9 & 3 & 1 & 2 & 4 \\
\$ & A & A & A & A & G & T & T & T \\
G & T & T & T & A & $ & A & A \\
A & $ & A & A & T & G & T \\
T & G & T & $ & A & A \\
$ & A & A & T & G \\
T & G & $ & A \\
$ & A & T \\
T & $ \\
$ \\
\end{array}
\]

Extracted: AT
Let’s see how to extract the suffix starting in position $SA[5]$. We store: $\psi$ and first letters (underlined). Space: $nH_0 + O(n)$ bits.

\[
\begin{array}{cccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\psi = & 5 & 6 & 7 & 8 & 9 & 3 & 1 & 2 & 4 \\
\$ & A & A & A & A & G & T & T & T \\
G & T & T & T & A & $ & A & A \\
A & $ & A & A & T & G & T \\
T & G & T & $ & A & A \\
$ & A & A & T & G \\
T & G & $ & A \\
$ & A & T \\
T & $ \\
$ & 
\end{array}
\]

Extracted: ATA
Let’s see how to extract the suffix starting in position $SA[5]$. We store: $\psi$ and first letters (underlined). Space: $nH_0 + O(n)$ bits.

$$
\psi = \begin{array}{cccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
5 & 6 & 7 & 8 & 9 & 3 & 1 & 2 & 4 \\
\end{array}
$$

$A\ A\ A\ A\ A\ G\ T\ T\ T\ A\ A\ A\ A\ T\ G\ T\ A\ A\ T\ G\ T\ A\ T\ T\ A$  

Extracted: ATAT
The range of suffixes prefixed by a pattern $P$ can be found with binary search using $\psi$. 
The range of suffixes prefixed by a pattern $P$ can be found with binary search using $\psi$.

By sampling the suffix array every $O(\log n)$ text positions, we obtain a **Compressed Suffix Array**.
The Compressed Suffix Array

Trade-offs (later slightly improved):

- **Space**: $nH_0 + O(n)$ bits.
- **Count**: $O(m \log n)$.
- **Locate**: $O((m + occ) \log n)$ (needs a sampling of $SA$)
- **Extract**: $O(\ell + \log n)$ (needs a sampling of $SA^{-1}$)

First described in:

*Grossi, Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In STOC 2000 (pp. 397-406).*
We achieved $nH_0$. What about $nH_k$?
We achieved $nH_0$. What about $nH_k$?

We use an apparently different (but actually equivalent) idea: the Burrows-Wheeler Transform (BWT, Burrows, Wheeler, 1994)
Burrows-Wheeler Transform

Sort all circular permutations of $S = \textit{mississippi}\$. BWT = last column.

<table>
<thead>
<tr>
<th>F</th>
<th>m</th>
<th>i</th>
<th>s</th>
<th>s</th>
<th>s</th>
<th>i</th>
<th>s</th>
<th>s</th>
<th>i</th>
<th>p</th>
<th>p</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>$</td>
<td>m</td>
<td>i</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>i</td>
<td>s</td>
<td>s</td>
<td>i</td>
<td>p</td>
<td>p</td>
</tr>
<tr>
<td>i</td>
<td>p</td>
<td>p</td>
<td>i</td>
<td>$</td>
<td>m</td>
<td>i</td>
<td>s</td>
<td>s</td>
<td>i</td>
<td>s</td>
<td>s</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>s</td>
<td>s</td>
<td>i</td>
<td>p</td>
<td>p</td>
<td>i</td>
<td>$</td>
<td>m</td>
<td>i</td>
<td>s</td>
<td>s</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>s</td>
<td>s</td>
<td>i</td>
<td>s</td>
<td>s</td>
<td>i</td>
<td>p</td>
<td>p</td>
<td>i</td>
<td>$</td>
<td>m</td>
<td></td>
</tr>
<tr>
<td>m</td>
<td>i</td>
<td>s</td>
<td>s</td>
<td>i</td>
<td>s</td>
<td>s</td>
<td>i</td>
<td>p</td>
<td>p</td>
<td>i</td>
<td>$</td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>i</td>
<td>$</td>
<td>m</td>
<td>i</td>
<td>s</td>
<td>s</td>
<td>i</td>
<td>s</td>
<td>s</td>
<td>i</td>
<td>p</td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>p</td>
<td>i</td>
<td>$</td>
<td>m</td>
<td>i</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>i</td>
<td>i</td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>i</td>
<td>p</td>
<td>p</td>
<td>i</td>
<td>$</td>
<td>m</td>
<td>i</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>i</td>
<td>s</td>
<td>s</td>
<td>i</td>
<td>p</td>
<td>p</td>
<td>i</td>
<td>$</td>
<td>m</td>
<td>i</td>
<td>s</td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>s</td>
<td>i</td>
<td>p</td>
<td>p</td>
<td>i</td>
<td>$</td>
<td>m</td>
<td>i</td>
<td>s</td>
<td>s</td>
<td>i</td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>s</td>
<td>i</td>
<td>s</td>
<td>s</td>
<td>i</td>
<td>p</td>
<td>p</td>
<td>i</td>
<td>$</td>
<td>m</td>
<td>i</td>
<td></td>
</tr>
</tbody>
</table>
**LF property.** Let $c \in \Sigma$. Then, the $i$-th occurrence of $c$ in $L$ corresponds to the $i$-th occurrence of $c$ in $F$ (i.e. same position in $T$).

