Bitvector Rank & Select: Primitives of succinct data structures
Thinking theoretically about data structure size

Assume that storing some data, in an information-theoretically optimal manner, requires $Z$ bits.

Representation of this data is:

*Implicit*: $Z + O(1)$ bits

  Only a constant size larger than the theoretical minimum

*Succinct*: $Z + o(Z)$ bits

  $Z$ bits, plus some term strictly smaller than $Z$ bits

*Compact*: $O(Z)$ bits

  On the order of $Z$ bits (grows linearly in $Z$)

The idea of succinct data structures was first introduced by Jacobson in his thesis “Succinct static data structures”*

In this thesis, among other things, he introduced succinct representations of trees and graphs that could be efficiently navigated.

As data sizes grow large, data structures that consume a lot of extra space become increasingly less feasible and so succinct data structures become increasingly important.

The rank and select operations become the basic building blocks of succinct data structures.

Slides for the following taken from:

credit to Simon J. Puglisi, University of Helsinki
Succinct Data Structures

• Succinct data structure
  = succinct representation of data + a succinct index

• (usually static)

• High-level goal: reduce space so the data structure might fit in RAM and therefore be faster to use

• Examples
  – Sets
  – Trees, graphs
  – Strings
  – Permutations, functions

Credit to Simon J. Puglisi, University of Helsinki
Succinct Representation

• A representation of data whose size (roughly) matches the information-theoretic lower bound

• If the input is taken from L distinct possible inputs, then its information-theoretic lower bound is $\lceil \log L \rceil$ bits
  – To be considered succinct a data structure must use: $\lceil \log L \rceil + o(\log L)$ bits

• Example: a lower bound for a set $S$, subset of $\{1,2,\ldots,n\}$
  – $\log(2^n) = n$ bits
  – $n = 3$ we have 8 distinct sets… so d.s. will need at least 3 bits
    $\emptyset$
    $\{1\}$  $\{2\}$  $\{3\}$
    $\{1,2\}$  $\{1,3\}$  $\{2,3\}$
    $\{1,2,3\}$

credit to Simon J. Puglisi, University of Helsinki
Succinct Index

- Auxiliary data structure to support queries on the succinct representation

- Size: $o(\log L)$ bits

- The index should allow queries/operations on the succinct representation in (almost) the same time complexity as using a conventional data structure
  - This is the aim anyway

- Computational model is the word RAM
  - Assume word length $w = \log \log L$
  - (this is the same pointer size as conventional data structures)
  - read/write $w$ bits of memory in $O(1)$ time
  - arithmetic/logical operations on $w$ bit numbers take $O(1)$ time
  - $+, -, *, /, \log, &, |, \!, >>, <<$

credit to Simon J. Puglisi, University of Helsinki
The ability to answer rank and select queries over bit vectors (binary strings, bit arrays) is essential for implementing succinct data structures.

Given a binary string B[1..n]

- $\text{rank}_B(i)$ returns the number of 1 bits in $B[1..i]$
- $\text{select}_B(i)$ returns the position of the $i^{\text{th}}$ 1 bit in $B$

credit to Simon J. Puglisi, University of Helsinki
Naïve rank

- To answer \( \text{rank}(i) \) scan \( B[1..i] \) and count 1-bits

- Simple but slow
  - \( O(i) \) time = \( O(n) \) time in the worst case

- How can we do better?
  - After all, what are we?

credit to Simon J. Puglisi, University of Helsinki
(Slightly) Less naïve rank

- Store an table \( A[1..n] \), containing the rank answers

\[
\begin{array}{cccccccccccccccc}
1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\
1 & 1 & 1 & 2 & 2 & 3 & 4 & 5 & 5 & 6 & 6 & 6 & 7 & 7 & 8 & 8 \\
\end{array}
\]

- \( A[i] = \text{rank}(i) \)
  - Now \( \text{rank}(i) \) takes constant time - just an array lookup!

- **Drawback:**
  - \( A \) requires \( n \log n \) bits - \( \log n \) times the size of \( B \) - not succinct!
  - We’d like a solution with \( O(1) \) queries and \( o(n) \) extra space...

credit to Simon J. Puglisi, University of Helsinki
We want $O(1)$ queries with $o(n)$ extra bits...

