Space and Time-Efficient Data Structures for Massive Datasets

Giulio Ermanno Pibiri
giulio.pibiri@di.unipi.it

Supervisor
Rossano Venturini

Computer Science Department
University of Pisa

17/10/2016
The increase of information does not scale with technology.
The increase of information does not scale with technology.

“Software is getting slower more rapidly than hardware becomes faster.”

Niklaus Wirth, A Plea for Lean Software
The increase of information does **not** scale with technology.

“Software is getting slower more rapidly than hardware becomes faster.”

Niklaus Wirth, A Plea for Lean Software
Scenario

Data Structures
PERFORMANCE
“How quickly a program does its work” - faster work

Algorithms
EFFICIENCY
“How much work is required by a program” - less work
Scenario

Data Structures
PERFORMANCE

“how quickly a program does its work” - faster work

Algorithms
EFFICIENCY

“how much work is required by a program” - less work

+ time
- space
Scenario

Data Structures
PERFORMANCE
“how quickly a program does its work” - faster work

Algorithms
EFFICIENCY
“how much work is required by a program” - less work

Data Compression
Scenario

Data Structures
PERFORMANCE
“how quickly a program does its work” - faster work

+ time - space

Data Compression

Algorithms
EFFICIENCY
“how much work is required by a program” - less work

+ space - time
Scenario

Data Structures
PERFORMANCE
“how quickly a program does its work” - faster work

+ time
- space

Algorithms
EFFICIENCY
“how much work is required by a program” - less work

Data Compression
+ space
- time
Small VS Fast?
Small VS Fast?
Choose one.
Small VS Fast?
Choose one.

NO
Data Structures + Data Compression ➞ Faster Algorithms
“Space optimization is closely related to time optimization in a disk memory.”

Donald E. Knuth, The Art of Computer Programming
Hierarchical Memory Organisation

- CPU
- L1
- L2
- RAM
- DISK
Hierarchical Memory Organisation

- CPU
- L1
- L2
- RAM
- DISK

Properties:
- Size
- Speed

- Registers
Hierarchical Memory Organisation

Numbers are taken from: https://gist.github.com/jboner/2841832
Hierarchical Memory Organisation

CPU → L1 → L2 → RAM → DISK

**Size**
- 32-64 bits
- 32 KB
- 256 KB
- 4-32 GB

**Speed**
- 1 ns
- 0.5 ns
- 7 ns
- 100 ns

**Numbers are taken from:** [https://gist.github.com/jboner/2841832](https://gist.github.com/jboner/2841832)
Design **space-efficient** *ad-hoc* data structures, both from a theoretical *and* practical perspective, that support **fast data extraction**.
Design **space-efficient** *ad-hoc* data structures, both from a theoretical *and* practical perspective, that support **fast data extraction**.

Data compression & Fast Retrieval *together*.

Mature algorithmic solutions *now* ready for technology transfer.
Must exploit properties of the addressed problem.

Design **space-efficient** *ad-hoc* data structures, both from a theoretical *and* practical perspective, that support **fast data extraction**.

Data compression & Fast Retrieval *together*.

Mature algorithmic solutions *now* ready for technology transfer.
Because engineered data structures are the ones the *boost* the *availability* and *wealth* of information around us.
Why? Who cares?

Because engineered data structures are the ones the *boost* the *availability* and *wealth* of information around us.

- Inverted Indexes
- N-grams
- B-trees
- ...


Why? Who cares?

Because engineered data structures are the ones the *boost* the *availability* and *wealth* of information around us.

- Inverted Indexes
- N-grams
- B-trees
- ...

Why? Who cares?

Because engineered data structures are the ones that boost the availability and wealth of information around us.

- Inverted Indexes
- N-grams
- B-trees
- …
Why? Who cares?

Because engineered data structures are the ones that boost the availability and wealth of information around us.

- Inverted Indexes
- N-grams
- B-trees
- …
Why? Who cares?

Because engineered data structures are the ones that boost the availability and wealth of information around us.

- Inverted Indexes
- N-grams
- B-trees
- …
Because engineered data structures are the ones that boost the availability and wealth of information around us.

- Inverted Indexes
- N-grams
- B-trees
- …
Why? Who cares?

Because engineered data structures are the ones that boost the availability and wealth of information around us.

- Inverted Indexes
- N-grams
- B-trees
- ...
Wait!

Can’t we use existing libraries?
Wait!

Can’t we use existing libraries?

C++ **Standard Template Library (STL)?**
Wait!

Can’t we use existing libraries?

C++ Standard Template Library (STL)?

std::list
std::stack
std::queue
std::map
std::unordered_map
Wait!

Can’t we use existing libraries?

C++ Standard Template Library (STL)?

std::list
std::stack
std::queue
std::map
std::unordered_map
Can’t we use existing libraries?

C++ Standard Template Library (STL)?

std::list
std::stack
std::queue
std::map
std::unordered_map

Prefer cache-friendly (non discontiguous) data structures. 
Always.

Use std::vector.
Prefer **cache-friendly** data structures.
Example: vector of strings VS string pool

Prefer **cache-friendly** data structures.

**vector of pointers**

**memory**

**vector of offsets**

```plaintext
<table>
<thead>
<tr>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3</td>
</tr>
<tr>
<td>4 5 6 7</td>
</tr>
<tr>
<td>8 9 10 11</td>
</tr>
<tr>
<td>12 13 14 15</td>
</tr>
<tr>
<td>16 17 18 19</td>
</tr>
<tr>
<td>20 21 22 23</td>
</tr>
<tr>
<td>24 25 26 27</td>
</tr>
<tr>
<td>28 29 30 31</td>
</tr>
<tr>
<td>32 33 34 35</td>
</tr>
<tr>
<td>36 37 38 39</td>
</tr>
<tr>
<td>40 41 42</td>
</tr>
</tbody>
</table>
```
Example: vector of strings VS string pool

Prefer **cache-friendly** data structures.

Every access is a cache miss. Pointer chasing :-(

- Vector of pointers
- Vector of offsets
Prefer **cache-friendly** data structures.

**Example: vector of strings VS string pool**

Every access is a cache miss. Pointer chasing :-(

Example: vector of strings VS string pool

Prefer **cache-friendly** data structures.

Every access is a cache miss. Pointer chasing :-(

Example: vector of strings VS string pool

Prefer **cache-friendly** data structures.

Every access is a cache miss. Pointer chasing :-(

Example: vector of strings VS string pool

Prefer **cache-friendly** data structures.

- **vector of pointers**
  - Every access is a cache miss.
  - Pointer chasing :-(

- **vector of offsets**
  - Offsets instead of pointers.
  - Contiguous memory layout.

Code snippet:

```
example@xorg:sxlm/build$ ./string_vector_benchmark 50000000 ~/random_strings.50M.128
2016-10-12 09:43:31: Loading strings
2016-10-12 09:43:39: Scanning strings
3224822962
```
Example: vector of strings VS string pool

Prefer **cache-friendly** data structures.

Every access is a cache miss. Pointer chasing :-(


~5.2GBs
Example: vector of strings VS string pool

Prefer **cache-friendly** data structures.

Every access is a cache miss. Pointer chasing :-(


~5.2GBs
Example: vector of strings VS string pool

Prefer **cache-friendly** data structures.

Every access is a cache miss.
Pointer chasing :-(

Offsets instead of pointers.
Contiguous memory layout.

~5.2GBs
Prefer **cache-friendly** data structures.