Red arrows: LF function (only character 'i' is shown)
Black arrows: implicit backward links (backward navigation of $T$)
Backward search of the pattern ‘si’

<table>
<thead>
<tr>
<th>F</th>
<th>Unknown</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>mississippi</td>
<td>i</td>
</tr>
<tr>
<td>fr ⇒ i</td>
<td>$mississippi</td>
<td>p</td>
</tr>
<tr>
<td>i ppip$mississippi</td>
<td>s</td>
<td></td>
</tr>
<tr>
<td>i ssippi$mississippi</td>
<td>s</td>
<td></td>
</tr>
<tr>
<td>lr ⇒ i</td>
<td>ssissippi$ m</td>
<td></td>
</tr>
<tr>
<td>m ississippi</td>
<td>$</td>
<td></td>
</tr>
<tr>
<td>p i$mississippi</td>
<td>p</td>
<td></td>
</tr>
<tr>
<td>p pi$mississippi</td>
<td>i</td>
<td></td>
</tr>
</tbody>
</table>

**Step 1:**
Rows prefixed by ‘i’

**Step 2:**
Rows prefixed by ‘si’

Find first and last ‘s’ and apply LF mapping
Finally, note: in BWT, characters are partitioned by context (example: $k = 2$)

We can compress each context independently using a zero-order compressor (e.g. Huffman) and obtain $nH_k$
This structure is known as **FM-index**. Simplified trade-offs (later improved):

- **Space**: $nH_k + o(n \log \sigma)$ bits for $k = \alpha \log_\sigma n - 1$, $0 < \alpha < 1$.
- **Count**: $O(m \log \sigma)$.
- **Locate**: $O(m \log \sigma + \text{occ} \log^{1+\epsilon} n)$ (needs a sampling of $SA$)
- **Extract**: $O(\ell \log \sigma + \log^{1+\epsilon} n)$ (needs a sampling of $SA^{-1}$)

First described (with slightly different trade-offs) in:

*Ferragina, Manzini. Opportunistic data structures with applications. In FOCS 2000, Nov 12 (pp. 390-398).*
The FM index

This structure is known as **FM-index**. Simplified trade-offs (later improved):

- **Space**: $nH_k + o(n \log \sigma)$ bits for $k = \alpha \log \sigma n - 1, \ 0 < \alpha < 1$.
- **Count**: $O(m \log \sigma)$.
- **Locate**: $O(m \log \sigma + \text{occ} \log^{1+\epsilon} n)$ (needs a sampling of $SA$)
- **Extract**: $O(\ell \log \sigma + \log^{1+\epsilon} n)$ (needs a sampling of $SA^{-1}$)

First described (with slightly different trade-offs) in:

*Ferragina, Manzini. Opportunistic data structures with applications. In FOCS 2000, Nov 12 (pp. 390-398).*

**Huge** impact in medicine and bioinformatics: if you get your own genome sequenced, it will be analyzed using software based on the FM-index.
The *compressed indexing* revolution happened in the early 2000s.
The *compressed indexing* revolution happened in the early 2000s.

Then, **the data** changed!
New data

The compressed indexing revolution happened in the early 2000s.

Then, the data changed!

The last decade has been characterized by an explosion in the production of highly repetitive massive data.
The *compressed indexing* revolution happened in the early 2000s.

