Ordered Set Problems

Giulio Ermanno Pibiri

giulio.pibiri@di.unipi.it
http://pages.di.unipi.it/pibiri

07/06/2019
Given a set of \( n \) items and an order relation defined on them, we are asked to design a data structure that supports \textbf{Access}, \textbf{Contains}, \textbf{Successor}, \textbf{Predecessor} efficiently.
The Static Ordered Set Problem

Given a set of \( n \) items and an order relation defined on them, we are asked to design a data structure that supports Access, Contains, Successor, Predecessor efficiently.

Let us assume our items are integers drawn from some universe of size \( u \geq n \).
Given a set of $n$ items and an order relation defined on them, we are asked to design a data structure that supports **Access**, **Contains**, **Successor**, **Predecessor** efficiently.

Let us assume our items are integers drawn from some universe of size $u \geq n$.

If the integers are **not to be compressed**: use an **array**. Operations are made efficient by **binary search with loop unrolling** with cut-off to SSE/AVX (SIMD) **linear search** on small segments.

If the keys are **uniformly distributed**, **interpolation search** can help: **$O(\log \log n)$** time *with high probability*.
The Static Ordered Set Problem

Given a set of \( n \) items and an order relation defined on them, we are asked to design a data structure that supports Access, Contains, Successor, Predecessor efficiently.

Let us assume our items are integers drawn from some universe of size \( u \geq n \).

If the integers are not to be compressed:
use an array.
Operations are made efficient by binary search with loop unrolling with cut-off to SSE/AVX (SIMD) linear search on small segments.

If the keys are uniformly distributed, interpolation search can help: \( O(\log \log n) \) time with high probability.

Let us also assume \( n \) is so big that we must compress the set.
<table>
<thead>
<tr>
<th>Sorted integer sets are ubiquitous</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inverted indexes</td>
</tr>
<tr>
<td>Databases</td>
</tr>
<tr>
<td>E-Commerce</td>
</tr>
<tr>
<td>Graph compression</td>
</tr>
<tr>
<td>Semantic data</td>
</tr>
<tr>
<td>Geospatial data</td>
</tr>
</tbody>
</table>
The Static *Compressed* Ordered Set Problem

**Large** research corpora describing different **space/time** trade-offs.

- Elias’ Gamma and Delta
- Elias-Fano
- Variable-Byte Family
- Binary Interpolative Coding
- Simple Family
- PForDelta
- QMX
- Quasi-Succinct
- Partitioned Elias-Fano
- **Clustered Elias-Fano**
- Optimal Variable-Byte
- DINT

~1970

+ set intersection, union and decode

2019
The problem of (almost all) such representations is that Access, Contains, Predecessor/Successor are not natively supported, but we can just decode sequentially.
The problem of (almost all) such representations is that Access, Contains, Predecessor/Successor are not natively supported, but we can just decode sequentially.

Solution 1
Introduce some redundancy to accelerate queries: the so-called skip pointers.
The problem of (almost all) such representations is that Access, Contains, Predecessor/Successor are not natively supported, but we can just decode sequentially.

**Solution 1**
Introduce some redundancy to accelerate queries: the so-called skip pointers.
The problem of (almost all) such representations is that Access, Contains, Predecessor/Successor are not natively supported, but we can just decode sequentially.

**Solution 1**
Introduce some redundancy to accelerate queries: the so-called *skip pointers*.

Upperbounds: 14, 34, 49, 98
The problem of (almost all) such representations is that Access, Contains, Predecessor/Successor are *not natively supported*, but we can just decode sequentially.

**Solution 1**
Introduce some redundancy to accelerate queries: the so-called *skip pointers*.

Upperbounds: 14 34 49 98

3 9 10 14 23 24 25 34 38 42 44 49 50 65 71 98
The problem of (almost all) such representations is that Access, Contains, Predecessor/Successor are **not natively supported**, but we can just decode sequentially.

**Solution 1**
Introduce some redundancy to accelerate queries: the so-called *skip pointers*.

**Solution 2**
Redesign the data structure.
Partitioning by Universe

(a) partitioning by cardinality – PC

(b) partitioning by universe – PU
Partitioning by Universe

(a) partitioning by cardinality – PC

(b) partitioning by universe – PU

Does this remind you of something?
Partitioning by Universe

(a) partitioning by cardinality – PC

(b) partitioning by universe – PU

Does this remind you of something?