Example: vector of strings VS string pool

- Every access is a cache miss.
- Pointer chasing :-(
- Offsets instead of pointers.
- Contiguous memory layout.

```bash
giulio@xor:/sxlm/build$ ./string_vector_benchmark 50000000 ~/random_strings.50M.128
2016-10-12 09:43:31: Loading strings
2016-10-12 09:43:39: Scanning strings
3224822962

~5.2GBs

giulio@xor:/sxlm/build$ ./string_pool_benchmark ~/random_strings.50M.128
2016-10-12 09:44:18: Loading strings
2016-10-12 09:44:26: read 3224822962 bytes
2016-10-12 09:44:26: Scanning strings
3224822962
2016-10-12 09:44:27: elapsed 1.341409 [sec]

~3GBs
Prefer **cache-friendly** data structures.

Example: vector of strings VS string pool

Every access is a cache miss.
Pointer chasing :-(

Offsets instead of pointers.
Contiguous memory layout.

~5.2GBs

~3GBs

X5
Inverted Indexes

Given a *textual collection* $D$, each document can be seen as a (multi-)set of terms. The set of terms occurring in $D$ is the *lexicon* $T$.

For each term $t$ in $T$ we store in a list $L_t$ the identifiers of the documents in which $t$ appears.

The collection of all inverted lists $\{L_{t_1},...,L_{t_T}\}$ is the inverted index.
Inverted Indexes

Given a *textual collection* $D$, each document can be seen as a (multi-)set of terms. The set of terms occurring in $D$ is the *lexicon* $T$.

For each term $t$ in $T$ we store in a list $L_t$ the identifiers of the documents in which $t$ appears.

The collection of all inverted lists $\{L_t, \ldots, L_T\}$ is the inverted index.
Given a *textual collection* $D$, each document can be seen as a (multi-)set of terms. The set of terms occurring in $D$ is the *lexicon* $T$.

For each term $t$ in $T$ we store in a list $L_t$ the identifiers of the documents in which $t$ appears.

The collection of all inverted lists $\{L_{t_1}, \ldots, L_{t_T}\}$ is the inverted index.

$T = \{\text{always, boy, good, house, hungry, is, red, the}\}$
Inverted Indexes

Given a *textual collection* \( D \), each document can be seen as a (multi-)set of terms. The set of terms occurring in \( D \) is the *lexicon* \( T \).

For each term \( t \) in \( T \) we store in a list \( L_t \) the identifiers of the documents in which \( t \) appears.

The collection of all inverted lists \( \{L_{t_1}, \ldots, L_{t_T}\} \) is the inverted index.

\( T = \{\text{always, boy, good, house, hungry, is, red, the}\} \)
Inverted Indexes

Given a *textual collection* $D$, each document can be seen as a (multi-)set of terms. The set of terms occurring in $D$ is the *lexicon* $T$.

For each term $t$ in $T$ we store in a list $L_t$ the identifiers of the documents in which $t$ appears.

The collection of all inverted lists $\{L_t\}_{t \in T}$ is the inverted index.

$$T = \{\text{always, boy, good, house, hungry, is, red, the}\}$$

$$L_t = \begin{cases} [1, 3] & t_1 = \text{red} \\ [4, 5] & t_2 = \text{is} \\ [1] & t_3 = \text{always} \\ [2, 3] & t_4 = \text{boy} \\ [3, 5] & t_5 = \text{good} \\ [1, 2, 3, 4, 5] & t_6 = \text{house} \\ [1, 2, 4] & t_7 = \text{hungry} \\ [2, 3, 5] & t_8 = \text{the} \end{cases}$$
Inverted Indexes owe their popularity to the efficient resolution of queries, such as: “return me all documents in which terms \( \{t_1, \ldots, t_k\} \) occur”.
Inverted Indexes owe their popularity to the *efficient resolution of queries*, such as: “return me all documents in which terms \( \{t_1, \ldots, t_k\} \) occur”.

\[
T = \{\text{always, boy, good, house, hungry, is, red, the}\}
\]

\[
\begin{align*}
L_{t_1} &= [1, 3] \\
L_{t_2} &= [4, 5] \\
L_{t_3} &= [1] \\
L_{t_4} &= [2, 3] \\
L_{t_5} &= [3, 5] \\
L_{t_6} &= [1, 2, 3, 4, 5] \\
L_{t_7} &= [1, 2, 4] \\
L_{t_8} &= [2, 3, 5]
\end{align*}
\]
Inverted Indexes owe their popularity to the *efficient resolution of queries*, such as: “return me all documents in which terms \{t_1, \ldots, t_k\} occur”.

\[ T = \{\text{always, boy, good, house, hungry, is, red, the}\} \]

\[ q = \{\text{boy, is, the}\} \]

\[ L_{t_1} = [1, 3] \]
\[ L_{t_2} = [4, 5] \]
\[ L_{t_3} = [1] \]
\[ L_{t_4} = [2, 3] \]
\[ L_{t_5} = [3, 5] \]
\[ L_{t_6} = [1, 2, 3, 4, 5] \]
\[ L_{t_7} = [1, 2, 4] \]
\[ L_{t_8} = [2, 3, 5] \]
Inverted Indexes owe their popularity to the \textit{efficient resolution of queries}, such as: “return me all documents in which terms \{t_1, \ldots, t_k\} occur”.

\begin{align*}
T &= \{\text{always, boy, good, house, hungry, is, red, the}\} \\
q &= \{\text{boy, is, the}\}
\end{align*}
Inverted Indexes owe their popularity to the *efficient resolution of queries*, such as: “return me all documents in which terms \( \{t_1, \ldots, t_k\} \) occur”.

\[
T = \{\text{always, boy, good, house, hungry, is, red, the}\}
\]

\[
q = \{\text{boy, is, the}\}
\]

\[
\begin{align*}
L_{t_1} &= [1, 3] \\
L_{t_2} &= [4, 5] \\
L_{t_3} &= [1] \\
L_{t_4} &= [2, 3] \\
L_{t_5} &= [3, 5] \\
L_{t_6} &= [1, 2, 3, 4, 5] \\
L_{t_7} &= [1, 2, 4] \\
L_{t_8} &= [2, 3, 5]
\end{align*}
\]
General Problem

Consider a sequence $S[0,n)$ of $n$ positive and monotonically increasing integers, i.e., $S[i] \leq S[i+1]$ for $0 \leq i < n-1$, possibly repeated.

How to represent it as a \textit{bit vector} in which each original integer is \textit{self-delimited}, using as few as possible bits?
Consider a sequence $S[0,n)$ of $n$ positive and monotonically increasing integers, i.e., $S[i] \leq S[i+1]$ for $0 \leq i < n-1$, possibly repeated.

How to represent it as a bit vector in which each original integer is self-delimited, using as few as possible bits?

Huge research corpora describing different space/time trade-offs:

- Elias gamma/delta [Elias, TIT 1975]
- Variable Byte [Salomon, Springer 2007]
- Binary Interpolative Coding [Moffat and Stuiver, IRJ 2000]
- Simple-9 [Anh and Moffat, IRJ 2005]
- PForDelta [Zukowski et al., ICDE 2006]
- OptPFD [Yan et al., WWW 2009]
- Simple-16 [Anh and Moffat, SPE 2010]
- Varint-G8IU [Stepanov et al., CIKM 2011]
- Elias-Fano [Vigna, WSDM 2013]
- Partitioned Elias-Fano [Ottaviano and Venturini, SIGIR 2014]
General Problem

Consider a sequence $S[0,n)$ of $n$ positive and monotonically increasing integers, i.e., $S[i] \leq S[i+1]$ for $0 \leq i < n-1$, possibly repeated.

How to represent it as a bit vector in which each original integer is self-delimited, using as few as possible bits?

Huge research corpora describing different space/time trade-offs.

- Elias gamma/delta [Elias, TIT 1975]
- Variable Byte [Salomon, Springer 2007]
- Binary Interpolative Coding [Moffat and Stuiver, IRJ 2000]
- Simple-9 [Anh and Moffat, IRJ 2005]
- PForDelta [Zukowski et al., ICDE 2006]
- OptPFD [Yan et al., WWW 2009]
- Simple-16 [Anh and Moffat, SPE 2010]
- Varint-G8IU [Stepanov et al., CIKM 2011]
- Elias-Fano [Vigna, WSDM 2013]
- Partitioned Elias-Fano [Ottaviano and Venturini, SIGIR 2014]
For Firefly, Dropbox full text-search engine, speed has always been a priority.

They were unable to scale because of the dimension of their (distributed) inverted index. Consequence? Query time latencies deteriorate from 250ms to 1s.
For Firefly, Dropbox full text-search engine, speed has always been a priority.

They were unable to scale because of the dimension of their (distributed) inverted index. Consequence? Query time latencies deteriorate from 250ms to 1s.
For Firefly, Dropbox full text-search engine, speed has always been a priority.

They were unable to scale because of the dimension of their (distributed) inverted index. Consequence? Query time latencies deteriorate from 250ms to 1s.

Solution?
For Firefly, Dropbox full text-search engine, speed has always been a priority.

They were unable to scale because of the dimension of their (distributed) inverted index. Consequence? Query time latencies deteriorate from 250ms to 1s.

Solution?

Compress the index to reduce I/O pressure.
For Firefly, Dropbox full text-search engine, speed has always been a priority.

They were unable to scale because of the dimension of their (distributed) inverted index. Consequence? Query time latencies deteriorate from 250ms to 1s.

Solution?

Compress the index to reduce I/O pressure.

Data Structures + Data Compression ➔ Faster Algorithms


Elias-Fano - Encoding example

\[ u = 43 \]
### Elias-Fano - Encoding example

<table>
<thead>
<tr>
<th>Number</th>
<th>Binary representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>000011</td>
</tr>
<tr>
<td>4</td>
<td>000100</td>
</tr>
<tr>
<td>7</td>
<td>000111</td>
</tr>
<tr>
<td>13</td>
<td>001101</td>
</tr>
<tr>
<td>14</td>
<td>001110</td>
</tr>
<tr>
<td>15</td>
<td>001111</td>
</tr>
<tr>
<td>21</td>
<td>010101</td>
</tr>
<tr>
<td>43</td>
<td>101011</td>
</tr>
</tbody>
</table>

$u = 43$
Elias-Fano - Encoding example

<table>
<thead>
<tr>
<th>high</th>
<th>low</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lg n )</td>
<td>( \lg(u/n) )</td>
</tr>
<tr>
<td>0 0 0 0 1 1</td>
<td>3</td>
</tr>
<tr>
<td>0 0 0 1 0 0</td>
<td>4</td>
</tr>
<tr>
<td>0 0 0 1 1 1</td>
<td>7</td>
</tr>
<tr>
<td>0 0 1 1 0 1</td>
<td>13</td>
</tr>
<tr>
<td>0 0 1 1 1 0</td>
<td>14</td>
</tr>
<tr>
<td>0 0 1 1 1 1</td>
<td>15</td>
</tr>
<tr>
<td>0 1 0 1 0 1</td>
<td>21</td>
</tr>
<tr>
<td>1 0 1 0 1 1</td>
<td>43</td>
</tr>
</tbody>
</table>

\( u = 43 \)
Elias-Fano - Encoding example

<table>
<thead>
<tr>
<th>high</th>
<th>low</th>
</tr>
</thead>
<tbody>
<tr>
<td>lg n</td>
<td>lg(u/n)</td>
</tr>
<tr>
<td>0 0 0 0 1 1</td>
<td>3 1</td>
</tr>
<tr>
<td>0 0 0 1 0 0</td>
<td>4 2</td>
</tr>
<tr>
<td>0 0 0 1 1 1</td>
<td>7 3</td>
</tr>
<tr>
<td>0 0 1 1 0 1</td>
<td>13 4</td>
</tr>
<tr>
<td>0 0 1 1 1 0</td>
<td>14 5</td>
</tr>
<tr>
<td>0 0 1 1 1 1</td>
<td>15 6</td>
</tr>
<tr>
<td>0 1 0 1 0 1</td>
<td>21 7</td>
</tr>
<tr>
<td>1 0 1 0 1 1</td>
<td>u = 43 8</td>
</tr>
</tbody>
</table>

u = \[43\]
Elias-Fano - Encoding example

\[
\begin{array}{c|c}
\text{high} & \text{low} \\
\hline
\lg n & \lg(u/n) \\
0000011 & 3 \\
0001000 & 4 \\
0001111 & 7 \\
0011011 & 13 \\
0011100 & 14 \\
0011111 & 15 \\
0101111 & 21 \\
1010111 & \text{u = 43}
\end{array}
\]

\[
L = 011100111101110111011011101011
\]
Elias-Fano - Encoding example

L = 011100111101110111101011

u = 43

L = 011100111101110111101011
Elias-Fano - Encoding example

\[
L = 011100111101110111101011
\]

\[
\begin{array}{c|c}
\text{high} & \text{low} \\
\hline
\lg n & \lg(u/n) \\
\hline
0 0 0 0 & 0 1 1 \\
0 0 0 1 & 0 0 0 \\
0 0 0 1 & 1 1 1 \\
0 0 1 1 & 0 1 0 \\
0 0 1 1 & 1 1 1 \\
0 1 0 1 & 1 0 1 \\
1 0 1 0 & 1 1 1 \\
\end{array}
\]

\[u = 43\]
Elias-Fano - Encoding example

<table>
<thead>
<tr>
<th>high</th>
<th>low</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\lg n)</td>
<td>(\lg(u/n))</td>
</tr>
<tr>
<td>0 0 0 0</td>
<td>1 1</td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>0 0</td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>1 1</td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>0 1</td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>1 0</td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>1 1</td>
</tr>
<tr>
<td>0 1 0 1</td>
<td>0 1</td>
</tr>
<tr>
<td>1 0 1 0</td>
<td>1 1</td>
</tr>
</tbody>
</table>

\(L = 0111001110111011101011\)

\(u = 43\)
Elias-Fano - Encoding example

L = 011100111101110111101011

u = 43

L = 011100111101110111101011
Elias-Fano - Encoding example

\[ L = 011100111101110111101011 \]

\[ u = 43 \]
Elias-Fano - Encoding example

L = 0111001111011101111010111

u = 43

L = 0111001111011101111010111
Elias-Fano - Encoding example

high

low

$\log n$

$\log(u/n)$

0 0 0 0 1 1

0 0 0 1 0 0

0 0 0 1 1 1

0 0 1 1 0 1

0 0 1 1 1 0

0 0 1 1 1 1

0 1 0 1 0 1

1 0 1 0 1 1

1 1 0 1 0 1

1 1 1 0 1 1

1 1 1 1 0 0

3 3 1 0 0 1 0 0

$L = 0111001111011101111010111$

$u = 43$

3 1

4 2

7 3

13 4

14 5

15 6

21 7

$L = 0111001111011101111010111$
Elias-Fano - Encoding example

0 1 1
0 1 0 0
0 1 1 0
0 1 1 1
1 0 1 0 1 0 1
1 1 0 1 0 1 1
3 3 1 0 0 1 0 0

H = 1110 1110 10 0 0 10 0 0
L = 0111001111011101111010111101011

u = 43

Elias-Fano - Properties

\[ EF(S[0,n)) = n \left\lfloor \lg \frac{u}{n} \right\rfloor + 2n \text{ bits} \]

\[ EF(S[0,n)) = n \left\lfloor \lg u \right\rfloor + 2n \text{ bits} \]
Elias-Fano - Properties

\[ EF(S[0,n)) = n \left\lfloor \log \frac{u}{n} \right\rfloor + 2n \text{ bits} \]

\( X \) is the set of all monotone sequence of length \( n \) drawn from a universe \( u \).
Elias-Fano - Properties

$$EF(S[0,n)) = n \left\lfloor \lg \frac{u}{n} \right\rfloor + 2n \text{ bits}$$

$\mathcal{X}$ is the set of all monotone sequence of length $n$ drawn from a universe $u$.

$$|\mathcal{X}| = \binom{u+n}{n}$$
\[ EF(S[0,n)) = n \left\lceil \lg \frac{u}{n} \right\rceil + 2n \text{ bits} \]

\( X \) is the set of all monotone sequence of length \( n \) drawn from a universe \( u \).

\[ |X| = \binom{u+n}{n} \]

\[ \left\lceil \lg \left( \frac{u+n}{n} \right) \right\rceil \approx n \lg \frac{u+n}{n} \]
Elias-Fano - Properties

\[
\text{EF}(S[0,n)) = n \left\lfloor \lg \frac{u}{n} \right\rfloor + 2n \text{ bits}
\]

\(\mathcal{X}\) is the set of all monotone sequence of length \(n\) drawn from a universe \(u\).

\[
|\mathcal{X}| = \binom{u+n}{n}
\]

\[
\left\lfloor \lg \left( \frac{u+n}{n} \right) \right\rfloor \approx n \lg \frac{u+n}{n}
\]
Elias-Fano - Properties

\[ EF(S[0,n)) = n \left\lfloor \log \frac{u}{n} \right\rfloor + 2n \] bits

(less than half a bit away [Elias, JACM 1974])

\( X \) is the set of all monotone sequence of length \( n \) drawn from a universe \( u \).

\[ |X| = \binom{u+n}{n} \]

\[ \left\lfloor \log \left( \binom{u+n}{n} \right) \right\rfloor \approx n \log \frac{u+n}{n} \]
Elias-Fano - Properties

\[ EF(S[0,n)) = n \left\lceil \log \frac{u}{n} \right\rceil + 2n \text{ bits} \]

(less than half a bit away [Elias, JACM 1974])

\( \mathcal{X} \) is the set of all monotone sequence of length \( n \) drawn from a universe \( u \).

\[ |\mathcal{X}| = \binom{u+n}{n} \]

\[ \left\lceil \log \left( \frac{u+n}{n} \right) \right\rceil \approx n \log \frac{u+n}{n} \]
Elias-Fano - Properties

\[ EF(S[0,n)) = n \left\lfloor \lg \frac{u}{n} \right\rfloor + 2n \text{ bits} \]

\( X \) is the set of all monotone sequence of length \( n \) drawn from a universe \( u \).

\[ |X| = \binom{u+n}{n} \]

\[ \left\lfloor \lg \left( \binom{u+n}{n} \right) \right\rfloor \approx n \lg \frac{u+n}{n} \]
Elias-Fano - Properties

$$\text{EF}(S[0,n)) = n \left\lceil \log \frac{u}{n} \right\rceil + 2n \text{ bits}$$
Elias-Fano - Properties

$$\text{EF}(S[0,n)) = n \left\lfloor \log \frac{u}{n} \right\rfloor + 2n \text{ bits}$$

access to each $S[i]$ in $O(1)$ worst-case
Elias-Fano - Properties

\[ EF(S[0,n)) = n \left\lceil \log \frac{u}{n} \right\rceil + 2n \text{ bits} \]

access to each \( S[i] \) in \( O(1) \) worst-case

predecessor(\( x \)) = \max\{S[i] \mid S[i] < x\}

successor(\( x \)) = \min\{S[i] \mid S[i] \geq x\}

queries in \( O\left(\log \frac{u}{n}\right) \) worst-case
Elias-Fano - Properties

\[ EF(S[0,n)) = n \left\lceil \lg \frac{u}{n} \right\rceil + 2n \text{ bits} \]

- access to each \( S[i] \) in \( O(1) \) worst-case

\[ \text{predecessor}(x) = \max\{S[i] \mid S[i] < x\} \]
\[ \text{successor}(x) = \min\{S[i] \mid S[i] \geq x\} \]

- queries in \( O\left(\lg \frac{u}{n}\right) \) worst-case
Elias-Fano - Properties

\[
EF(S[0,n)) = n \left\lceil \log \frac{u}{n} \right\rceil + 2n \text{ bits}
\]

access to each \(S[i]\) in \(O(1)\) worst-case

\[