Then, **the data** changed!

The last decade has been characterized by an explosion in the production of **highly repetitive massive data**

- DNA repositories (1000genomes project, sequencing,...)
The *compressed indexing* revolution happened in the early 2000s.

Then, **the data** changed!

The last decade has been characterized by an explosion in the production of **highly repetitive massive data**

- DNA repositories (1000genomes project, sequencing, ...)
- Versioned repositories (wikipedia, github, ...)

Limitations of entropy became apparent: being memory-less, entropy is insensitive to long repetitions (remember: context length $k$ is small!).

- $H_0(\text{banana}) \approx 1.45$
Limitations of entropy became apparent: being memory-less, entropy is **insensitive to long repetitions** (remember: context length $k$ is small!).

- $H_0(\text{banana}) \approx 1.45$
- $H_0(\text{bananabanana}) \approx 1.45$
Limitations of entropy became apparent: being memory-less, entropy is insensitive to long repetitions (remember: context length $k$ is small!).

- $H_0(\text{banana}) \approx 1.45$
- $H_0(\text{bananabana}) \approx 1.45$
- $H_0(\text{bananabananabanana}) \approx 1.45$
- ...
As a result, $S^3 = \text{bananabananabanana}$ compresses to
\[ |S^3|H(S^3) = 3 \cdot |S|H(S) \text{ bits} \ldots \]
As a result, $S^3 = \text{bananabananabanana}$ compresses to

$|S^3|H(S^3) = 3 \cdot |S|H(S)$ bits ...

Can you come up with a better compressor?
Beating entropy

As a result, $S^3 = \text{bananabananabanana}$ compresses to
$|S^3|H(S^3) = 3 \cdot |S|H(S)$ bits ...

Can you come up with a better compressor?

compress

(3 bananas)

= 

(1 banana) \times 5
As a result, $S^3 = \text{bananabananabanana}$ compresses to $|S^3|H(S^3) = 3 \cdot |S|H(S)$ bits ...

Can you come up with a better compressor?

$|S|H(S) + O(\log t) \ll t \cdot |S|H(S)$ bits.
Dictionary Compression
Ideal compressor: Kolmogorov complexity.
Ideal compressor: Kolmogorov complexity. Non computable/approximable!
Ideal compressor: Kolmogorov complexity. Non computable/approximable!

⇒ We need to fix a text model: exact repetitions
Ideal compressor: Kolmogorov complexity. Non computable/approximable!

⇒ We need to fix a text model: **exact repetitions**

A different generation of compressors comes at rescue: **Dictionary compressors**

General idea:

- Break $S$ into substrings belonging to some dictionary $D$
- Represent $S$ as pointers to $D$
- Usually, $D$ is the set of substrings of $S$ (self-referential compression)
LZ77 (Lempel-Ziv, 1977) — 7-zip, winzip

- LZ77 = Greedy partition of text into shortest factors not appearing before: a|n|na|and|nan|ab|anan|anas|andb|ananas
LZ77 (Lempel-Ziv, 1977) — 7-zip, winzip

- LZ77 = Greedy partition of text into shortest factors not appearing before: a|n|na|and|nan|ab|anan|anas|andb|ananas

- To encode each phrase: just a pointer back, phrase length, and 1 character: $|LZ77| = \mathcal{O}(\# \ of \ phrases)$
LZ77 (Lempel-Ziv, 1977) — 7-zip, winzip

- LZ77 = Greedy partition of text into shortest factors not appearing before: a | n | na | and | nan | ab | anan | anas | andb | ananas
- To encode each phrase: just a pointer back, phrase length, and 1 character: $|LZ77| = \mathcal{O}(\# \text{ of phrases})$
- Compresses orders of magnitude better than entropy on repetitive texts
Run-Length Burrows-Wheeler Transform (RLBWT)

Run-length BWT — bzip2

Input: \( S = \text{BANANA} \)

1. Build the matrix of all circular permutations

\[
\begin{align*}
A & N & A & N & A & $ & B \\
N & A & N & A & $ & B & A \\
N & A & $ & B & A & N & A \\
A & $ & B & A & N & A \\
A & $ & B & A & N & A \\
$ & B & A & N & A & N \\
$ & B & A & N & A & N \\
$ & B & A & N & A & N
\end{align*}
\]