<table>
<thead>
<tr>
<th>input</th>
<th>3</th>
<th>4</th>
<th>7</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>21</th>
<th>25</th>
<th>36</th>
<th>38</th>
<th>54</th>
<th>62</th>
</tr>
</thead>
<tbody>
<tr>
<td>high</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>low</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>H</td>
<td>1110</td>
<td>1110</td>
<td>10</td>
<td>10</td>
<td>110</td>
<td>10</td>
<td>10</td>
<td>110</td>
<td>0</td>
<td>10</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>001-100-111</td>
<td>101-110-111</td>
<td>101</td>
<td>001</td>
<td>100-110</td>
<td>110</td>
<td>110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[Elia-Fano 1971-1975]
Partitioning by Universe

(a) partitioning by cardinality – PC

(b) partitioning by universe – PU

Does this remind you of something?

<table>
<thead>
<tr>
<th>input</th>
<th>3</th>
<th>4</th>
<th>7</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>21</th>
<th>25</th>
<th>36</th>
<th>38</th>
<th>54</th>
<th>62</th>
</tr>
</thead>
<tbody>
<tr>
<td>high</td>
<td>0 0 0</td>
<td>0 0 0</td>
<td>0 0 1 1</td>
<td>1 1</td>
<td>1 1</td>
<td>1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>low</td>
<td>0 0 0</td>
<td>0 0 0</td>
<td>1 1 0 0</td>
<td>0 0</td>
<td>1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 0 0</td>
<td>1 1 1</td>
<td>0 1 0 0</td>
<td>1 0</td>
<td>1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[van Emde Boas 1974-1975]

[Elías-Fano 1971-1975]
Partitioning by Universe

Assume a slice size of $2^3$
Partitioning by Universe

Assume a slice size of $2^3$

**Contains(x):**

\[ i = x >> 3 \]

search for \( x - (i << 3) \) in the i-th slice
Partitioning by Universe

Assume a slice size of $2^3$

Contains($x$): $x = 010101$

- $i = x >> 3$
- search for $x - (i << 3)$ in the i-th slice
Assume a slice size of $2^3$

Contains($x$): $x = 010101$

\[
i = x \gg 3 = 010101
\]

search for $x - (i << 3)$ in the i-th slice
Assume a slice size of $2^3$

Contains(x):

\[ i = x >> 3 \]
\[ x - (i << 3) \]

search for \( x - (i << 3) \) in the \( i \)-th slice

\[ x = 010101 \]
\[ i = 010101 \]
\[ x - 16 = 5 \]
Partitioning by Universe

Assume a slice size of $2^3$

Contains($x$):

- $x = 010101$
- $i = x \gg 3$
- $010101$
- search for $x - (i << 3)$ in the $i$-th slice

$x - 16 = 5$

Successor($x$):

- $i = x \gg 3$
- search for successor of $x - (i << 3)$ in the $i$-th slice
- (if $i$-th slice is empty or $x - (i << 3) > \text{max\_value}$ in $i$-th slice, then return first value on the right)
Partitioning by Universe

Assume a slice size of $2^3$

Contains($x$): $x = 010101$

i = $x >> 3$  

search for $x - (i << 3)$ in the i-th slice  

$x - 16 = 5$

Successor($x$):

i = $x >> 3$

search for successor of $x - (i << 3)$ in the i-th slice

(if i-th slice is empty or $x - (i << 3) > \text{max\_value}$ in i-th slice,
then return first value on the right)

Intersection between lists has to intersect only the slices in common between the lists.
Good old data structure for storing dense sets: x-th bit is set if integer x is in the set.
Bitmaps

Good old data structure for storing **dense sets**: x-th bit is set if integer x is in the set.

$$S = \{0, 1, 5, 7, 8, 10, 11, 14, 18, 21, 22, 28, 29, 30\}$$
Good old data structure for storing **dense sets**: x-th bit is set if integer x is in the set.

\[ S = \{0, 1, 5, 7, 8, 10, 11, 14, 18, 21, 22, 28, 29, 30\} \]

Contains: testing a bit
Successor/Predecessor: `__builtin_ctzll`
Select: `__builtin_ctzll`
Max: `__builtin_clzll`
Min: `__builtin_ctzll`
Decode: `__builtin_ctzll`
Insertion: setting a bit
Deletion: clearing a bit
Bitmaps

Good old data structure for storing **dense sets**: x-th bit is set if integer x is in the set.

\[ S = \{0,1,5,7,8,10,11,14,18,21,22,28,29,30\} \]

- **Contains**: testing a bit
- **Successor/Predecessor**: `__builtin_ctzll`
- **Select**: `__builtin_ctzll`
- **Max**: `__builtin_clzll`
- **Min**: `__builtin_ctzll`
- **Decode**: `__builtin_ctzll`
- **Insertion**: setting a bit
- **Deletion**: clearing a bit

Nothing is better than a bitmap for dense sets.
Assume $u = 2^{32}$

$2^{16}$ spans of $2^{16}$ values each
Assume $u = 2^{32}$

**Dense**: cardinality > 4096

**Sparse**: otherwise

Ensure at most 16 bits x key (excluding overhead)

Dense spans are represented with **bitmaps** of $2^{16}$ bits.