\text{predecessor}(x) = \max\{S[i] \mid S[i] < x\}
\]

\[
\text{successor}(x) = \min\{S[i] \mid S[i] \geq x\}
\]

queries in \(O\left(\log \frac{u}{n}\right)\) worst-case

but…
Elias-Fano - Properties

\[ EF(S[0,n)) = n \lceil \log \frac{u}{n} \rceil + 2n \text{ bits} \]

access to each \( S[i] \) in \( O(1) \) worst-case

\[
\text{predecessor}(x) = \max\{S[i] \mid S[i] < x\} \\
\text{successor}(x) = \min\{S[i] \mid S[i] \geq x\}
\]

queries in \( O\left(\log \frac{u}{n}\right) \) worst-case

but…

need \( o(n) \) bits more to support fast rank/select primitives on bitvector \( H \)
Elias-Fano - Properties

\[
EF(S[0,n)) = n \left\lceil \log \frac{u}{n} \right\rceil + 2n \text{ bits}
\]

access to each \( S[i] \) in \( O(1) \) worst-case

\[
\text{predecessor}(x) = \max\{S[i] \mid S[i] < x\}
\]

\[
\text{successor}(x) = \min\{S[i] \mid S[i] \geq x\}
\]

queries in \( O\left(\log \frac{u}{n}\right) \) worst-case

but…

need \( o(n) \) bits more to support \textit{fast} rank/select primitives on bitvector \( H \)
Clustered Elias-Fano Indexes

Every encoder represents each sequence *individually*.

No exploitation of redundancy.
Clustered Elias-Fano Indexes

Every encoder represents each sequence *individually*.

No exploitation of redundancy.
Clustered Elias-Fano Indexes

Every encoder represents each sequence *individually*.

No exploitation of redundancy.

Idea: encode *clusters* of posting lists.
Clustered Elias-Fano Indexes

cluster of posting lists
Clustered Elias-Fano Indexes

cluster of posting lists

unbounded universe

$\lg u$ bits
Clustered Elias-Fano Indexes

cluster of posting lists

unbounded universe

\( \lg u \) bits

reference list
Clustered Elias-Fano Indexes

- Cluster of posting lists

- Unbounded universe: \( \lg u \) bits

- Reference list: \( R \)
Clustered Elias-Fano Indexes

- Cluster of posting lists
- Unbounded universe
- Reference list

\[ \lg u \text{ bits} \]
Clustered Elias-Fano Indexes

cluster of posting lists

unbounded universe

\[ \lg u \text{ bits} \]

VS

reference list

\[ R \]

\[ R \ll u \]

\[ \lg R \text{ bits} \]
Clustered Elias-Fano Indexes

Problems
1. how to build clusters
2. how to synthesise the reference list
Clustered Elias-Fano Indexes

Problems
1. how to build clusters
2. how to synthesise the reference list

unbounded universe $\lg u$ bits VS $\lg R$ bits

reference list $R \ll u$

NP-hard problem already for a simplified formulation.
Clustered Elias-Fano Indexes

Time VS Space tradeoffs by varying reference size

Figure 2: Bits per posting of Gov2 and ClueWeb09 by varying the reference size.

Table 2: Bits per posting in selected trade-off points.

Table 3: Timings in milliseconds for AND queries on ClueWeb09 and Gov2, using query sets TREC 05 and TREC 05. In parentheses we show the relative percentage against CPEF.
Clustered Elias-Fano Indexes

Time VS Space tradeoffs by varying reference size

Figure 2: Bits per posting of Gov2 and ClueWeb09 by varying the reference size.

Figure 3: Timings for AND queries by varying the reference size on Gov2 and ClueWeb09, using the query set TREC 06.

Table 2: Bits per posting in selected trade-off points.

<table>
<thead>
<tr>
<th></th>
<th>MIN</th>
<th>MID</th>
<th>MAX</th>
<th></th>
<th>MIN</th>
<th>MID</th>
<th>MAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>PEF</td>
<td>2.94 (+5.60%)</td>
<td>2.94 (+5.97%)</td>
<td>2.94 (+10.19%)</td>
<td>PEF</td>
<td>4.80 (+2.13%)</td>
<td>4.80 (+3.09%)</td>
<td>4.80 (+6.25%)</td>
</tr>
<tr>
<td>CPEF</td>
<td>2.78</td>
<td>2.72</td>
<td>2.65</td>
<td>CPEF</td>
<td>4.70</td>
<td>4.62</td>
<td>4.52</td>
</tr>
<tr>
<td>BIC</td>
<td>2.80 (+0.33%)</td>
<td>2.80 (+2.7%)</td>
<td>2.80 (+5.61%)</td>
<td>BIC</td>
<td>4.27 (-5.22%)</td>
<td>4.27 (-7.58%)</td>
<td>4.27 (-5.56%)</td>
</tr>
</tbody>
</table>

(a) Gov2

(b) ClueWeb09

Table 3: Timings in milliseconds for AND queries on ClueWeb09 and Gov2, using query sets TREC 05 and TREC 05. In parentheses we show the relative percentage against CPEF.

<table>
<thead>
<tr>
<th></th>
<th>MIN</th>
<th>MID</th>
<th>MAX</th>
<th></th>
<th>MIN</th>
<th>MID</th>
<th>MAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>PEF</td>
<td>14.6 (-17.9%)</td>
<td>14.6 (-29.2%)</td>
<td>14.6 (-49.7%)</td>
<td>PEF</td>
<td>3.7 (-30.4%)</td>
<td>3.7 (-37.5%)</td>
<td>3.7 (-52.1%)</td>
</tr>
<tr>
<td>CPEF</td>
<td>17.7</td>
<td>20.6</td>
<td>29.1</td>
<td>CPEF</td>
<td>5.3</td>
<td>5.9</td>
<td>7.8</td>
</tr>
<tr>
<td>BIC</td>
<td>41.1 (+131.9%)</td>
<td>41.1 (+199.5%)</td>
<td>41.1 (+41.3%)</td>
<td>BIC</td>
<td>10.5 (+196.2%)</td>
<td>10.5 (+76.2%)</td>
<td>10.5 (+53.0%)</td>
</tr>
<tr>
<td>PEF</td>
<td>17.7 (-14.0%)</td>
<td>17.7 (-29.3%)</td>
<td>17.7 (-40.3%)</td>
<td>PEF</td>
<td>6.1 (-37.4%)</td>
<td>6.1 (-33.2%)</td>
<td>6.1 (-50.1%)</td>
</tr>
<tr>
<td>CPEF</td>
<td>21.2</td>
<td>25.0</td>
<td>35.6</td>
<td>CPEF</td>
<td>8.3</td>
<td>9.3</td>
<td>11.9</td>
</tr>
<tr>
<td>BIC</td>
<td>55.1 (+139.9%)</td>
<td>55.1 (+120.8%)</td>
<td>55.1 (+94.7%)</td>
<td>BIC</td>
<td>18.5 (+322.6%)</td>
<td>18.5 (+316.6%)</td>
<td>18.5 (+266.5%)</td>
</tr>
</tbody>
</table>
### Clustered Elias-Fano Indexes

**Time VS Space tradeoffs by varying reference size**

**Figure 2:** Bits per posting of Gov2 and ClueWeb09 by varying the reference size.

**Figure 3:** Timings for AND queries by varying the reference size on Gov2 and ClueWeb09, using the query set TREC 06.

**Table 2:** Bits per posting in selected trade-off points.

<table>
<thead>
<tr>
<th></th>
<th>MIN</th>
<th>MID</th>
<th>MAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>PEF</td>
<td>2.94</td>
<td>2.94</td>
<td>2.94</td>
</tr>
<tr>
<td>CPEF</td>
<td>2.78</td>
<td>2.72</td>
<td>2.65</td>
</tr>
<tr>
<td>BIC</td>
<td>2.80</td>
<td>2.80</td>
<td>2.80</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>MIN</th>
<th>MID</th>
<th>MAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>PEF</td>
<td>4.80</td>
<td>4.80</td>
<td>4.80</td>
</tr>
<tr>
<td>CPEF</td>
<td>4.70</td>
<td>4.62</td>
<td>4.52</td>
</tr>
<tr>
<td>BIC</td>
<td>4.27</td>
<td>4.27</td>
<td>4.27</td>
</tr>
</tbody>
</table>

**Table 3:** Timings in milliseconds for AND queries on ClueWeb09 and Gov2, using query sets TREC 05 and TREC 05. In parentheses we show the relative percentage against CPEF.

<table>
<thead>
<tr>
<th></th>
<th>MIN</th>
<th>MID</th>
<th>MAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>PEF</td>
<td>14.6</td>
<td>14.6</td>
<td>14.6</td>
</tr>
<tr>
<td>CPEF</td>
<td>17.7</td>
<td>20.6</td>
<td>29.1</td>
</tr>
<tr>
<td>BIC</td>
<td>41.1</td>
<td>41.1</td>
<td>41.1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>MIN</th>
<th>MID</th>
<th>MAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>PEF</td>
<td>17.7</td>
<td>17.7</td>
<td>17.7</td>
</tr>
<tr>
<td>CPEF</td>
<td>21.2</td>
<td>25.0</td>
<td>35.6</td>
</tr>
<tr>
<td>BIC</td>
<td>55.1</td>
<td>55.1</td>
<td>55.1</td>
</tr>
</tbody>
</table>

**Always better than PEF (by up to 11%) and better than BIC (by up to 6.25%)**
Clustered Elias-Fano Indexes

Time VS Space tradeoffs by varying reference size

Figure 2: Bits per posting of Gov2 and ClueWeb09 by varying the reference size.

Table 2: Bits per posting in selected trade-off points.

<table>
<thead>
<tr>
<th></th>
<th>MIN</th>
<th>MID</th>
<th>MAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>PEF</td>
<td>2.94 (+2.32%)</td>
<td>2.94 (+2.99%)</td>
<td>2.94 (+10.93%)</td>
</tr>
<tr>
<td>CPEF</td>
<td>2.78</td>
<td>2.72</td>
<td>2.65</td>
</tr>
<tr>
<td>BIC</td>
<td>2.80 (-0.31%)</td>
<td>2.80 (+2.79%)</td>
<td>2.80 (+5.61%)</td>
</tr>
</tbody>
</table>

(a) Gov2

<table>
<thead>
<tr>
<th></th>
<th>MIN</th>
<th>MID</th>
<th>MAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>PEF</td>
<td>4.80 (+2.13%)</td>
<td>4.80 (+3.98%)</td>
<td>4.80 (+6.25%)</td>
</tr>
<tr>
<td>CPEF</td>
<td>4.70</td>
<td>4.62</td>
<td>4.52</td>
</tr>
<tr>
<td>BIC</td>
<td>4.27 (-5.23%)</td>
<td>4.27 (-7.38%)</td>
<td>4.27 (-3.56%)</td>
</tr>
</tbody>
</table>

(b) ClueWeb09

Always better than PEF (by up to 11%) and better than BIC (by up to 6.25%)

Figure 3: Timings for AND queries by varying the reference size on Gov2 and ClueWeb09, using the query set TREC 06.

Table 3: Timings in milliseconds for AND queries on ClueWeb09 and Gov2, using query sets TREC 05 and TREC 05. In parentheses we show the relative percentage against CPEF.

<table>
<thead>
<tr>
<th></th>
<th>MIN</th>
<th>MID</th>
<th>MAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>PEF</td>
<td>14.6 (-27.3%)</td>
<td>14.6 (-29.2%)</td>
<td>14.6 (-49.7%)</td>
</tr>
<tr>
<td>CPEF</td>
<td>17.7 (+13.5%)</td>
<td>20.6 (+21.4%)</td>
<td>29.1 (+51.7%)</td>
</tr>
<tr>
<td>BIC</td>
<td>41.1 (+49.5%)</td>
<td>41.1 (+49.5%)</td>
<td>41.1 (+49.5%)</td>
</tr>
</tbody>
</table>

(a) ClueWeb09

<table>
<thead>
<tr>
<th></th>
<th>MIN</th>
<th>MID</th>
<th>MAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>PEF</td>
<td>3.7 (-30.4%)</td>
<td>3.7 (-37.3%)</td>
<td>3.7 (-52.1%)</td>
</tr>
<tr>
<td>CPEF</td>
<td>5.3</td>
<td>5.9</td>
<td>7.8</td>
</tr>
<tr>
<td>BIC</td>
<td>10.5 (+18.2%)</td>
<td>10.5 (+18.2%)</td>
<td>10.5 (+18.2%)</td>
</tr>
</tbody>
</table>

(b) Gov2
Clustered Elias-Fano Indexes

Time VS Space tradeoffs by varying reference size

Figure 2: Bits per posting of Gov2 and ClueWeb09 by varying the reference size.

Figure 3: Timings for AND queries by varying the reference size on Gov2 and ClueWeb09, using the query set TREC 06.

Table 2: Bits per posting in selected trade-off points.

Always better than PEF (by up to 11%) and better than BIC (by up to 6.25%)

Much faster than BIC (103% on average) Slightly slower than PEF (20% on average)
Elias-Fano matches the
information theoretic minimum.

\[ n \lg(u/n) + 2n + o(n) \text{ bits} \]
Elias-Fano matches the *information theoretic minimum*.

\[ n \lg(u/n) + 2n + o(n) \text{ bits} \]

- \( O(1) \) random access
- \( O(\lg(u/n)) \) predecessor/successor
Elias-Fano matches the *information theoretic minimum*.

\[ n \log(u/n) + 2n + o(n) \text{ bits} \]

- \( O(1) \) random access
- \( O(\log(u/n)) \) predecessor/successor

*Static* succinct data structure.

NO dynamic updates.
Elias-Fano matches the \textit{information theoretic minimum}.

\[ n \lg(u/n) + 2n + o(n) \text{ bits} \]

- \(O(1)\) random access
- \(O(\lg(u/n))\) predecessor/successor

\textit{Static} succinct data structure.

No dynamic updates.

\begin{itemize}
  \item \textbf{vEB Trees} [van Emde Boas, FOCS 1975]
  \item \textbf{x/y-Fast Tries} [Willard, IPL 1983]
  \item \textbf{Fusion Trees} [Fredman and Willard, JCSS 1993]
  \item \textbf{Exponential Search Trees} [Andersson and Thorup, JACM 2007]
  \item …
\end{itemize}

\begin{itemize}
  \item Dynamic
  \item Most of them take optimal time
\end{itemize}
Integer Data Structures

Elias-Fano matches the *information theoretic minimum*.

\[ n \log(u/n) + 2n + o(n) \text{ bits} \]

- \( O(1) \) random access
- \( O(\log(u/n)) \) predecessor/successor

*Static* succinct data structure.

NO dynamic updates.

- **vEB Trees** [van Emde Boas, FOCS 1975]
- **x/y-Fast Tries** [Willard, IPL 1983]
- **Fusion Trees** [Fredman and Willard, JCSS 1993]
- **Exponential Search Trees** [Andersson and Thorup, JACM 2007]
- ...
The (general) Dictionary problem

The dynamic dictionary problem consists in representing a set $S$ of $n$ objects so that the following operations are supported.

- $\text{insert}(x)$ inserts $x$ in $S$
- $\text{delete}(x)$ deletes $x$ from $S$
- $\text{search}(x)$ checks whether $x$ belongs to $S$
- $\text{minimum}()$ returns the minimum element of $S$
- $\text{maximum}()$ returns the maximum element of $S$
- $\text{predecessor}(x)$ returns $\max\{y \in S : y < x\}$
- $\text{successor}(x)$ returns $\min\{y \in S : y \geq x\}$
The (general) Dictionary problem

The dynamic dictionary problem consists in representing a set $S$ of $n$ objects so that the following operations are supported.

- insert($x$) inserts $x$ in $S$
- delete($x$) deletes $x$ from $S$
- search($x$) checks whether $x$ belongs to $S$
- minimum() returns the minimum element of $S$
- maximum() returns the maximum element of $S$
- predecessor($x$) returns $\max\{y \in S : y < x\}$
- successor($x$) returns $\min\{y \in S : y \geq x\}$
**The (general) Dictionary problem**

The dynamic dictionary problem consists in representing a set $S$ of $n$ objects so that the following operations are supported.

- **insert**(x) inserts x in S
- **delete**(x) deletes x from S
- **search**(x) checks whether x belongs to S
- **minimum**() returns the minimum element of S
- **maximum**() returns the maximum element of S
- **predecessor**(x) returns $\max\{y \in S : y < x\}$
- **successor**(x) returns $\min\{y \in S : y \geq x\}$

**The Dynamic List Representation problem**

[Fredman and Saks, STC 1989]

Given a list $S$ of $n$ sorted integer, support the following operations

- **access**(i) return the i-th smallest element of S
- **insert**(x) inserts x in S
- **delete**(x) deletes x from S

under the assumption that $w \leq \lg^y n$ for some $y$. 
The (general) Dictionary problem

The dynamic dictionary problem consists in representing a set $S$ of $n$ objects so that the following operations are supported.

- **insert**($x$) inserts $x$ in $S$
- **delete**($x$) deletes $x$ from $S$
- **search**($x$) checks whether $x$ belongs to $S$
- **minimum**() returns the minimum element of $S$
- **maximum**() returns the maximum element of $S$
- **predecessor**($x$) returns $\max\{y \in S : y < x\}$
- **successor**($x$) returns $\min\{y \in S : y \geq x\}$

$\Omega((\lg n)/\lg \lg n)$ amortized time per operation, in the cell-probe computational model.

The Dynamic List Representation problem

[Fredman and Saks, STC 1989]

Given a list $S$ of $n$ sorted integer, support the following operations

- **access**($i$) return the $i$-th smallest element of $S$
- **insert**($x$) inserts $x$ in $S$
- **delete**($x$) deletes $x$ from $S$

under the assumption that $w \leq \lg^\gamma n$ for some $\gamma$. 

$\Omega((\lg n)/\lg \lg n)$ amortized time per operation, in the cell-probe computational model.
The (general) Dictionary problem

The dynamic dictionary problem consists in representing a set $S$ of $n$ objects so that the following operations are supported.

- insert($x$) inserts $x$ in $S$
- delete($x$) deletes $x$ from $S$
- search($x$) checks whether $x$ belongs to $S$
- minimum() returns the minimum element of $S$
- maximum() returns the maximum element of $S$
- predecessor($x$) returns $\max\{y \in S : y < x\}$
- successor($x$) returns $\min\{y \in S : y \geq x\}$

$\Omega(\lg n/\lg \lg n)$ amortized time per operation, in the cell-probe computational model.

---

The Dynamic List Representation problem

[Fredman and Saks, STC 1989]

Given a list $S$ of $n$ sorted integer, support the following operations

- access($i$) return the $i$-th smallest element of $S$
- insert($x$) inserts $x$ in $S$
- delete($x$) deletes $x$ from $S$

under the assumption that $w \leq \lg^\gamma n$ for some $\gamma$. 

The (general) Dictionary problem

The dynamic dictionary problem consists in representing a set $S$ of $n$ objects so that the following operations are supported.

- $\text{insert}(x)$ inserts $x$ in $S$
- $\text{delete}(x)$ deletes $x$ from $S$
- $\text{search}(x)$ checks whether $x$ belongs to $S$