Output: \( \text{RLBWT} = (1, A), (2, N), (1, B), (1, $), (2, A) \)
Run-length BWT — bzip2

Input: $S = \text{BANANA}$

1. Build the matrix of all circular permutations

2. Sort the rows. BWT = last column.

<table>
<thead>
<tr>
<th>Input</th>
<th>Sorted Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>B A N A N A $</td>
<td>$ B A N A N A</td>
</tr>
<tr>
<td>A N A N A $ B</td>
<td>A $ B A N A N</td>
</tr>
<tr>
<td>N A N A $ B A</td>
<td>A N A $ B A N</td>
</tr>
<tr>
<td>A N A $ B A N</td>
<td>A N A N A $ B</td>
</tr>
<tr>
<td>N A $ B A N A</td>
<td>B A N A N A $</td>
</tr>
<tr>
<td>A $ B A N A N</td>
<td>N A $ B A N A</td>
</tr>
<tr>
<td>$ B A N A N A</td>
<td>N A N A $ B A</td>
</tr>
</tbody>
</table>
Run-length BWT — bzip2

Input: $S = \text{BANANA}$

1. Build the matrix of all circular permutations

2. Sort the rows. $\text{BWT} = \text{last column.}$

3. Apply run-length compression to $\text{BWT} = \text{ANNB$AA}$

$$
\begin{array}{cccccc}
B & A & N & A & N & A \\
A & N & A & N & A & $ \\
N & A & N & A & $ & B \\
A & N & A & $ & B & A \\
N & A & $ & B & A & N \\
A & $ & B & A & N & A \\
N & A & $ & B & A & N \\
$ & B & A & N & A & N \\
\end{array}
$$

$\text{BWT}$
Run-length BWT — bzip2

Input: \( S = \text{BANANA} \)

1. Build the matrix of all circular permutations

\[
\begin{array}{cccccc}
B & A & N & A & N & A \\
A & N & A & N & A & $ \\
N & A & N & A & $ & B \\
A & N & A & $ & B & A \\
N & A & $ & B & A & N \\
A & $ & B & A & N & A \\
$ & B & A & N & A & N \\
\end{array}
\]

2. Sort the rows. BWT = last column.

\[
\begin{array}{cccccc}
$ & B & A & N & A & N \\
A & $ & B & A & N & A \\
A & N & A & $ & B & A \\
A & N & A & $ & B & A \\
B & A & N & A & $ & A \\
N & A & $ & B & A & N \\
N & A & N & A & $ & B \\
\end{array}
\]

3. Apply run-length compression to \( BWT = \text{ANNB}$AA \)

Output: \( \text{RLBWT} = (1,A), (2,N), (1,B), (1,$), (2,A) \)
How do these compressors perform in practice?

Real-case example

- All revisions of en.wikipedia.org/wiki/Albert_Einstein
Highly repetitive text collections

How do these compressors perform in practice?

Real-case example

- All revisions of en.wikipedia.org/wiki/Albert_Einstein
- Uncompressed: 456 MB

\[ nH \approx 110 \text{ MB}. \] 4x compression rate.

\[ |RLBWT(T)| \approx 544 \text{ KB}. \] 840x compression rate.

\[ |LZ77(T)| \approx 310 \text{ KB}. \] 1400x compression rate.
Highly repetitive text collections

How do these compressors perform in practice?

Real-case example

- All revisions of en.wikipedia.org/wiki/Albert_Einstein
- Uncompressed: 456 MB
- $nH_5 \approx 110\,MB$. 4x compression rate.
Highly repetitive text collections

How do these compressors perform in practice?

Real-case example

• All revisions of en.wikipedia.org/wiki/Albert_Einstein
• Uncompressed: 456 MB
• $nH_5 \approx 110\, MB$. 4x compression rate.
• $|RLBWT(T)| \approx 544\, KB$. 840x compression rate.
Highly repetitive text collections

How do these compressors perform in practice?