Sparse spans are represented with **sorted-arrays** of 16-bit integers.
Slicing

Assume $u = 2^{32}$

Dense slices are represented with bitmaps of $2^{16}$ or $2^8$ bits.
Sparse slices are represented with sorted-arrays of 8-bit integers.
Intersection between lists has to intersect **only the slices in common** between the lists.

- **Dense vs. Dense (Bitmap vs. Bitmap):**
  bitwise AND operations + (usually) automatic compiler vectorization

- **Dense vs. Sparse (Bitmap vs. Array):**
  Given the array $A$: check if bit $A[i]$ is set in the bitmap

- **Sparse vs. Sparse (Array vs. Array):**
  Vectorized processing using `_mm_cmpestrm` and `_mm_shuffle_epi8` SIMD instructions
Summing up

2 different paradigms

Partitioning by Cardinality (PC)

Partitioning by Universe (PU)
Experimental Comparison — Setting

Datasets

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Gov2</th>
<th>CW09</th>
<th>CCNews</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lists</td>
<td>35,636,425</td>
<td>92,094,694</td>
<td>43,844,574</td>
</tr>
<tr>
<td>Universe</td>
<td>24,622,347</td>
<td>50,131,015</td>
<td>43,530,315</td>
</tr>
<tr>
<td>Integers</td>
<td>5,742,630,292</td>
<td>15,857,983,641</td>
<td>20,150,335,440</td>
</tr>
</tbody>
</table>

Machine
Intel i7-4790K CPU @4GHz, 32 GiB RAM, Linux 4.13.0

Compiler
gcc 7.2.0 (with all optimizations: \texttt{-march=native} and \texttt{-03})

C++ sources
https://github.com/jermp/sindexes
https://github.com/jermp/dint
https://github.com/ot/ds2i
https://github.com/RoaringBitmap/CRoaring
## Experimental Comparison — Setting

### Datasets

<table>
<thead>
<tr>
<th>Density</th>
<th>Statistic</th>
<th>Gov2</th>
<th>CW09</th>
<th>CCNews</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^{-2}$</td>
<td>Lists</td>
<td>3513</td>
<td>5802</td>
<td>5930</td>
</tr>
<tr>
<td></td>
<td>Integers</td>
<td>4,347,653,438</td>
<td>11,676,154,022</td>
<td>16,677,342,102</td>
</tr>
<tr>
<td></td>
<td>%</td>
<td>76</td>
<td>74</td>
<td>83</td>
</tr>
<tr>
<td>$10^{-3}$</td>
<td>Lists</td>
<td>13,276</td>
<td>21,924</td>
<td>23,085</td>
</tr>
<tr>
<td></td>
<td>Integers</td>
<td>5,066,748,826</td>
<td>13,864,451,283</td>
<td>18,969,946,075</td>
</tr>
<tr>
<td></td>
<td>%</td>
<td>88</td>
<td>87</td>
<td>94</td>
</tr>
<tr>
<td>$10^{-4}$</td>
<td>Lists</td>
<td>85,893</td>
<td>99,227</td>
<td>79,954</td>
</tr>
<tr>
<td></td>
<td>Integers</td>
<td>5,390,038,277</td>
<td>14,805,194,135</td>
<td>19,681,352,639</td>
</tr>
<tr>
<td></td>
<td>%</td>
<td>94</td>
<td>93</td>
<td>98</td>
</tr>
</tbody>
</table>