- $\text{minimum}()$ returns the minimum element of $S$
- $\text{maximum}()$ returns the maximum element of $S$
- $\text{predecessor}(x)$ returns $\max\{y \in S : y < x\}$
- $\text{successor}(x)$ returns $\min\{y \in S : y \geq x\}$

[Patrascu and Thorup, STC 2007]

Optimal space/time trade-off for a static data structure taking $m = n2^a w$ bits, where $a$ is the number of bits necessary to represent the mean number of bits per integer, i.e., $a = \lg(m/n) - \lg w$

$$\Theta\left(\min\left\{ \log_w n, \log_w \frac{w - \lg n}{a}, \log\left(\frac{\log_w \frac{w}{a}}{\frac{a}{\log n}}\right), \frac{\log_w \frac{w}{a}}{\log\left(\frac{\log_w \frac{w}{a}}{\frac{a}{\log n}}\right)}\right\}\right)$$

The Dynamic List Representation problem

[Fredman and Saks, STC 1989]

Given a list $S$ of $n$ sorted integer, support the following operations

- $\text{access}(i)$ return the $i$-th smallest element of $S$
- $\text{insert}(x)$ inserts $x$ in $S$
- $\text{delete}(x)$ deletes $x$ from $S$

under the assumption that $w \leq \log\gamma n$ for some $\gamma$.

$\Omega\left(\frac{\log n}{\log \log n}\right)$ amortized time per operation, in the cell-probe computational model.
The (general) Dictionary problem

The dynamic dictionary problem consists in representing a set \( S \) of \( n \) objects so that the following operations are supported.

- \( \text{insert}(x) \) inserts \( x \) in \( S \)
- \( \text{delete}(x) \) deletes \( x \) from \( S \)
- \( \text{search}(x) \) checks whether \( x \) belongs to \( S \)
- \( \text{minimum}() \) returns the minimum element of \( S \)
- \( \text{maximum}() \) returns the maximum element of \( S \)
- \( \text{predecessor}(x) \) returns \( \max\{y \in S : y < x\} \)
- \( \text{successor}(x) \) returns \( \min\{y \in S : y \geq x\} \)

[Patrascu and Thorup, STC 2007]

Optimal space/time trade-off for a static data structure taking \( m = n2^a w \) bits, where \( a \) is the number of bits necessary to represent the mean number of bits per integer, i.e., \( a = \lg(m/n) - \lg w \)

\[
\Theta\left( \min\left\{ \log_w n, \frac{w - \lg n}{a}, \frac{\lg \frac{w}{a}}{\lg \left( \frac{a}{\lg n} \log \frac{w}{a} \right)}, \frac{\lg \frac{w}{a}}{\lg \left( \log \frac{w}{a} / \log \frac{\lg n}{a} \right)} \right\} \right)
\]

The Dynamic List Representation problem

[Fredman and Saks, STC 1989]

Given a list \( S \) of \( n \) sorted integer, support the following operations

- \( \text{access}(i) \) return the \( i \)-th smallest element of \( S \)
- \( \text{insert}(x) \) inserts \( x \) in \( S \)
- \( \text{delete}(x) \) deletes \( x \) from \( S \)

under the assumption that \( w \leq \lg^\gamma n \) for some \( \gamma \).

\( \Omega(\lg n/\lg \lg n) \) amortized time per operation, in the cell-probe computational model.
Goals
Dynamic Integer Sets in Succinct Space and Optimal Time

Goals

$$n \lg(u/n) + 2n + o(n) \text{ bits}$$
Dynamic Integer Sets in Succinct Space and Optimal Time

Goals

\[ n \lg(u/n) + 2n + o(n) \text{ bits} + o(n) \text{ bits} \]
Dynamic Integer Sets in Succinct Space and Optimal Time

Goals

\[ n \lg(u/n) + 2n + o(n) \text{ bits} + o(n) \text{ bits} \]

negligible redundancy!
Goals


\[ n \lg(u/n) + 2n + o(n) \text{ bits} + o(n) \text{ bits} \]

negligible redundancy!

1. Extend the static Elias-Fano representation to support predecessor and successor queries in optimal worst-case \(O(\lg \lg n)\) time.

2. Maintain \(S\) in a fully dynamic fashion, supporting in optimal worst-case time all the operations defined in the Dynamic Dictionary and Dynamic List Representation problems.
• optimal time/space trade-off for successor search [Patrascu and Thorup, STC 2007]
• y-fast tries [Willard, IPL 1983]

**Theorem 1.** There exists a data structure representing an ordered set $S(n, u)$ of $n$ integers drawn from a polynomial universe of size $u = n^\gamma$, for any $\gamma = \Theta(1)$, that takes $\text{EF}(S(n, u)) + o(n)$ bits of space and supports Access in $O(1)$ worst-case and Predecessor/Successor queries in optimal $O(\min\{1 + \log \frac{u}{n}, \log \log n\})$ worst-case time.
Results - Static Elias-Fano Optimal Successor Queries

- optimal time/space trade-off for successor search [Patrascu and Thorup, STC 2007]
- y-fast tries [Willard, IPL 1983]

Theorem 1. There exists a data structure representing an ordered set $S(n, u)$ of $n$ integers drawn from a polynomial universe of size $u = n^\gamma$, for any $\gamma = \Theta(1)$, that takes $EF(S(n, u)) + o(n)$ bits of space and supports Access in $O(1)$ worst-case and Predecessor/Successor queries in optimal $O(\min\{1 + \log \frac{u}{n}, \log \log n\})$ worst-case time.

Idea: divide the sequence into blocks and use a y-fast trie to index the blocks.
Results - Dynamic Elias-Fano

- **optimal time/space trade-off for successor search** [Patrascu and Thorup, STC 2007]
- **y-fast tries** [Willard, IPL 1983]
- **dynamic prefix-sum data structure** [Bille et al., arXiv preprint 2015]

**Lemma 4.** The total order of the blocks of $C$ can be maintained by using a data structure that takes $\mathcal{O}(\text{polylog } n \cdot \log \log n)$ bits of space and supports the following operations in $\mathcal{O}(\log \log n)$ worst-case time: Search$(x)$ which returns a pointer to the block containing the integer $x$; Access$(i)$ which returns the $i$-th integer of the total order; Insert/Delete of a block.

**Theorem 3.** There exists a data structure representing an ordered set $S(n, u)$ of $n$ integers drawn from a polynomial universe of size $u = n^\gamma$, for any $\gamma = \Theta(1)$, that takes $\text{EF}(S(n, u)) + o(n)$ bits of space and supports: Access in $\mathcal{O}(\log n / \log \log n)$ worst-case; Insert/Delete in $\mathcal{O}(\log n / \log \log n)$ amortized; Minimum/Maximum in $\mathcal{O}(1)$ and Predecessor/Succesor queries in $\mathcal{O}(\min\{1 + \log \frac{u}{n}, \log \log n\})$ worst-case time. These time bounds are optimal.
Results - Dynamic Elias-Fano

- **Idea:** use a 2-level indexing data structure.
  - First level indexes **blocks** using a **y-fast trie** and the **dynamic prefix-sum** data structure by Bille et al.
  - Second level indexes **mini blocks** using the data structure of the Lemma.

**Lemma 4.** The total order of the blocks of $C$ can be maintained by using a data structure that takes $O(\text{polylog } n \cdot \log \log n)$ bits of space and supports the following operations in $O(\log \log n)$ worst-case time: $\text{Search}(x)$ which returns a pointer to the block containing the integer $x$; $\text{Access}(i)$ which returns the $i$-th integer of the total order; Insert/Delete of a block.

**Theorem 3.** There exists a data structure representing an ordered set $S(n, u)$ of $n$ integers drawn from a polynomial universe of size $u = n^\gamma$, for any $\gamma = \Theta(1)$, that takes $EF(S(n, u)) + o(n)$ bits of space and supports: $\text{Access}$ in $O(\log n / \log \log n)$ worst-case; $\text{Insert}/\text{Delete}$ in $O(\log n / \log \log n)$ amortized; $\text{Minimum}/\text{Maximum}$ in $O(1)$ and $\text{Predecessor}/\text{Successor}$ queries in $O(\min\{1 + \log \frac{u}{n}, \log \log n\})$ worst-case time. These time bounds are optimal.
Strings of at most N words.

N typically ranges from 1 to 5.
N-grams

Strings of at most N words.
N typically ranges from 1 to 5.

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As an example, in Chapter 2, we will see two algorithms for sorting. The first, known as insertion sort, takes time roughly equal to \( c_1 n^2 \) to sort \( n \) items, where \( c_1 \) is a constant that does not depend on \( n \). That is, it takes time roughly proportional to \( n^2 \). The second, merge sort, takes time roughly equal to \( c_2 n \log n \), where \( \log n \) stands for \( \log_2 n \) and \( c_2 \) is another constant that also does not depend on \( n \). Insertion sort typically has a smaller constant factor than merge sort, so that \( c_1 < c_2 \). We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size \( n \). Let’s write insertion sort’s running time as \( c_1 n \cdot n \) and merge sort’s running time as \( c_2 n \cdot \log n \). Then we see that where insertion sort has a factor of \( n \) in its running time, merge sort has a factor of \( \log n \), which is much smaller. (For example, when \( n = 1000 \), \( \log n \) is approximately 10, and when \( n \) equals one million, \( \log n \) is approximately only 20.) Although insertion sort usually runs faster than merge sort for small input sizes, once the input size \( n \) becomes large enough, merge sort’s advantage of \( \log n \) vs. \( n \) will more than compensate for the difference in constant factors. No matter how much smaller \( c_1 \) is than \( c_2 \), there will always be a crossover point beyond which merge sort is faster.
Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As an example, in Chapter 2, we will see two algorithms for sorting. The first, known as *insertion sort*, takes time roughly equal to $c_1n^2$ to sort $n$ items, where $c_1$ is a constant that does not depend on $n$. That is, it takes time roughly proportional to $n^2$. The second, *merge sort*, takes time roughly equal to $c_2n \cdot \log n$, where $\log n$ stands for $\log_2 n$ and $c_2$ is another constant that also does not depend on $n$. Insertion sort typically has a smaller constant factor than merge sort, so that $c_1 < c_2$. We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size $n$. Let’s write insertion sort’s running time as $c_1n \cdot n$ and merge sort’s running time as $c_2n \cdot \log n$. Then we see that where insertion sort has a factor of $n$ in its running time, merge sort has a factor of $\log n$, which is much smaller. (For example, when $n = 1000$, $\log n$ is approximately 10, and when $n$ equals one million, $\log n$ is approximately only 20.) Although insertion sort usually runs faster than merge sort for small input sizes, once the input size $n$ becomes large enough, merge sort’s advantage of $\log n$ vs. $n$ will more than compensate for the difference in constant factors. No matter how much smaller $c_1$ is than $c_2$, there will always be a crossover point beyond which merge sort is faster.
N-grams

Strings of at most N words.
N typically ranges from 1 to 5.

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As an example, in Chapter 2, we will see two algorithms for sorting. The first, known as insertion sort, takes time roughly equal to $c_1 n^2$ to sort $n$ items, where $c_1$ is a constant that does not depend on $n$. That is, it takes time roughly proportional to $n^2$. The second, merge sort, takes time roughly equal to $c_2 n \log n$, where $\log n$ stands for $\log_2 n$ and $c_2$ is another constant that also does not depend on $n$. Insertion sort typically has a smaller constant factor than merge sort, so that $c_1 < c_2$.

We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size $n$. Let’s write insertion sort’s running time as $c_1 n \cdot n$ and merge sort’s running time as $c_2 n \cdot \log n$. Then we see that where insertion sort has a factor of $n$ in its running time, merge sort has a factor of $\log n$, which is much smaller. (For example, when $n = 1000$, $\log n$ is approximately 10, and when $n$ equals one million, $\log n$ is approximately only 20.) Although insertion sort usually runs faster than merge sort for small input sizes, once the input size $n$ becomes large enough, merge sort’s advantage of $\log n$ vs. $n$ will more than compensate for the difference in constant factors. No matter how much smaller $c_1$ is than $c_2$, there will always be a crossover point beyond which merge sort is faster.
N-grams

Strings of at most N words.
N typically ranges from 1 to 5.

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As an example, in Chapter 2, we will see two algorithms for sorting. The first, known as insertion sort, takes time roughly equal to $c_1 n^2$ to sort $n$ items, where $c_1$ is a constant that does not depend on $n$. That is, it takes time roughly proportional to $n^2$. The second, merge sort, takes time roughly equal to $c_2 n \log n$, where $\log n$ stands for $\log_2 n$ and $c_2$ is another constant that also does not depend on $n$. Insertion sort typically has a smaller constant factor than merge sort, so that $c_1 < c_2$. We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size $n$. Let’s write insertion sort’s running time as $c_1 n \cdot n$ and merge sort’s running time as $c_2 n \cdot \log n$. Then we see that where insertion sort has a factor of $n$ in its running time, merge sort has a factor of $\log n$, which is much smaller. (For example, when $n = 1000$, $\log n$ is approximately 10, and when $n$ equals one million, $\log n$ is approximately only 20.) Although insertion sort usually runs faster than merge sort for small input sizes, once the input size $n$ becomes large enough, merge sort’s advantage of $\log n$ vs. $n$ will more than compensate for the difference in constant factors. No matter how much smaller $c_1$ is than $c_2$, there will always be a crossover point beyond which merge sort is faster.
Strings of at most N words.
N typically ranges from 1 to 5.

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As an example, in Chapter 2, we will see two algorithms for sorting. The first, known as insertion sort, takes time roughly equal to $c_1 n^2$ to sort $n$ items, where $c_1$ is a constant that does not depend on $n$. That is, it takes time roughly proportional to $n^2$. The second, merge sort, takes time roughly equal to $c_2 n \log n$, where $\log n$ stands for $\log_2 n$ and $c_2$ is another constant that also does not depend on $n$. Insertion sort typically has a smaller constant factor than merge sort, so that $c_1 < c_2$. We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size $n$. Let’s write insertion sort’s running time as $c_1 n \cdot n$ and merge sort’s running time as $c_2 n \cdot \log n$. Then we see that where insertion sort has a factor of $n$ in its running time, merge sort has a factor of $\log n$, which is much smaller. (For example, when $n = 1000$, $\log n$ is approximately 10, and when $n$ equals one million, $\log n$ is approximately only 20.) Although insertion sort usually runs faster than merge sort for small input sizes, once the input size $n$ becomes large enough, merge sort’s advantage of $\log n$ vs. $n$ will more than compensate for the difference in constant factors. No matter how much smaller $c_1$ is than $c_2$, there will always be a crossover point beyond which merge sort is faster.
Strings of at most N words.

N typically ranges from 1 to 5.

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As an example, in Chapter 2, we will see two algorithms for sorting. The first, known as insertion sort, takes time roughly equal to $c_1 n^2$ to sort $n$ items, where $c_1$ is a constant that does not depend on $n$. That is, it takes time roughly proportional to $n^2$. The second, merge sort, takes time roughly equal to $c_2 n \log n$, where $\log n$ stands for $\log_2 n$ and $c_2$ is another constant that also does not depend on $n$. Insertion sort typically has a smaller constant factor than merge sort, so that $c_1 < c_2$.