Real-case example

- All revisions of en.wikipedia.org/wiki/Albert_Einstein
- Uncompressed: 456 MB
- $nH_5 \approx 110\text{MB}$. 4x compression rate.
- $|RLBWT(T)| \approx 544\text{KB}$. 840x compression rate.
- $|LZ77(T)| \approx 310\text{KB}$. 1400x compression rate.
Known dictionary compressors (compressed size between parentheses):

1. RLBWT \((r)\)
2. LZ77 \((z)\)
3. macro schemes \((b)\) = bidirectional LZ77 [Storer, Szymanski ’78]
4. SLPs \((g)\) = context-free grammar generating \(S\) [Kieffer, Yang ’00]
5. RLSLPs \((g_{rl})\) = SLPs with run-length rules \(Z \rightarrow A^\ell\) [Nishimoto et al. ’16]
6. collage systems \((c)\) = RLSLPs with substring operator [Kida et al. ’03]
7. word graphs \((e)\) = automata accepting \(S\)'s substrings [Blumer et al. ’87]

(3-6) NP-hard to optimize

Note the zoo of compressibility measures (we’ll come back to this later)
Can we build compressed indexes taking $|RLBWT|$ or $|LZ77|$ space?
Can we build compressed indexes taking $|RLBWT|$ or $|LZ77|$ space?

Notation:

- $r =$ number of equal-letter runs in the BWT
Can we build compressed indexes taking $|RLBWT|$ or $|LZ77|$ space?

Notation:

- $r =$ number of equal-letter runs in the BWT
- $z =$ number of phrases in the Lempel-Ziv parse
Can we build compressed indexes taking $|RLBWT|$ or $|LZ77|$ space?

Notation:

- $r =$ number of equal-letter runs in the BWT
- $z =$ number of phrases in the Lempel-Ziv parse

Note: while it can be proven that $z, r$ are related to $nH_k$, we don’t actually want to do that: we will measure space complexity as a function of $z, r$. 
Given the success of Compressed Suffix Arrays, the first natural try has been to run-length compress them.
# The run-length FM index (RLFM-index)

## 2010: the Run-Length CSA (RLCSA)

<table>
<thead>
<tr>
<th>Name</th>
<th>Space (words/bits)</th>
<th>Count</th>
<th>Locate</th>
<th>Extract</th>
</tr>
</thead>
<tbody>
<tr>
<td>suffix tree ('73)</td>
<td>$O(n)$ words</td>
<td>$O(m)$</td>
<td>$O(m + occ)$</td>
<td>$O(ℓ)$</td>
</tr>
<tr>
<td>suffix array ('93)</td>
<td>2$n$ words + text</td>
<td>$O(m)$</td>
<td>$O(m + occ)$</td>
<td>$O(ℓ)$</td>
</tr>
<tr>
<td>CSA ('00)</td>
<td>$nH_0 + O(n)$ bits</td>
<td>$\tilde{O}(m)$</td>
<td>$\tilde{O}(m + occ)$</td>
<td>$\tilde{O}(ℓ)$</td>
</tr>
<tr>
<td>FM-index ('00)</td>
<td>$nH_k + o(n \log σ)$ bits</td>
<td>$\tilde{O}(m)$</td>
<td>$\tilde{O}(m + occ)$</td>
<td>$\tilde{O}(ℓ)$</td>
</tr>
<tr>
<td><strong>RLCSA ('10)</strong></td>
<td>$O(r + n/d)$ words</td>
<td>$\tilde{O}(m)$</td>
<td>$\tilde{O}(m + occ \cdot d)$</td>
<td>$\tilde{O}(ℓ + d)$</td>
</tr>
</tbody>
</table>

Mäkinen, Navarro, Sirén, and Välimäki. *Storage and retrieval of highly repetitive sequence collections.* Journal of Computational Biology, 2010
## The run-length FM index (RLFM-index)

### 2010: the Run-Length CSA (RLCSA)

<table>
<thead>
<tr>
<th>name</th>
<th>space (words/bits)</th>
<th>Count</th>
<th>Locate</th>
<th>Extract</th>
</tr>
</thead>
<tbody>
<tr>
<td>suffix tree ('73)</td>
<td>$O(n)$ words</td>
<td>$O(m)$</td>
<td>$O(m + \text{occ})$</td>
<td>$O(\ell)$</td>
</tr>
<tr>
<td>suffix array ('93)</td>
<td>$2n$ words + text</td>
<td>$O(m)$</td>
<td>$O(m + \text{occ})$</td>
<td>$O(\ell)$</td>
</tr>
<tr>
<td>CSA ('00)</td>
<td>$nH_0 + O(n)$ bits</td>
<td>$\tilde{O}(m)$</td>
<td>$\tilde{O}(m + \text{occ})$</td>
<td>$\tilde{O}(\ell)$</td>
</tr>
<tr>
<td>FM-index ('00)</td>
<td>$nH_k + o(n \log \sigma)$ bits</td>
<td>$\tilde{O}(m)$</td>
<td>$\tilde{O}(m + \text{occ})$</td>
<td>$\tilde{O}(\ell)$</td>
</tr>
<tr>
<td>RLCSA ('10)</td>
<td>$O(r + n/d)$ words</td>
<td>$\tilde{O}(m)$</td>
<td>$\tilde{O}(m + \text{occ} \cdot d)$</td>
<td>$\tilde{O}(\ell + d)$</td>
</tr>
</tbody>
</table>