### Configurations

<table>
<thead>
<tr>
<th>Method</th>
<th>Shorthand</th>
<th>Strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable-Byte</td>
<td>V</td>
<td>PC; fixed-sized partitions of 128 integers; byte-aligned</td>
</tr>
<tr>
<td>Elias-Fano</td>
<td>EF</td>
<td>PC; fixed-sized partitions of 128 integers; bit-aligned</td>
</tr>
<tr>
<td>Interpolative</td>
<td>BIC</td>
<td>PC; fixed-sized partitions of 128 integers; bit-aligned</td>
</tr>
<tr>
<td>Elias-Fano $\epsilon$-opt.</td>
<td>PEF</td>
<td>PC; variable-sized partitions; bit-aligned</td>
</tr>
<tr>
<td>Roaring without run opt.</td>
<td>R2</td>
<td>PU; single-span; 2 container types; byte-aligned</td>
</tr>
<tr>
<td>Roaring with run opt.</td>
<td>R3</td>
<td>PU; single-span; 3 container types; byte-aligned</td>
</tr>
<tr>
<td>Slicing</td>
<td>S</td>
<td>PU; multi-span; byte-aligned</td>
</tr>
</tbody>
</table>
Experimental Comparison — Compression Effectiveness

<table>
<thead>
<tr>
<th>Method</th>
<th>$d = 10^{-2}$</th>
<th></th>
<th>$d = 10^{-3}$</th>
<th></th>
<th>$d = 10^{-4}$</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Gov2</td>
<td>CW09</td>
<td>CCNews</td>
<td>Gov2</td>
<td>CW09</td>
<td>CCNews</td>
</tr>
<tr>
<td>V</td>
<td>8.60</td>
<td>8.72</td>
<td>8.66</td>
<td>8.72</td>
<td>9.00</td>
<td>9.08</td>
</tr>
<tr>
<td>EF</td>
<td>2.72</td>
<td>4.44</td>
<td>4.72</td>
<td>3.25</td>
<td>5.14</td>
<td>5.37</td>
</tr>
<tr>
<td>BIC</td>
<td>2.33</td>
<td>3.59</td>
<td>4.37</td>
<td>2.72</td>
<td>4.11</td>
<td>4.97</td>
</tr>
<tr>
<td>PEF</td>
<td>2.37</td>
<td>4.01</td>
<td>4.52</td>
<td>2.85</td>
<td>4.62</td>
<td>5.16</td>
</tr>
<tr>
<td>R2</td>
<td>6.00</td>
<td>8.88</td>
<td>8.25</td>
<td>7.03</td>
<td>9.99</td>
<td>9.21</td>
</tr>
<tr>
<td>S</td>
<td>3.23</td>
<td>5.44</td>
<td>5.98</td>
<td>3.91</td>
<td>6.39</td>
<td>7.18</td>
</tr>
</tbody>
</table>

PC-based methods, such as BIC and PEF, are best for space usage. Slicing (PU-based) stands in trade-off position.
Experimental Comparison — Sequential Decoding Time

### ns per integer

<table>
<thead>
<tr>
<th>Method</th>
<th>$d = 10^{-2}$</th>
<th>$d = 10^{-3}$</th>
<th>$d = 10^{-4}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Gov2</td>
<td>CW09</td>
<td>CCNews</td>
</tr>
<tr>
<td>V</td>
<td>0.51</td>
<td>0.61</td>
<td>0.53</td>
</tr>
<tr>
<td>EF</td>
<td>0.87</td>
<td>1.29</td>
<td>1.36</td>
</tr>
<tr>
<td>BIC</td>
<td>5.26</td>
<td>6.73</td>
<td>7.71</td>
</tr>
<tr>
<td>PEF</td>
<td>0.78</td>
<td>1.15</td>
<td>1.34</td>
</tr>
<tr>
<td>R2</td>
<td>0.53</td>
<td>0.72</td>
<td>0.68</td>
</tr>
<tr>
<td>R3</td>
<td>0.55</td>
<td>0.76</td>
<td>0.70</td>
</tr>
<tr>
<td>S</td>
<td>0.56</td>
<td>0.67</td>
<td>0.65</td>
</tr>
</tbody>
</table>

PU-based methods, are as fast as the fastest (vectorized) PC-based methods.
## Experimental Comparison — Intersection Time

### musec per intersection

<table>
<thead>
<tr>
<th>Method</th>
<th>$d = 10^{-2}$</th>
<th>$d = 10^{-3}$</th>
<th>$d = 10^{-4}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Gov2</td>
<td>CW09</td>
<td>CCNews</td>
</tr>
<tr>
<td>V</td>
<td>3648</td>
<td>6671</td>
<td>16954</td>
</tr>
<tr>
<td>EF</td>
<td>4652</td>
<td>8356</td>
<td>22818</td>
</tr>
<tr>
<td>BIC</td>
<td>12169</td>
<td>23608</td>
<td>58349</td>
</tr>
<tr>
<td>PEF</td>
<td>4380</td>
<td>7920</td>
<td>21710</td>
</tr>
<tr>
<td>R2</td>
<td>377</td>
<td>598</td>
<td>1138</td>
</tr>
<tr>
<td>R3</td>
<td>503</td>
<td>962</td>
<td>1338</td>
</tr>
<tr>
<td>S</td>
<td>507</td>
<td>1080</td>
<td>2370</td>
</tr>
</tbody>
</table>