We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size $n$. Let's write insertion sort's running time as $c_1 n \cdot n$ and merge sort's running time as $c_2 n \cdot \log n$. Then we see that where insertion sort has a factor of $n$ in its running time, merge sort has a factor of $\log n$, which is much smaller. (For example, when $n = 1000$, $\log n$ is approximately 10, and when $n$ equals one million, $\log n$ is approximately only 20.) Although insertion sort usually runs faster than merge sort for small input sizes, once the input size $n$ becomes large enough, merge sort's advantage of $\log n$ vs. $n$ will more than compensate for the difference in constant factors. No matter how much smaller $c_1$ is than $c_2$, there will always be a crossover point beyond which merge sort is faster.

N = 1

different algorithms
N-grams

Strings of at most N words.
N typically ranges from 1 to 5.

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As an example, in Chapter 2, we will see two algorithms for sorting. The first, known as insertion sort, takes time roughly equal to $c_1 n^2$ to sort $n$ items, where $c_1$ is a constant that does not depend on $n$. That is, it takes time roughly proportional to $n^2$. The second, merge sort, takes time roughly equal to $c_2 n \log n$, where $\log n$ stands for $\log_2 n$ and $c_2$ is another constant that also does not depend on $n$. Insertion sort typically has a smaller constant factor than merge sort, so that $c_1 < c_2$. We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size $n$. Let’s write insertion sort’s running time as $c_1 n \cdot n$ and merge sort’s running time as $c_2 n \cdot \log n$. Then we see that where insertion sort has a factor of $n$ in its running time, merge sort has a factor of $\log n$, which is much smaller. (For example, when $n = 1000$, $\log n$ is approximately 10, and when $n$ equals one million, $\log n$ is approximately only 20.) Although insertion sort usually runs faster than merge sort for small input sizes, once the input size $n$ becomes large enough, merge sort’s advantage of $\log n$ vs. $n$ will more than compensate for the difference in constant factors. No matter how much smaller $c_1$ is than $c_2$, there will always be a crossover point beyond which merge sort is faster.
N-grams

Strings of at most N words.
N typically ranges from 1 to 5.

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As an example, in Chapter 2, we will see two algorithms for sorting. The first, known as **insertion sort**, takes time roughly equal to $c_1 n^2$ to sort $n$ items, where $c_1$ is a constant that does not depend on $n$. That is, it takes time roughly proportional to $n^2$. The second, **merge sort**, takes time roughly equal to $c_2 n \log n$, where $\log n$ stands for $\log_2 n$ and $c_2$ is another constant that also does not depend on $n$. Insertion sort typically has a smaller constant factor than merge sort, so that $c_1 < c_2$. We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size $n$. Let's write insertion sort's running time as $c_1 n \cdot n$ and merge sort's running time as $c_2 n \cdot \log n$. Then we see that where insertion sort has a factor of $n$ in its running time, merge sort has a factor of $\log n$, which is much smaller. (For example, when $n = 1000$, $\log n$ is approximately 10, and when $n$ equals one million, $\log n$ is approximately only 20.) Although insertion sort usually runs faster than merge sort for small input sizes, once the input size $n$ becomes large enough, merge sort's advantage of $\log n$ vs. $n$ will more than compensate for the difference in constant factors. No matter how much smaller $c_1$ is than $c_2$, there will always be a crossover point beyond which merge sort is faster.
N-grams

Strings of at most N words.
N typically ranges from 1 to 5.

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As an example, in Chapter 2, we will see two algorithms for sorting. The first, known as insertion sort, takes time roughly equal to \( c_1 n^2 \) to sort \( n \) items, where \( c_1 \) is a constant that does not depend on \( n \). That is, it takes time roughly proportional to \( n^2 \). The second, merge sort, takes time roughly equal to \( c_2 n \log n \), where \( \log n \) stands for \( \log_2 n \) and \( c_2 \) is another constant that also does not depend on \( n \). Insertion sort typically has a smaller constant factor than merge sort, so that \( c_1 < c_2 \). We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size \( n \). Let’s write insertion sort’s running time as \( c_1 n \cdot n \) and merge sort’s running time as \( c_2 n \cdot \log n \). Then we see that where insertion sort has a factor of \( n \) in its running time, merge sort has a factor of \( \log n \), which is much smaller. (For example, when \( n = 1000 \), \( \log n \) is approximately 10, and when \( n \) equals one million, \( \log n \) is approximately only 20.) Although insertion sort usually runs faster than merge sort for small input sizes, once the input size \( n \) becomes large enough, merge sort’s advantage of \( \log n \) vs. \( n \) will more than compensate for the difference in constant factors. No matter how much smaller \( c_1 \) is than \( c_2 \), there will always be a crossover point beyond which merge sort is faster.
N-grams

Strings of at most N words.
N typically ranges from 1 to 5.

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As an example, in Chapter 2, we will see two algorithms for sorting. The first, known as insertion sort, takes time roughly equal to \( c_1 n^2 \) to sort \( n \) items, where \( c_1 \) is a constant that does not depend on \( n \). That is, it takes time roughly proportional to \( n^2 \). The second, merge sort, takes time roughly equal to \( c_2 n \log_2 n \), where \( \log_2 n \) stands for \( \log_2 n \) and \( c_2 \) is another constant that also does not depend on \( n \). Insertion sort typically has a smaller constant factor than merge sort, so that \( c_1 < c_2 \). We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size \( n \). Let's write insertion sort's running time as \( c_1 n \cdot n \) and merge sort's running time as \( c_2 n \cdot \log_2 n \). Then we see that where insertion sort has a factor of \( n \) in its running time, merge sort has a factor of \( \log_2 n \), which is much smaller. (For example, when \( n = 1000 \), \( \log_2 n \) is approximately 10, and when \( n \) equals one million, \( \log_2 n \) is approximately only 20.) Although insertion sort usually runs faster than merge sort for small input sizes, once the input size \( n \) becomes large enough, merge sort's advantage of \( \log_2 n \) vs. \( n \) will more than compensate for the difference in constant factors. No matter how much smaller \( c_1 \) is than \( c_2 \), there will always be a crossover point beyond which merge sort is faster.
N-grams

Strings of at most N words.
N typically ranges from 1 to 5.

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As an example, in Chapter 2, we will see two algorithms for sorting. The first, known as insertion sort, takes time roughly equal to \( c_1 n^2 \) to sort \( n \) items, where \( c_1 \) is a constant that does not depend on \( n \). That is, it takes time roughly proportional to \( n^2 \). The second, merge sort, takes time roughly equal to \( c_2 n \lg n \), where \( \lg n \) stands for \( \log_2 n \) and \( c_2 \) is another constant that also does not depend on \( n \). Insertion sort typically has a smaller constant factor than merge sort, so that \( c_1 < c_2 \).

We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size \( n \). Let’s write insertion sort’s running time as \( c_1 n \cdot n \) and merge sort’s running time as \( c_2 n \cdot \lg n \). Then we see that where insertion sort has a factor of \( n \) in its running time, merge sort has a factor of \( \lg n \), which is much smaller. (For example, when \( n = 1000 \), \( \lg n \) is approximately 10, and when \( n \) equals one million, \( \lg n \) is approximately only 20.) Although insertion sort usually runs faster than merge sort for small input sizes, once the input size \( n \) becomes large enough, merge sort’s advantage of \( \lg n \) vs. \( n \) will more than compensate for the difference in constant factors. No matter how much smaller \( c_1 \) is than \( c_2 \), there will always be a crossover point beyond which merge sort is faster.
N-grams

Strings of at most N words.
N typically ranges from 1 to 5.

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As an example, in Chapter 2, we will see two algorithms for sorting. The first, known as insertion sort, takes time roughly equal to \( c_1 n^2 \) to sort \( n \) items, where \( c_1 \) is a constant that does not depend on \( n \). That is, it takes time roughly proportional to \( n^2 \). The second, merge sort, takes time roughly equal to \( c_2 n \lg n \), where \( \lg n \) stands for \( \log_2 n \) and \( c_2 \) is another constant that also does not depend on \( n \). Insertion sort typically has a smaller constant factor than merge sort, so that \( c_1 < c_2 \). We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size \( n \). Let’s write insertion sort’s running time as \( c_1 n \cdot n \) and merge sort’s running time as \( c_2 n \cdot \lg n \). Then we see that where insertion sort has a factor of \( n \) in its running time, merge sort has a factor of \( \lg n \), which is much smaller. (For example, when \( n = 1000 \), \( \lg n \) is approximately 10, and when \( n \) equals one million, \( \lg n \) is approximately only 20.) Although insertion sort usually runs faster than merge sort for small input sizes, once the input size \( n \) becomes large enough, merge sort’s advantage of \( \lg n \) vs. \( n \) will more than compensate for the difference in constant factors. No matter how much smaller \( c_1 \) is than \( c_2 \), there will always be a crossover point beyond which merge sort is faster.
Strings of at most $N$ words.

$N$ typically ranges from 1 to 5.

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As an example, in Chapter 2, we will see two algorithms for sorting. The first, known as insertion sort, takes time roughly equal to $c_1 n^2$ to sort $n$ items, where $c_1$ is a constant that does not depend on $n$. That is, it takes time roughly proportional to $n^2$. The second, merge sort, takes time roughly equal to $c_2 n \lg n$, where $\lg n$ stands for $\log_2 n$ and $c_2$ is another constant that also does not depend on $n$. Insertion sort typically has a smaller constant factor than merge sort, so that $c_1 < c_2$.

We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size $n$. Let’s write insertion sort’s running time as $c_1 n \cdot n$ and merge sort’s running time as $c_2 n \cdot \lg n$. Then we see that where insertion sort has a factor of $n$ in its running time, merge sort has a factor of $\lg n$, which is much smaller. (For example, when $n = 1000$, $\lg n$ is approximately 10, and when $n$ equals one million, $\lg n$ is approximately only 20.) Although insertion sort usually runs faster than merge sort for small input sizes, once the input size $n$ becomes large enough, merge sort’s advantage of $\lg n$ vs. $n$ will more than compensate for the difference in constant factors. No matter how much smaller $c_1$ is than $c_2$, there will always be a crossover point beyond which merge sort is faster.
N-grams

Strings of at most N words.
N typically ranges from 1 to 5.

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As an example, in Chapter 2, we will see two algorithms for sorting. The first, known as insertion sort, takes time roughly equal to \( c_1 n^2 \) to sort \( n \) items, where \( c_1 \) is a constant that does not depend on \( n \). That is, it takes time roughly proportional to \( n^2 \). The second, merge sort, takes time roughly equal to \( c_2 n \lg n \), where \( \lg n \) stands for \( \log_2 n \) and \( c_2 \) is another constant that also does not depend on \( n \). Insertion sort typically has a smaller constant factor than merge sort, so that \( c_1 < c_2 \).

We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size \( n \). Let’s write insertion sort’s running time as \( c_1 n \cdot n \) and merge sort’s running time as \( c_2 n \cdot \lg n \). Then we see that where insertion sort has a factor of \( n \) in its running time, merge sort has a factor of \( \lg n \), which is much smaller. (For example, when \( n = 1000 \), \( \lg n \) is approximately 10, and when \( n \) equals one million, \( \lg n \) is approximately only 20.) Although insertion sort usually runs faster than merge sort for small input sizes, once the input size \( n \) becomes large enough, merge sort’s advantage of \( \lg n \) vs. \( n \) will more than compensate for the difference in constant factors. No matter how much smaller \( c_1 \) is than \( c_2 \), there will always be a crossover point beyond which merge sort is faster.
N-grams

Strings of at most N words.
N typically ranges from 1 to 5.

As an example, in Chapter 2, we will see two algorithms for sorting. The first, known as insertion sort, takes time roughly equal to $c_1 n^2$ to sort $n$ items, where $c_1$ is a constant that does not depend on $n$. That is, it takes time roughly proportional to $n^2$. The second, merge sort, takes time roughly equal to $c_2 n \lg n$, where $\lg n$ stands for $\log_2 n$ and $c_2$ is another constant that also does not depend on $n$. Insertion sort typically has a smaller constant factor than merge sort, so that $c_1 < c_2$. We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size $n$. Let’s write insertion sort’s running time as $c_1 n \cdot n$ and merge sort’s running time as $c_2 n \cdot \lg n$. Then we see that where insertion sort has a factor of $n$ in its running time, merge sort has a factor of $\lg n$, which is much smaller. (For example, when $n = 1000$, $\lg n$ is approximately 10, and when $n$ equals one million, $\lg n$ is approximately only 20.) Although insertion sort usually runs faster than merge sort for small input sizes, once the input size $n$ becomes large enough, merge sort’s advantage of $\lg n$ vs. $n$ will more than compensate for the difference in constant factors. No matter how much smaller $c_1$ is than $c_2$, there will always be a crossover point beyond which merge sort is faster.
N-grams

Strings of at most N words.

N typically ranges from 1 to 5.

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As an example, in Chapter 2, we will see two algorithms for sorting. The first, known as \textit{insertion sort}, takes time roughly equal to \( c_1 n^2 \) to sort \( n \) items, where \( c_1 \) is a constant that does not depend on \( n \). That is, it takes time roughly proportional to \( n^2 \). The second, \textit{merge sort}, takes time roughly equal to \( c_2 n \log n \), where \( \log n \) stands for \( \log_2 n \) and \( c_2 \) is another constant that also does not depend on \( n \). Insertion sort typically has a smaller constant factor than merge sort, so that \( c_1 < c_2 \). We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size \( n \). Let's write insertion sort's running time as \( c_1 n \cdot n \) and merge sort's running time as \( c_2 n \cdot \log n \). Then we see that where insertion sort has a factor of \( n \) in its running time, merge sort has a factor of \( \log n \), which is much smaller. (For example, when \( n = 1000 \), \( \log n \) is approximately 10, and when \( n \) equals one million, \( \log n \) is approximately only 20.) Although insertion sort usually runs faster than merge sort for small input sizes, once the input size \( n \) becomes large enough, merge sort's advantage of \( \log n \) vs. \( n \) will more than compensate for the difference in constant factors. No matter how much smaller \( c_1 \) is than \( c_2 \), there will always be a crossover point beyond which merge sort is faster.