*Mäkinen, Navarro, Sirén, and Välimäki. Storage and retrieval of highly repetitive sequence collections. Journal of Computational Biology, 2010*

**Issue:** The trade-off $d$ (sampling rate of the suffix array) makes the index impractical on highly-repetitive texts (where $r \ll n$)
What about Lempel-Ziv indexing?

<table>
<thead>
<tr>
<th>index</th>
<th>compression</th>
<th>space (words)</th>
<th>locate time</th>
</tr>
</thead>
<tbody>
<tr>
<td>KU-LZI[1]</td>
<td>LZ78</td>
<td>$O(z) + n$</td>
<td>$\tilde{O}(m^2 + \text{occ})$</td>
</tr>
<tr>
<td>NAV-LZI[2]</td>
<td>LZ78</td>
<td>$O(z)$</td>
<td>$\tilde{O}(m^3 + \text{occ})$</td>
</tr>
<tr>
<td>KN-LZI[3]</td>
<td>LZ77</td>
<td>$O(z)$</td>
<td>$\tilde{O}(m^2 h + \text{occ})$</td>
</tr>
</tbody>
</table>

$h \leq n$ is the parse height. In practice small, but worst-case $h = \Theta(n)$


How do they work? geometric range search

Example: search splitted-pattern $\overrightarrow{CA}|\overrightarrow{C}$ (to find all splitted occurrences, we have to try all possible splits)

LZ78 = A | C | G | C G | A C | A C A | C A | C G G | T | G G | G T | $

TGGGTGGCACACACAGCGCA
A
ACACACAGCGCA
ACACAGCGCA
CA
CAGCGCA
GCA
GCGCA
GGCACACACAGCGCA
GGTGGCACACACAGCGCA
TGGCACACACAGCGCA
TGGGTGGCACACACAGCGCA$

A
AC
ACA
C
CA
CG
CGG
G
GG
GT
T
1
2
3
5
7
15
16
18
20
12
3
10
2
7
5
20

$
Problems:

- Locate time quadratic in \( m \)
- These index cannot count (without locating)!
The problem has recently (2018) been solved going back to Run-Length CSAs:

Theorem [1]

Let $SA_{[l,...,r]}$ be the suffix array range of a pattern $P$. We can sample $r$ positions of the suffix array (at BWT run-borders) such that:

1. We can return $SA[l]$ in $O(m \log \log n)$ time
2. Given $SA[i]$, we can compute $SA[i+1]$ in $O(\log \log n)$ time.


The problem has recently (2018) been solved going back to Run-Length CSAs:

**Theorem [1]** Let $SA[l, \ldots, r]$ be the suffix array range of a pattern $P$. We can sample $r$ positions of the suffix array (at BWT run-borders) such that:

1. We can return $SA[l]$ in $O(m \log \log n)$ time.
2. Given $SA[i]$, we can compute $SA[i + 1]$ in $O(\log \log n)$ time.


The problem has recently (2018) been solved going back to Run-Length CSAs:

**Theorem [1]** Let $SA[l, \ldots, r]$ be the suffix array range of a pattern $P$. We can sample $r$ positions of the suffix array (at BWT run-borders) such that:

1. We can return $SA[l]$ in $O(m \log \log n)$ time

---


The problem has recently (2018) been solved going back to Run-Length CSAs:

**Theorem [1]** Let $SA[l, \ldots, r]$ be the suffix array range of a pattern $P$. We can sample $r$ positions of the suffix array (at BWT run-borders) such that:

1. We can return $SA[l]$ in $O(m \log \log n)$ time
2. Given $SA[i]$, we can compute $SA[i+1]$ in $O(\log \log n)$ time.


smaller, orders of magnitude faster ($r$-index): the right tool to index thousands of genomes!