- General approach will be to precompute some tables

- Each table stores part of the answer to every query
  - For any given query, we can extract needed parts in $O(1)$ time
  - The total size of the tables is $o(n)$ bits

<table>
<thead>
<tr>
<th>B</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
</table>

- Premise:
  - Can read $O(\log n)$ bits into an integer in range 1..$n$ in $O(1)$ time
  - However, to inspect each of those bits take $O(\log n)$ time

credit to Simon J. Puglisi, University of Helsinki
Tables: Superblocks

- Divide B into superblocks of size \( s = \log^2 n/2 \) = \( 4 \times 4/2 = 8 \)

- Build a small table \( R_s \) containing ranks for only some positions

\[
\begin{array}{cccccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\
B & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\
\end{array}
\]

- Store in \( R_s[j] = \text{rank}_B(j \times s) \), for all \( 0 \leq j < n/s \)

credit to Simon J. Puglisi, University of Helsinki


<table>
<thead>
<tr>
<th>B</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rs</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>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Rb</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- Divide each superblock into blocks of size \( b = \log n/2 = 2 \)

- Build a table \( R_b \) which contains the rank from the start of each block to the start of its superblock

- Store \( R_b[k/b] = \text{rank}_B(k*s) - \text{rank}_B(j*s) \), for all \( 0 \leq k < n/b \)

credit to Simon J. Puglisi, University of Helsinki
Intermission

- What we have so far (tables $R_s$ and $R_b$) almost gets us the answer we’re after

\[
\begin{array}{cccccccccccccccc}
& 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\
B & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\
R_s & 0 & & & & & & & & & & & & & & & \\
R_b & 0 & 1 & 2 & 3 & 0 & 1 & 1 & 2 & 3 & & & & & & & & \\
\end{array}
\]

- \( \text{rank}_B(i) \approx R_s[i/s] + R_b[i/b] \)
  - Just need to answer in-block queries in \( O(1) \) time

credit to Simon J. Puglisi, University of Helsinki
Tables: Resolving in-block queries

- **Solution? Use another table!**

- **Blocks have size** \( b = \log_2 n/2 \)
  - There are \( 2^b \) such blocks possible
  - In each block there are \( b \) possible rank queries
  - Each answer (relative to the block) is in the range 1..\( b \)

<table>
<thead>
<tr>
<th>Type</th>
<th>0</th>
<th>1</th>
<th>rank(0)</th>
<th>rank(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

*Credit to Simon J. Puglisi, University of Helsinki*
Final Data Structure

\[
\begin{array}{cccccccccccccc}
\text{B} & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\
\hline
0 & & & & & & & & & & & & & & & & \\
1 & & & & & & & & & & & & & & & & \\
\end{array}
\]

\[
\begin{array}{cc}
\text{R}_s & 0 & 5 \\
\hline
0 & 1 & 2 & 3 & 0 & 1 & 1 & 2 \\
\end{array}
\]

\[
\begin{array}{cccc}
\text{Type} & 0 & 1 & \text{rank(0)} & \text{rank(1)} \\
0 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 1 \\
2 & 1 & 0 & 1 & 1 \\
3 & 1 & 1 & 1 & 2 \\
\end{array}
\]

credit to Simon J. Puglisi, University of Helsinki
Size of table for within-block queries

- Blocks have size \( b = \log_2 n/2 \)
  - There are \( 2^b \) such blocks possible
  - In each block there are \( b \) possible rank queries
  - Each answer (relative to the block) is in the range 1..\( b \)

- Therefore size of \( R_p \), the in-block data structure is
  \[ 2^b \cdot b \cdot \log b = n^{1/2} \cdot \log n \cdot \log \log n/2 \text{ bits} = o(n) \text{ bits} \]

credit to Simon J. Puglisi, University of Helsinki
Summing up sizes...

- The size of $R_s$, the superblock data structure is
  - $2n/\log^2 n$ superblocks, each of size $\log n$ bits
  - $(n/\log^2 n)*\log n = 2n/\log n$ bits $= o(n)$ bits

- The size of $R_b$, the block data structure is
  - $2n/\log n$ blocks, each of size $\log\log n$ bits
  - $2n\log\log n/\log n$ bits $= o(n)$ bits

- $R_s + R_b + R_p = o(n)$ extra bits for $O(1)$ time rank queries
  - It is possible to construct this data structure in $O(n)$ time

Credit to Simon J. Puglisi, University of Helsinki
Variations

- Just store $R_s$ + use manual counting within superbblocks
  - Saves space for $R_b$ and $R_p$, takes time $O(\log^2 n)$ per query

- Store $R_s$ and $R_b$ + use manual counting within blocks
  - Saves only space for $R_p$, takes time $O(\log n)$ per query