Strings of at most $N$ words.

$N$ typically ranges from 1 to 5.

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As an example, in Chapter 2, we will see two algorithms for sorting. The first, known as insertion sort, takes time roughly equal to $c_1 n^2$ to sort $n$ items, where $c_1$ is a constant that does not depend on $n$. That is, it takes time roughly proportional to $n^2$. The second, merge sort, takes time roughly equal to $c_2 n \log n$, where $\log n$ stands for $\log_2 n$ and $c_2$ is another constant that also does not depend on $n$. Insertion sort typically has a smaller constant factor than merge sort, so that $c_1 < c_2$. We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size $n$. Let’s write insertion sort’s running time as $c_1 n \cdot n$ and merge sort’s running time as $c_2 n \cdot \log n$. Then we see that where insertion sort has a factor of $n$ in its running time, merge sort has a factor of $\log n$, which is much smaller. (For example, when $n = 1000$, $\log n$ is approximately 10, and when $n$ equals one million, $\log n$ is approximately only 20.) Although insertion sort usually runs faster than merge sort for small input sizes, once the input size $n$ becomes large enough, merge sort’s advantage of $\log n$ vs. $n$ will more than compensate for the difference in constant factors. No matter how much smaller $c_1$ is than $c_2$, there will always be a crossover point beyond which merge sort is faster.
Strings of at most N words. N typically ranges from 1 to 5.

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As an example, in Chapter 2, we will see two algorithms for sorting. The first, known as *insertion sort*, takes time roughly equal to $c_1 n^2$ to sort $n$ items, where $c_1$ is a constant that does not depend on $n$. That is, it takes time roughly proportional to $n^2$. The second, *merge sort*, takes time roughly equal to $c_2 n \log n$, where $\log n$ stands for $\log_2 n$ and $c_2$ is another constant that also does not depend on $n$. Insertion sort typically has a smaller constant factor than merge sort, so that $c_1 < c_2$.

We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size $n$. Let’s write insertion sort’s running time as $c_1 n \cdot n$ and merge sort’s running time as $c_2 n \cdot \log n$. Then we see that where insertion sort has a factor of $n$ in its running time, merge sort has a factor of $\log n$, which is much smaller. (For example, when $n = 1000$, $\log n$ is approximately 10, and when $n$ equals one million, $\log n$ is approximately only 20.) Although insertion sort usually runs faster than merge sort for small input sizes, once the input size $n$ becomes large enough, merge sort’s advantage of $\log n$ vs. $n$ will more than compensate for the difference in constant factors. No matter how much smaller $c_1$ is than $c_2$, there will always be a crossover point beyond which merge sort is faster.
N-grams

Strings of at most $N$ words.

$N$ typically ranges from 1 to 5.

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As an example, in Chapter 2, we will see two algorithms for sorting. The first, known as insertion sort, takes time roughly equal to $c_1 n^2$ to sort $n$ items, where $c_1$ is a constant that does not depend on $n$. That is, it takes time roughly proportional to $n^2$. The second, merge sort, takes time roughly equal to $c_2 n \log n$, where $\log n$ stands for $\log_2 n$ and $c_2$ is another constant that also does not depend on $n$. Insertion sort typically has a smaller constant factor than merge sort, so that $c_1 < c_2$. We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size $n$. Let’s write insertion sort’s running time as $c_1 n \cdot n$ and merge sort’s running time as $c_2 n \cdot \log n$. Then we see that where insertion sort has a factor of $n$ in its running time, merge sort has a factor of $\log n$, which is much smaller. (For example, when $n = 1000$, $\log n$ is approximately 10, and when $n$ equals one million, $\log n$ is approximately only 20.) Although insertion sort usually runs faster than merge sort for small input sizes, once the input size $n$ becomes large enough, merge sort’s advantage of $\log n$ vs. $n$ will more than compensate for the difference in constant factors. No matter how much smaller $c_1$ is than $c_2$, there will always be a crossover point beyond which merge sort is faster.
N-grams

Strings of at most N words.
N typically ranges from 1 to 5.

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As an example, in Chapter 2, we will see two algorithms for sorting. The first, known as insertion sort, takes time roughly equal to \( c_1 n^2 \) to sort \( n \) items, where \( c_1 \) is a constant that does not depend on \( n \). That is, it takes time roughly proportional to \( n^2 \). The second, merge sort, takes time roughly equal to \( c_2 n \lg n \), where \( \lg n \) stands for \( \log_2 n \) and \( c_2 \) is another constant that also does not depend on \( n \). Insertion sort typically has a smaller constant factor than merge sort, so that \( c_1 < c_2 \). We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size \( n \). Let’s write insertion sort’s running time as \( c_1 n \cdot n \) and merge sort’s running time as \( c_2 n \cdot \lg n \). Then we see that where insertion sort has a factor of \( n \) in its running time, merge sort has a factor of \( \lg n \), which is much smaller. (For example, when \( n = 1000 \), \( \lg n \) is approximately 10, and when \( n \) equals one million, \( \lg n \) is approximately only 20.) Although insertion sort usually runs faster than merge sort for small input sizes, once the input size \( n \) becomes large enough, merge sort’s advantage of \( \lg n \) vs. \( n \) will more than compensate for the difference in constant factors. No matter how much smaller \( c_1 \) is than \( c_2 \), there will always be a crossover point beyond which merge sort is faster.
N-grams

Strings of at most N words.
N typically ranges from 1 to 5.

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As an example, in Chapter 2, we will see two algorithms for sorting. The first, known as insertion sort, takes time roughly equal to $c_1 n^2$ to sort $n$ items, where $c_1$ is a constant that does not depend on $n$. That is, it takes time roughly proportional to $n^2$. The second, merge sort, takes time roughly equal to $c_2 n \log n$, where $\log n$ stands for $\log_2 n$ and $c_2$ is another constant that also does not depend on $n$. Insertion sort typically has a smaller constant factor than merge sort, so that $c_1 < c_2$. We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size $n$. Let’s write insertion sort’s running time as $c_1 n \cdot n$ and merge sort’s running time as $c_2 n \cdot \log n$. Then we see that where insertion sort has a factor of $n$ in its running time, merge sort has a factor of $\log n$, which is much smaller. (For example, when $n = 1000$, $\log n$ is approximately 10, and when $n$ equals one million, $\log n$ is approximately only 20.) Although insertion sort usually runs faster than merge sort for small input sizes, once the input size $n$ becomes large enough, merge sort’s advantage of $\log n$ vs. $n$ will more than compensate for the difference in constant factors. No matter how much smaller $c_1$ is than $c_2$, there will always be a crossover point beyond which merge sort is faster.

<table>
<thead>
<tr>
<th>N</th>
<th>number of grams</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8761</td>
</tr>
<tr>
<td>2</td>
<td>38900</td>
</tr>
<tr>
<td>3</td>
<td>61516</td>
</tr>
<tr>
<td>4</td>
<td>70186</td>
</tr>
<tr>
<td>5</td>
<td>73187</td>
</tr>
</tbody>
</table>
N-grams

Strings of at most N words.
N typically ranges from 1 to 5.
N-grams

Strings of at most N words. 
N typically ranges from 1 to 5. 

~6% of the books ever published
**N-grams**

Strings of at most N words.

N typically ranges from 1 to 5.

---

**Google Books**

~6% of the books ever published

<table>
<thead>
<tr>
<th>N</th>
<th>number of grams</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>24,359,473</td>
</tr>
<tr>
<td>2</td>
<td>667,284,771</td>
</tr>
<tr>
<td>3</td>
<td>7,397,041,901</td>
</tr>
<tr>
<td>4</td>
<td>1,644,807,896</td>
</tr>
<tr>
<td>5</td>
<td>1,415,355,596</td>
</tr>
</tbody>
</table>

More than 11 billion grams.
Word prediction.
N-grams - Why?

Word prediction.

space and time-efficient ?
N-grams - Why?

Word prediction.

space and time-efficient

context
N-grams - Why?

Word prediction.

space and time-efficient

context

? algorithms

foo

data structures

bar

baz
Word prediction.

N-grams - Why?

space and time-efficient

context

? -

algorithms

foo

data structures

bar

baz

frequency count

1214

2

3647

3

1
N-grams - Why?

Word prediction.

\[
P(\text{"data structures"} | \text{"space and time-efficient"}) \approx \frac{f(\text{"space and time-efficient data structures"})}{f(\text{"space and time-efficient"})}
\]

space and time-efficient

context

\text{frequency count}
\begin{array}{c}
\text{algorithms} \\
\text{foo} \\
\text{data structures} \\
\text{bar} \\
\text{baz}
\end{array}
\begin{array}{c}
1214 \\
2 \\
3647 \\
3 \\
1
\end{array}
All Our N-gram are Belong to You
Thursday, August 03, 2006

Posted by Alex Franz and Thorsten Brants, Google Machine Translation Team

Here at Google Research we have been using word n-gram models for a variety of R&D projects, such as statistical machine translation, speech recognition, spelling correction, entity detection, information extraction, and others. While such models have usually been estimated from training corpora containing at most a few billion words, we have been harnessing the vast power of Google's datacenters and distributed processing infrastructure to process larger and larger training corpora. We found that there's no data like more data, and scaled up the size of our data by one order of magnitude, and then another, and then one more - resulting in a training corpus of one trillion words from public Web pages.
All Our N-gram are Belong to You

Thursday, August 03, 2006

Posted by Alex Franz and Thorsten Brants, Google Machine Translation Team

Here at Google Research we have been using word n-gram models for a variety of R&D projects, such as statistical machine translation, speech recognition, spelling correction, entity detection, information extraction, and others. While such models have usually been estimated from training corpora containing at most a few billion words, we have been harnessing the vast power of Google's datacenters and distributed processing infrastructure to process larger and larger training corpora. We found that there's no data like more data, and scaled up the size of our data by one order of magnitude, and then another, and then one more - resulting in a training corpus of one trillion words from public Web pages.
All Our N-gram are Belong to You

Thursday, August 03, 2006

Posted by Alex Franz and Thorsten Brants, Google Machine Translation Team

Here at Google Research we have been using word **n-gram models** for a variety of R&D projects, such as **statistical machine translation**, speech recognition, **spelling correction**, entity detection, information extraction, and others. While such models have usually been estimated from training corpora containing at most a few billion words, we have been harnessing the vast power of Google's datacenters and distributed processing **infrastructure** to process larger and larger training corpora. We found that there's no data like more data, and scaled up the size of our data by one order of magnitude, and then another, and then one more - resulting in a training corpus of **one trillion words** from public Web pages.
What can I help you with?
What can I help you with?

Siri
Store massive N-grams datasets such that given a pattern, we can return its frequency count at light speed.
Store massive N-grams datasets such that given a pattern, we can return its frequency count at light speed.

Efficient map.
Store massive N-grams datasets such that given a pattern, we can return its frequency count at light speed.

**Efficient map.**

Data Structures + Data Compression ➔ Faster Algorithms
Open-addressing  VS  Tries
# N-grams - Data structures

## Open-addressing VS Tries

<table>
<thead>
<tr>
<th>10100101</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>10001011</td>
<td>24</td>
</tr>
<tr>
<td>00001010</td>
<td>582</td>
</tr>
<tr>
<td>11011110</td>
<td>24</td>
</tr>
<tr>
<td>00010101</td>
<td>582</td>
</tr>
<tr>
<td>01010011</td>
<td>36352</td>
</tr>
</tbody>
</table>
Open-addressing vs Tries

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>10100101</td>
<td>24</td>
</tr>
<tr>
<td>10001011</td>
<td>24</td>
</tr>
<tr>
<td>00001010</td>
<td>582</td>
</tr>
<tr>
<td>11011110</td>
<td>24</td>
</tr>
<tr>
<td>00101010</td>
<td>582</td>
</tr>
<tr>
<td>01010011</td>
<td>36352</td>
</tr>
</tbody>
</table>
Open-addressing VS Tries

+ time
- space

10100101 24
10001011 24
00001010 582
11011110 24
00101010 582
01010011 36352
N-grams - Data structures

Open-addressing VS Tries

- time +
- space -

\[
\begin{array}{|c|c|}
\hline
\text{Key} & \text{Value} \\
\hline
10100101 & 24 \\
10001011 & 24 \\
00001010 & 582 \\
11011110 & 24 \\
00010101 & 582 \\
01010011 & 36352 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|c|c|c|c|c|}
\hline
\text{Node} & \text{A} & \text{B} & \text{C} & \text{D} \\
\hline
24 & 100 & 120 & 90 & 43 & 225 & 350 & 114 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|c|c|c|c|c|}
\hline
\text{Node} & \text{B} & \text{A} & \text{B} & \text{D} & \text{A} & \text{A} & \text{C} & \text{A} & \text{B} & \text{D} & \text{A} \\
\hline
24 & 23 & 34 & 43 & 120 & 73 & 17 & 88 & 123 & 14 & 114 \\
\hline
\end{array}
\]
N-grams - Data structures

Open-addressing

+ time
- space

<table>
<thead>
<tr>
<th>10100101</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>10001011</td>
<td>24</td>
</tr>
<tr>
<td>00001010</td>
<td>582</td>
</tr>
<tr>
<td>11011110</td>
<td>24</td>
</tr>
<tr>
<td>00010101</td>
<td>582</td>
</tr>
<tr>
<td>01010011</td>
<td>36352</td>
</tr>
</tbody>
</table>

VS

Tries

BBC
### Open-addressing vs Tries

**Open-addressing**

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>10100101</td>
<td>24</td>
</tr>
<tr>
<td>10001011</td>
<td>24</td>
</tr>
<tr>
<td>00001010</td>
<td>582</td>
</tr>
<tr>
<td>11011110</td>
<td>24</td>
</tr>
<tr>
<td>00010101</td>
<td>582</td>
</tr>
<tr>
<td>01010011</td>
<td>36352</td>
</tr>
</tbody>
</table>

**Tries**