**PU-based methods outperform PC-based methods.**
Experimental Comparison — Point Queries

### Access: ns per query

<table>
<thead>
<tr>
<th>Method</th>
<th>(d = 10^{-2})</th>
<th>(d = 10^{-3})</th>
<th>(d = 10^{-4})</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Gov2</td>
<td>CW09</td>
<td>CCNews</td>
</tr>
<tr>
<td>V</td>
<td>195</td>
<td>174</td>
<td>240</td>
</tr>
<tr>
<td>EF</td>
<td>118</td>
<td>122</td>
<td>173</td>
</tr>
<tr>
<td>BIC</td>
<td>890</td>
<td>835</td>
<td>1295</td>
</tr>
<tr>
<td>PEF</td>
<td>154</td>
<td>171</td>
<td>210</td>
</tr>
<tr>
<td>R2</td>
<td>475</td>
<td>545</td>
<td>610</td>
</tr>
<tr>
<td>R3</td>
<td>5604</td>
<td>18710</td>
<td>2852</td>
</tr>
<tr>
<td>S</td>
<td>153</td>
<td>170</td>
<td>244</td>
</tr>
</tbody>
</table>

### Successor: ns per query

<table>
<thead>
<tr>
<th>Method</th>
<th>(d = 10^{-2})</th>
<th>(d = 10^{-3})</th>
<th>(d = 10^{-4})</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Gov2</td>
<td>CW09</td>
<td>CCNews</td>
</tr>
<tr>
<td>V</td>
<td>252</td>
<td>226</td>
<td>308</td>
</tr>
<tr>
<td>EF</td>
<td>187</td>
<td>122</td>
<td>250</td>
</tr>
<tr>
<td>BIC</td>
<td>955</td>
<td>897</td>
<td>1385</td>
</tr>
<tr>
<td>PEF</td>
<td>167</td>
<td>182</td>
<td>229</td>
</tr>
<tr>
<td>R2</td>
<td>115</td>
<td>137</td>
<td>185</td>
</tr>
<tr>
<td>R3</td>
<td>105</td>
<td>138</td>
<td>188</td>
</tr>
<tr>
<td>S</td>
<td>145</td>
<td>174</td>
<td>225</td>
</tr>
</tbody>
</table>
Experimental Comparison — The Trade-Off Curve

Density = 1/1000
Future Research Directions

The Static Ordered Set Problem

The *Dynamic* Ordered Set Problem

+ insertions / deletions
The Static Ordered Set Problem

The Dynamic Ordered Set Problem

+ insertions / deletions

Theory
- Fusion Trees
- van Emde Boas Trees
- Exponential Search Trees
- Y-Fast Tries
- Dynamic Elias-Fano

Practice
- Red-Black Trees
- B-Trees

Memory management is the challenge.
The Dynamic Ordered Set Problem — On-going Work

Insert

\[ n = 1,000,000 \] 32-bit keys uniformly distributed

---

**Graph**

- **std::set**
- **s_set**

**Axes**
- \( y \)-axis: ns/update
- \( x \)-axis: density

**Legend**

- \( 2^{-10}, 2^{-9}, 2^{-8}, 2^{-7}, 2^{-6}, 2^{-5}, 2^{-4}, 2^{-3}, 2^{-2}, 2^{-1} \)

---

**Details**

- **Insert** operation
- **Performance** comparison between std::set and s_set
The Dynamic Ordered Set Problem — On-going Work

Successor

\[ n = 1,000,000 \] 32-bit keys uniformly distributed

<table>
<thead>
<tr>
<th>std::set</th>
<th>s_set</th>
</tr>
</thead>
<tbody>
<tr>
<td>430 ns/query</td>
<td>140 ns/query</td>
</tr>
</tbody>
</table>

Density vs. ns/query graph for different densities.
The Dynamic Ordered Set Problem — On-going Work

Heap usage

- std::set
- s_set

bytes [MiB]

0 5 10 15 20 25 30 35 40

time [sec]

0.0 0.2 0.4 0.6 0.8 1.0 1.2

bytes [KiB]

0 2000 4000 6000 8000 10000

0.0 0.2 0.4 0.6 0.8 1.0 1.2
Thanks for your attention, time, patience!

Any questions?