- DNA
- boost
- einstein
- world_leaders

+ r-index  ○ rlcsa  △ lzi  × cdawg  ♦ slp  ▲ hyb  ■ fmi-r  ♠ fmi-succ
Exciting results:

- Index size for one human chromosome: 250 MB. 35 bps (bits per symbol).
- Index size for 1000 human chromosomes: 550 MB. **0.08 bps**
- **Faster** than the FM-index.
Up-to-date history of compressed suffix arrays:

<table>
<thead>
<tr>
<th>Name</th>
<th>Space (words/bits)</th>
<th>Count</th>
<th>Locate</th>
<th>Extract</th>
</tr>
</thead>
<tbody>
<tr>
<td>suffix tree ('73)</td>
<td>$\mathcal{O}(n)$ words</td>
<td>$\mathcal{O}(m)$</td>
<td>$\mathcal{O}(m + occ)$</td>
<td>$\mathcal{O}(\ell)$</td>
</tr>
<tr>
<td>suffix array ('93)</td>
<td>$2n$ words + text</td>
<td>$\mathcal{O}(m)$</td>
<td>$\mathcal{O}(m + occ)$</td>
<td>$\mathcal{O}(\ell)$</td>
</tr>
<tr>
<td>CSA ('00)</td>
<td>$nH_0 + \mathcal{O}(n)$ bits</td>
<td>$\tilde{\mathcal{O}}(m)$</td>
<td>$\tilde{\mathcal{O}}(m + occ)$</td>
<td>$\tilde{\mathcal{O}}(\ell)$</td>
</tr>
<tr>
<td>FM-index ('00)</td>
<td>$nH_k + o(n \log \sigma)$ bits</td>
<td>$\tilde{\mathcal{O}}(m)$</td>
<td>$\tilde{\mathcal{O}}(m + occ)$</td>
<td>$\tilde{\mathcal{O}}(\ell)$</td>
</tr>
<tr>
<td>RLCSA ('10)</td>
<td>$\mathcal{O}(r + n/d)$ words</td>
<td>$\tilde{\mathcal{O}}(m)$</td>
<td>$\tilde{\mathcal{O}}(m + occ \cdot d)$</td>
<td>$\tilde{\mathcal{O}}(\ell + d)$</td>
</tr>
<tr>
<td>r-index [1,2] ('18)</td>
<td>$\mathcal{O}(r)$ words</td>
<td>$\tilde{\mathcal{O}}(m)$</td>
<td>$\tilde{\mathcal{O}}(m + occ)$</td>
<td>$\mathcal{O}(\ell + \log(n/r))$*</td>
</tr>
</tbody>
</table>


* only in space $\mathcal{O}(r \log(n/r))$
Current directions
What next?
What next?

- Put some order in the zoo of complexity measures:
  - A definitive measure of "repetitiveness"
  - Relations between existing complexity measures
What next?

• Put some order in the zoo of complexity measures:
  • A definitive measure of "repetitiveness"
  • Relations between existing complexity measures

• Universal (compressor-independent) data structures
What next?

• Put some order in the zoo of complexity measures:
  • A definitive measure of "repetitiveness"
  • Relations between existing complexity measures
• Universal (compressor-independent) data structures
• Generalizations: indexing labeled graphs/regular languages
Universal Compression
String Attractors

String attractors [1]: a tentative to describe all complexity measures under the same framework. Observation:

- A repetitive string $S$ has a small set of distinct substrings $Q = \{S[i..j]\}$
- What if we fix a set of positions $\Gamma \subseteq [1..|S|]$ such that every $s \in Q$ appears in $S$ crossing some position of $\Gamma$?

String Attractors

String attractors [1]: a tentative to describe all complexity measures under the same framework. Observation:

- A repetitive string $S$ has a small set of distinct substrings $Q = \{ S[i..j] \}$
- What if we fix a set of positions $\Gamma \subseteq [1..|S|]$ such that every $s \in Q$ appears in $S$ crossing some position of $\Gamma$?

We call $\Gamma$ “string attractor”. Intuition: few distinct substrings $\Rightarrow$ small $\Gamma$.