```
+ time
- space
```

#### BBC

```
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>D</th>
<th>B</th>
<th>C</th>
<th>A</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>100</td>
<td>120</td>
<td>90</td>
<td>43</td>
<td>225</td>
<td>350</td>
<td>114</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>B</th>
<th>A</th>
<th>B</th>
<th>D</th>
<th>A</th>
<th>A</th>
<th>C</th>
<th>A</th>
<th>B</th>
<th>D</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>23</td>
<td>34</td>
<td>43</td>
<td>120</td>
<td>73</td>
<td>17</td>
<td>88</td>
<td>123</td>
<td>14</td>
<td>114</td>
</tr>
</tbody>
</table>
```
N-grams - Data structures

Open-addressing

+ time
- space

<table>
<thead>
<tr>
<th>X</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>10100101</td>
<td>24</td>
</tr>
<tr>
<td>10001011</td>
<td>24</td>
</tr>
<tr>
<td>00001010</td>
<td>582</td>
</tr>
<tr>
<td>11011110</td>
<td>24</td>
</tr>
<tr>
<td>00010101</td>
<td>582</td>
</tr>
<tr>
<td>01010011</td>
<td>36352</td>
</tr>
</tbody>
</table>

VS

Tries

BBC

```plaintext
A  24 100 120 90 43 225 350 114
B  24 34 43 120 73 17 88 123 14 114
D  24 34 43 120 73 17 88 123 14 114
```
Open-addressing

VS

Tries

+ time
- space

N-grams - Data structures
N-grams - Data structures

Open-addressing

VS

Tries

+ time
- space

10100101  24
10001011  24
00001010  582
11011110  24
00010101  582
01010011  36352

X

\( \hat{h} \)

BBC

A B C D
24 133 1 689

A B C
24 90 43

A C D
225 350 114

A B D A A C A B D A
24 100 120 73 17 88 123 14 114

VS + time - space
N-grams - Data structures

Open-addressing

+ time
- space

X

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>10100101</td>
<td>24</td>
</tr>
<tr>
<td>10001011</td>
<td>24</td>
</tr>
<tr>
<td>00001010</td>
<td>582</td>
</tr>
<tr>
<td>11011110</td>
<td>24</td>
</tr>
<tr>
<td>00010101</td>
<td>582</td>
</tr>
<tr>
<td>01010011</td>
<td>36352</td>
</tr>
</tbody>
</table>

VS

Tries

BBC

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>24</td>
<td>100</td>
</tr>
<tr>
<td>133</td>
<td></td>
</tr>
<tr>
<td>689</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>24</td>
<td>23</td>
</tr>
<tr>
<td>43</td>
<td>120</td>
</tr>
<tr>
<td>123</td>
<td>14</td>
</tr>
</tbody>
</table>
N-grams - Data structures

Open-addressing

+ time
- space

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>10100101</td>
<td>24</td>
</tr>
<tr>
<td>10001011</td>
<td>24</td>
</tr>
<tr>
<td>00001010</td>
<td>582</td>
</tr>
<tr>
<td>11011110</td>
<td>24</td>
</tr>
<tr>
<td>00010101</td>
<td>582</td>
</tr>
<tr>
<td>01010011</td>
<td>36352</td>
</tr>
</tbody>
</table>

X

\( h \)

VS

Tries

+ space
- time

BBC
N-grams - Data structures

Open-addressing VS Tries

+ time - space
+ space - time

Several software libraries

- KenLM [Heafield, 2011]
- BerkeleyLM [Pauls and Klein, 2011]
- IRSTLM [Federico et al., 2008]
- RandLM [Talbot and Osborne, 2007]
- Get1T [Hawker et al., 2007]
- SRILM [Stolcke, 2002]
Hash-based VS Trie-based

<table>
<thead>
<tr>
<th>Hash-based</th>
<th>Trie-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>10100101</td>
<td>24</td>
</tr>
<tr>
<td>10001011</td>
<td>24</td>
</tr>
<tr>
<td>00001010</td>
<td>582</td>
</tr>
<tr>
<td>11011110</td>
<td>24</td>
</tr>
<tr>
<td>00010101</td>
<td>582</td>
</tr>
<tr>
<td>01010011</td>
<td>36352</td>
</tr>
</tbody>
</table>
Hash-based

Open addressing?

VS

Trie-based

<table>
<thead>
<tr>
<th>10100101</th>
<th>10001011</th>
<th>00001010</th>
<th>11011110</th>
<th>00010101</th>
<th>01010011</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>24</td>
<td>582</td>
<td>24</td>
<td>582</td>
<td>36352</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>133</td>
<td>1</td>
<td>689</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>D</th>
<th>B</th>
<th>C</th>
<th>A</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>100</td>
<td>120</td>
<td>90</td>
<td>43</td>
<td>225</td>
<td>350</td>
<td>114</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B</th>
<th>A</th>
<th>B</th>
<th>D</th>
<th>A</th>
<th>A</th>
<th>C</th>
<th>A</th>
<th>B</th>
<th>D</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>23</td>
<td>34</td>
<td>43</td>
<td>120</td>
<td>73</td>
<td>17</td>
<td>88</td>
<td>123</td>
<td>14</td>
<td>114</td>
</tr>
</tbody>
</table>
Hash-based

Open addressing?
Data structure is static.

Minimal Perfect Hashing.

VS

Trie-based

<table>
<thead>
<tr>
<th>Hash-based Data Structure</th>
<th>Trie-based Data Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>10100101</td>
<td>24</td>
</tr>
<tr>
<td>10001011</td>
<td>24</td>
</tr>
<tr>
<td>00001010</td>
<td>582</td>
</tr>
<tr>
<td>11011110</td>
<td>24</td>
</tr>
<tr>
<td>00010101</td>
<td>582</td>
</tr>
<tr>
<td>01010011</td>
<td>36352</td>
</tr>
</tbody>
</table>

Diagram showing trie-based data structure with nodes A, B, C, D and values 24, 100, 120, 90, 43, 225, 350, 114.
Hash-based

Open addressing?
Data structure is *static*.

**Minimal Perfect Hashing.**
Hash-based

Open addressing?
Data structure is *static*.

Minimal Perfect Hashing.

<table>
<thead>
<tr>
<th>Binary</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>10100101</td>
<td>24</td>
</tr>
<tr>
<td>10001011</td>
<td>24</td>
</tr>
<tr>
<td>00001010</td>
<td>582</td>
</tr>
<tr>
<td>11011110</td>
<td>24</td>
</tr>
</tbody>
</table>

Trie-based

VS

<table>
<thead>
<tr>
<th>Character</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>244</td>
</tr>
<tr>
<td>B</td>
<td>133</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>689</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Character</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>24</td>
</tr>
<tr>
<td>B</td>
<td>100</td>
</tr>
<tr>
<td>D</td>
<td>120</td>
</tr>
<tr>
<td>B</td>
<td>90</td>
</tr>
<tr>
<td>C</td>
<td>43</td>
</tr>
<tr>
<td>A</td>
<td>225</td>
</tr>
<tr>
<td>C</td>
<td>350</td>
</tr>
<tr>
<td>D</td>
<td>114</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Character</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>24</td>
</tr>
<tr>
<td>A</td>
<td>23</td>
</tr>
<tr>
<td>B</td>
<td>34</td>
</tr>
<tr>
<td>D</td>
<td>43</td>
</tr>
<tr>
<td>A</td>
<td>120</td>
</tr>
<tr>
<td>A</td>
<td>73</td>
</tr>
<tr>
<td>C</td>
<td>17</td>
</tr>
<tr>
<td>A</td>
<td>88</td>
</tr>
<tr>
<td>B</td>
<td>123</td>
</tr>
<tr>
<td>D</td>
<td>14</td>
</tr>
<tr>
<td>A</td>
<td>114</td>
</tr>
</tbody>
</table>
Hash-based

Open addressing?
Data structure is static.
Minimal Perfect Hashing.

VS

Trie-based
Hash-based

Open addressing?
Data structure is *static*.

**Minimal Perfect Hashing.**

---

### Hash-based

<table>
<thead>
<tr>
<th>Hash</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>10100101</td>
<td>24</td>
</tr>
<tr>
<td>10001011</td>
<td>24</td>
</tr>
<tr>
<td>00001010</td>
<td>582</td>
</tr>
<tr>
<td>11011110</td>
<td>24</td>
</tr>
<tr>
<td>00010101</td>
<td>582</td>
</tr>
<tr>
<td>01010011</td>
<td>36352</td>
</tr>
</tbody>
</table>

### Trie-based

```
A   B   C   D
24   133  1   689
24   100  120 90   43   225 350  114
24   23   34  43  120  73   17   88  123 14   114
```

---

VS

---

### Trie-based

```
B   A   B   D   A   A   C   A   B   D   A
24   23   34  43  120  73   17   88  123 14   114
```
Hash-based

Open addressing?
Data structure is *static*.

**Minimal Perfect Hashing.**

```
   10100101  0
   10001011  0
   00001010  1
   11011110  0
   00010101  1
   01010011  2
```

---

Trie-based