Example

\[ S = \text{CDABCCDABCCA} \quad \Gamma = \{4, 7, 11, 12\} \]

in this case, \( \Gamma \) is also the *smallest* attractor ... why?
Main results:

- **Reductions** (universal: work for LZ77, RLBWT, grammars,...) [1]:
  - $|\Gamma| \leq |\text{dictionary compressors}| \leq O(|\Gamma|\text{polylog } n)$

String Attractors

Main results:

- **Reductions** (universal: work for LZ77, RLBWT, grammars,...) [1]:
  - $|\Gamma| \leq |\text{dictionary compressors}| \leq O(|\Gamma| \text{polylog } n)$

- Finding the smallest $\Gamma$ is **NP-complete** and **APX-hard** [1]

Main results:

- **Reductions** (universal: work for LZ77, RLBWT, grammars,...) [1]:
  - \[ \left| \Gamma \right| \leq \left| \text{dictionary compressors} \right| \leq O(\left| \Gamma \right| \text{polylog } n) \]

- Finding the smallest \( \Gamma \) is **NP-complete** and **APX-hard** [1]

- **Optimal** universal data structures of size \( \tilde{O}(\left| \Gamma \right|) \) [1,2,4,5]

---

Main results:

- **Reductions** (universal: work for LZ77, RLBWT, grammars,...) [1]:
  - $|\Gamma| \leq |\text{dictionary compressors}| \leq O(|\Gamma| \text{polylog } n)$
- Finding the smallest $\Gamma$ is **NP-complete** and **APX-hard** [1]
- **Optimal** universal data structures of size $\tilde{O}(|\Gamma|)$ [1,2,4,5]
- FPT algorithms + check if $\Gamma$ is a valid attractor in linear time [3]

Indexing Graphs
Recently, the concept of prefix-sorting has been extended to graphs:

**Wheeler graph [1]**: an edge-labeled graph whose nodes can be prefix-sorted

Recently, the concept of prefix-sorting has been extended to graphs:

**Wheeler graph** [1]: an edge-labeled graph whose nodes can be prefix-sorted

FM-indexes + Wheeler Graphs = **path queries**: find nodes reachable (from any node) by a path labeled \( w \in \Sigma^* \)

$L = (\epsilon|aa)b(ab|b)^*$

Sorted Wheeler automaton:

![Sorted Wheeler automaton diagram]

Note: paths lead to ranges of states (e.g. $a \rightarrow [q_1, q_3]$).
Indexing graphs

Not all graphs are Wheeler, and they are hard to recognize! Main results:

- Hardness results [1]
- Recognizing/sorting Wheeler NFAs (WNFAs) is NP-complete
- Remove min number of edges to obtain a W.G.: APX-complete
- Positive results: Indexing regular languages [2]
  - WNFAs → WDFA with linear blow-up
  - Recognizing/sorting WDFAs in linear time
  - WDFA minimization in $O(n \log n)$ time
  - Any acyclic DFA → smallest WDFA in almost-optimal time

Not all graphs are Wheeler, and they are hard to recognize! Main results:

- **Hardness results [1]**
  - Recognizing/sorting Wheeler NFAs (WNFAs) is NP-complete
  - Remove min number of edges to obtain a W.G.: APX-complete

Indexing graphs

Not all graphs are Wheeler, and they are hard to recognize! Main results:

- **Hardness results [1]**
  - Recognizing/sorting Wheeler NFAs (WNFAs) is NP-complete
  - Remove min number of edges to obtain a W.G.: APX-complete

- **Positive results: Indexing regular languages [2]**
  - $\text{WNFA} \xrightarrow{\text{powerset}} \text{WDFA}$ with linear blow-up
  - Recognizing/sorting WDFAs in linear time
  - WDFA minimization in $O(n \log n)$ time
  - Any acyclic DFA $\rightarrow$ smallest WDFA in almost-optimal time

---

Future Challenges
Future Challenges

What next?

- Index compressed graphs
- Index super-classes of the Wheeler languages
- Better measures of repetitiveness
- Practical compressed indexes (possibly dynamic)
Future Challenges

What next?

- Index compressed graphs
Future Challenges

What next?

- Index compressed graphs
- Index super-classes of the Wheeler languages
Future Challenges

What next?

- Index compressed graphs
- Index super-classes of the Wheeler languages
- Better measures of repetitiveness
Future Challenges

What next?

- Index compressed graphs
- Index super-classes of the Wheeler languages
- Better measures of repetitiveness
- Practical compressed indexes (possibly dynamic)
Thank you for your attention! questions?