```
  A  B  C  D
244 133  1  689

  A  B  D  B  C  A  C  D
24  100 120  90  43 225 350 114
```

---

```
  B  A  B  D  A  A  C  A  B  D  A
24  23  34  43 120  73  17  88 123 14 114
```
tongrams - Tons of N-Grams

Hash-based

Open addressing?
Data structure is static.

Minimal Perfect Hashing.

VS

Trie-based

<table>
<thead>
<tr>
<th>Vocabulary</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>244</td>
<td>133</td>
<td>1</td>
<td>689</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>23</td>
<td>34</td>
<td>43</td>
<td>120</td>
</tr>
<tr>
<td>24</td>
<td>100</td>
<td>120</td>
<td>90</td>
<td>43</td>
</tr>
<tr>
<td>24</td>
<td>100</td>
<td>120</td>
<td>90</td>
<td>43</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>350</td>
<td>114</td>
<td>123</td>
<td>14</td>
<td>114</td>
</tr>
<tr>
<td>350</td>
<td>114</td>
<td>123</td>
<td>14</td>
<td>114</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>D</th>
<th>B</th>
<th>D</th>
<th>A</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>114</td>
<td>133</td>
<td>120</td>
<td>73</td>
<td>17</td>
</tr>
<tr>
<td>114</td>
<td>133</td>
<td>120</td>
<td>73</td>
<td>17</td>
</tr>
<tr>
<td>114</td>
<td>133</td>
<td>120</td>
<td>73</td>
<td>17</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Hash-based

Open addressing?
Data structure is *static*.

**Minimal Perfect Hashing.**

---

VS

Trie-based

<table>
<thead>
<tr>
<th>Vocabulary</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

| 10100101   | 0 |
| 10010101   | 0 |
| 00001010   | 1 |
| 11011110   | 0 |
| 00010101   | 1 |
| 01010011   | 2 |

Hash table:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>24</td>
</tr>
<tr>
<td>1</td>
<td>582</td>
</tr>
<tr>
<td>2</td>
<td>36352</td>
</tr>
</tbody>
</table>

Trie structure:

```
     0   1   2   3
   ___-___-___-___-___-
   |     |     |     |     |
   24   133  1   689
   |     |     |     |
   24   100 120  90  43  225 350 114
   |     |     |     |     |     |     |     |     |
   24   23   34   43 120  73  17  88 123 14 114
```
Hash-based

Open addressing?
Data structure is static.

Minimal Perfect Hashing.

VS

Trie-based

<table>
<thead>
<tr>
<th>vocabulary</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>3</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>244</td>
<td>133</td>
<td>1</td>
<td>689</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>5</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>100</td>
<td>120</td>
<td>90</td>
<td>43</td>
<td>225</td>
<td>350</td>
<td>114</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>4</th>
<th>4</th>
<th>6</th>
<th>6</th>
<th>7</th>
<th>9</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>23</td>
<td>34</td>
<td>43</td>
<td>120</td>
<td>73</td>
<td>17</td>
<td>88</td>
<td>123</td>
<td>14</td>
<td>114</td>
</tr>
</tbody>
</table>
**Hash-based**

Open addressing?
Data structure is *static*.

**Minimal Perfect Hashing.**

---

**Trie-based**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>3</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>133</td>
<td>1</td>
<td>689</td>
</tr>
<tr>
<td>24</td>
<td>100</td>
<td>120</td>
<td>90</td>
</tr>
<tr>
<td>43</td>
<td>225</td>
<td>350</td>
<td>114</td>
</tr>
</tbody>
</table>

Encode each level with **Elias-Fano.**
Hash-based

Open addressing?
Data structure is static.

Minimal Perfect Hashing.

VS

Trie-based

<table>
<thead>
<tr>
<th>Vocabulary</th>
<th>A → 0</th>
<th>B → 1</th>
<th>C → 2</th>
<th>D → 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Encode each level with **Elias-Fano**.
Random access.
### tongrams - Preliminary results

<table>
<thead>
<tr>
<th>n</th>
<th>Number of n-grams</th>
<th>Maximum frequency count</th>
<th>Unique frequency counts</th>
<th>$[log]$ of unique frequency counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>24,359,472</td>
<td>468,491,999,592</td>
<td>246,588</td>
<td>18</td>
</tr>
<tr>
<td>2</td>
<td>5,089,239</td>
<td>155,178,163</td>
<td>44,822</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>52,635,338</td>
<td>102,329,901</td>
<td>71,690</td>
<td>17</td>
</tr>
<tr>
<td>4</td>
<td>11,149,161</td>
<td>6,401,274</td>
<td>21,127</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>8,261,975</td>
<td>958,556</td>
<td>12,171</td>
<td>14</td>
</tr>
<tr>
<td>Total</td>
<td>101,495,185</td>
<td>468,491,999,592</td>
<td>266,760</td>
<td>19</td>
</tr>
</tbody>
</table>

Table 4: Basic statistics for the GoogleWeb1T subset.

<table>
<thead>
<tr>
<th>HASH</th>
<th>Total space in GBs</th>
<th>Bytes per gram</th>
<th>Lookup time $[\mu s]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>KenLM</td>
<td>2.570</td>
<td>27.19</td>
<td>0.248</td>
</tr>
<tr>
<td>sxlm</td>
<td><strong>1.012</strong></td>
<td><strong>10.43</strong></td>
<td><strong>0.242</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TRIE</th>
<th>Total space in GBs</th>
<th>Bytes per gram</th>
<th>Lookup time $[\mu s]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>KenLM</td>
<td>1.829</td>
<td>21.5</td>
<td>1.272</td>
</tr>
<tr>
<td>sxlm</td>
<td><strong>0.541</strong></td>
<td><strong>5.7</strong></td>
<td><strong>1.229</strong></td>
</tr>
</tbody>
</table>

Table 5: Bytes per grams and average lookup time in $\mu s$ for the GoogleWeb1T subset.
tongrams - Preliminary results

<table>
<thead>
<tr>
<th>$n$</th>
<th>Number of $n$-grams</th>
<th>Maximum frequency count</th>
<th>Unique frequency counts</th>
<th>$\lfloor \log \rfloor$ of unique frequency counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>24,359,472</td>
<td>468,491,999,592</td>
<td>246,588</td>
<td>18</td>
</tr>
<tr>
<td>2</td>
<td>5,089,239</td>
<td>155,178,163</td>
<td>44,822</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>52,635,338</td>
<td>102,329,901</td>
<td>71,690</td>
<td>17</td>
</tr>
<tr>
<td>4</td>
<td>11,149,161</td>
<td>6,401,274</td>
<td>21,127</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>8,261,975</td>
<td>958,556</td>
<td>12,171</td>
<td>14</td>
</tr>
<tr>
<td>Total</td>
<td>101,495,185</td>
<td>468,491,999,592</td>
<td>266,760</td>
<td>19</td>
</tr>
</tbody>
</table>

Table 4: Basic statistics for the GoogleWeb1T subset.

<table>
<thead>
<tr>
<th>HASH</th>
<th>Total space in GBs</th>
<th>Bytes per gram</th>
<th>Lookup time [$\mu s$]</th>
</tr>
</thead>
<tbody>
<tr>
<td>KenLM</td>
<td>2.570</td>
<td>27.19</td>
<td>0.248</td>
</tr>
<tr>
<td>sxlm</td>
<td><strong>1.012</strong></td>
<td><strong>10.43</strong></td>
<td><strong>0.242</strong></td>
</tr>
<tr>
<td></td>
<td>(−61.64%)</td>
<td>(−2.42%)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TRIE</th>
<th>Total space in GBs</th>
<th>Bytes per gram</th>
<th>Lookup time [$\mu s$]</th>
</tr>
</thead>
<tbody>
<tr>
<td>KenLM</td>
<td>1.829</td>
<td>21.5</td>
<td>1.272</td>
</tr>
<tr>
<td>sxlm</td>
<td><strong>0.541</strong></td>
<td><strong>5.7</strong></td>
<td><strong>1.229</strong></td>
</tr>
<tr>
<td></td>
<td>(−73.34%)</td>
<td>(−3.38%)</td>
<td></td>
</tr>
</tbody>
</table>

Table 5: Bytes per grams and average lookup time in $\mu s$ for the GoogleWeb1T subset.
### tongrams - Preliminary results

<table>
<thead>
<tr>
<th>$n$</th>
<th>Number of $n$-grams</th>
<th>Maximum frequency count</th>
<th>Unique frequency counts</th>
<th>$[\lg]$ of unique frequency counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>24,359,472</td>
<td>468,491,999,592</td>
<td>246,588</td>
<td>18</td>
</tr>
<tr>
<td>2</td>
<td>5,089,239</td>
<td>155,178,163</td>
<td>44,822</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>52,635,338</td>
<td>102,329,901</td>
<td>71,690</td>
<td>17</td>
</tr>
<tr>
<td>4</td>
<td>11,149,161</td>
<td>6,401,274</td>
<td>21,127</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>8,261,975</td>
<td>958,556</td>
<td>12,171</td>
<td>14</td>
</tr>
<tr>
<td>Total</td>
<td>101,495,185</td>
<td>468,491,999,592</td>
<td>266,760</td>
<td>19</td>
</tr>
</tbody>
</table>

Table 4: Basic statistics for the GoogleWeb1T subset.

<table>
<thead>
<tr>
<th>HASH</th>
<th>Total space in GBs</th>
<th>Bytes per gram</th>
<th>Lookup time [μs]</th>
</tr>
</thead>
<tbody>
<tr>
<td>KenLM</td>
<td>2.570</td>
<td>27.19</td>
<td>X2.6</td>
</tr>
<tr>
<td>sxlm</td>
<td>1.012</td>
<td>10.43</td>
<td>0.242 (−61.64%)</td>
</tr>
<tr>
<td>Trie</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KenLM</td>
<td>1.829</td>
<td>21.5</td>
<td>X3.8</td>
</tr>
<tr>
<td>sxlm</td>
<td>0.541</td>
<td>5.7</td>
<td>1.229 (−3.38%)</td>
</tr>
</tbody>
</table>

Table 5: Bytes per grams and average lookup time in μs for the GoogleWeb1T subset.
Dynamic Inverted Indexes.
Dynamic Inverted Indexes.

Classic solution: use two indexes. One is big and **cold**; the other is small and **hot**. **Merge** them periodically.
Dynamic Inverted Indexes.

Classic solution: use two indexes. One is big and cold; the other is small and hot. Merge them periodically.
Compressed B-trees.
Compressed B-trees.

Problem: maintain a dictionary on disk.
Motivations: databases and file-systems.
Compressed B-trees.

Problem: maintain a dictionary on disk.
Motivations: databases and file-systems.

“Fancy indexing structures may be a luxury now, but they will be essential by the decade’s end.”
Compressed B-trees.

Problem: maintain a dictionary on disk.
Motivations: databases and file-systems.

“Fancy indexing structures may be a luxury now, but they will be essential by the decade’s end.”

Michael Bender  
Stony Brook University

Martin Farach-Colton  
Rutgers University

Bradley Kuszmaul  
MIT Laboratory for Computer Science
Fast Successor for IP-lookup.
Fast Successor for IP-lookup.

Successor search is what routers do for every incoming packet. *Hence*, the most run algorithm in the world.

Time *and* space efficiency is crucial.
Fast Successor for IP-lookup.

Successor search is what routers do for every incoming packet. Hence, the most run algorithm in the world.

Time and space efficiency is crucial.

We can directly jump to the position of the first address having the same $lg n$ bits as the searched pattern in $O(1)$ using the powerful search capabilities of Elias-Fano.
Fast Successor for IP-lookup.

Successor search is what routers do for every incoming packet. Hence, the most run algorithm in the world.

Time and space efficiency is crucial.

We can directly jump to the position of the first address having the same \( \lg n \) bits as the searched pattern in \( O(1) \) using the powerful search capabilities of Elias-Fano.
Fast Successor for IP-lookup.

Successor search is what routers do for every incoming packet. 

*Hence*, the most run algorithm in the world.

**Time and** space efficiency is crucial.

We can directly jump to the position of the first address having the same \( \lg n \) bits as the searched pattern in \( O(1) \) using the powerful search capabilities of Elias-Fano.
Fast Successor for IP-lookup.

Successor search is what routers do for every incoming packet. *Hence*, the most run algorithm in the world.

Time *and* space efficiency is crucial.

We can directly jump to the position of the first address having the same \( \lg n \) bits as the searched pattern in \( O(1) \) using the powerful search capabilities of Elias-Fano.

1110111010001000

Build an index on zeros.
Fast Successor for IP-lookup.

Successor search is what routers do for every incoming packet. Hence, the most run algorithm in the world.

Time and space efficiency is crucial.

We can directly jump to the position of the first address having the same $\lg n$ bits as the searched pattern in $O(1)$ using the powerful search capabilities of Elias-Fano.

1110111010001000

Build an index on zeros.

\[ p = \text{select}_0(h_x) - h_x \]
Fast Successor for IP-lookup.

Successor search is what routers do for every incoming packet. Hence, the most run algorithm in the world.

Time and space efficiency is crucial.

We can directly jump to the position of the first address having the same $\lg n$ bits as the searched pattern in $O(1)$ using the powerful search capabilities of Elias-Fano.

```
1110111010001000
```

Build an index on zeros.

\[
p = \text{select}_0(h_x) - h_x
\]

\[
\begin{array}{cccc}
0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 \\
0 & 0 & 1 & 1 \\
0 & 0 & 1 & 1 \\
0 & 0 & 1 & 1 \\
0 & 1 & 0 & 1 \\
1 & 0 & 1 & 0 \\
\end{array}
\]

\[x = 001100 \ (12)\]
(Some) Future Research Problems

Fast Successor for IP-lookup.

Successor search is what routers do for every incoming packet. *Hence*, the most run algorithm in the world.

Time *and* space efficiency is crucial.

We can directly jump to the position of the first address having the same $\lg n$ bits as the searched pattern in $O(1)$ using the powerful search capabilities of Elias-Fano.

\[
p = \text{select}_0(h_x) - h_x
\]

\[
\begin{array}{c}
1110111010001000 \\
\text{Build an index on zeros.}
\end{array}
\]

\[
x = 001100 \quad (12)
\]

\[
\begin{array}{c}
000011 \\
000100 \\
000111 \\
001101 \\
001110 \\
001111 \\
010101 \\
101011
\end{array}
\]
Fast Successor for IP-lookup.

Successor search is what routers do for every incoming packet. 

*Hence*, the most run algorithm in the world.

**Time and space efficiency is crucial.**

We can directly jump to the position of the first address having the same $lg n$ bits as the searched pattern in $O(1)$ using the powerful search capabilities of Elias-Fano.

\[
p = \text{select}_0(h_x) - h_x
\]
Thanks for your attention, time, patience!

Any questions?