- Use different superbblock & block sizes
  - No more theoretical guarantees, but...
  - Perhaps faster in practice: blocks that are multiples of word sizes (32-bits) can be faster to handle

credit to Simon J. Puglisi, University of Helsinki
Summary of rank

• Rank index takes $O(n \log \log n / \log n)$ = $o(n)$ bits so we use $n + o(n)$ overall and can answer queries in $O(1)$ time

• While it is sublinear, we’d still like the $o(n)$ term to be small
  – Best is by Patrascu: $O(n / \log^k n)$ bits, $O(k)$ time queries

• Dynamic solutions exist
  – Queries no longer constant: $O(\log n / \log \log n)$ time (Raman et al.)
We can use our solution to rank to get a (fairly) efficient solution to select(i), with this observation:

If rank(n/2) > i, then the i\textsuperscript{th} 1-bit is in B[1..n/2]

- Otherwise it is in B[n/2+1..n]

```plaintext
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

select\(_B\)(3)

- rank\(_B\)(8) = 5
- rank\(_B\)(4) = 2
Relationship to select(i)

- Applying this idea recursively to arrive at select(i)
  - $O(\log_2 n)$ time, $o(n)$ space

- $O(1)$ time, $o(n)$ space solutions for select also exist
  - Slightly more complicated than $O(1)$ rank
  - (Munro and Clark)

- Similar variations as we discussed with rank (trading space for query time) are also possible


Rank and select: Another lesson learned

Szymon Grabowski*, Marcin Raniszewski

Lodz University of Technology, Institute of Applied Computer Science, Al. Politechniki 11, Łódź 90–924, Poland

credit to Simon J. Puglisi, University of Helsinki
These rank & select operations work over a binary alphabet — can be extended

Introduces the idea of the wavelet tree, a versatile index that can be extended to arbitrary alphabets. We’ll discuss the simplest of variants according to the exposition of:

Wavelet Trees for All *

Gonzalo Navarro

Dept. of Computer Science, University of Chile. gnavarro@dcc.uchile.cl
S[1,n] = s_1s_2 \ldots s_n is a sequence of symbols where s_i in \Sigma

\Sigma = [1 \ldots \sigma] is an alphabet of symbols

Representing S requires n * \lceil \log \sigma \rceil = n * \log \sigma + O(n) bits.

Wavelet tree: balanced binary tree with \sigma nodes, where each subtree is also a wavelet tree (i.e. it is recursive)
**Structure.** A wavelet tree [54] for sequence $S[1, n]$ over alphabet $[1..\sigma]$ can be described recursively, over a sub-alphabet range $[a..b] \subseteq [1..\sigma]$. A wavelet tree over alphabet $[a..b]$ is a binary balanced tree with $b - a + 1$ leaves. If $a = b$, the tree is just a leaf labeled $a$. Else it has an internal root node, $v_{\text{root}}$, that represents $S[1, n]$. This root stores a bitmap $B_{v_{\text{root}}}[1, n]$ defined as follows: if $S[i] \leq (a + b)/2$ then $B_{v_{\text{root}}}[i] = 0$, else $B_{v_{\text{root}}}[i] = 1$. We define $S_0[1, n_0]$ as the subsequence of $S[1, n]$ formed by the symbols $c \leq (a + b)/2$, and $S_1[1, n_1]$ as the subsequence of $S[1, n]$ formed by the symbols $c > (a + b)/2$. Then, the left child of $v_{\text{root}}$ is a wavelet tree for $S_0[1, n_0]$ over alphabet $[a..\lfloor (a+b)/2 \rfloor]$ and the right child of $v_{\text{root}}$ is a wavelet tree for $S_1[1, n_1]$ over alphabet $[1 + \lfloor (a+b)/2 \rfloor..b]$. 
Fig. 1. A wavelet tree on string $S = "\text{alabar a la alabarda}"$. We draw the spaces as underscores. The subsequences of $S$ and the subsets of $\Sigma$ labeling the edges are drawn for illustration purposes; the tree stores only the topology and the bitmaps.
Example: Rank

Consider asking for the rank of this “a”
Example: Rank

Consider asking for the rank of this “a”

how many _,a,b occur up to (and including) this one? Count 0’s at this level — 8

```
111010101111
aaa_a_a_aaaa
00100111
```

```
lrllrd
010010
```

```
alabar_a_la_alabarda
0100010001000100110
```

```
aa _a_a_aabaa
0100000000100
```

```
___  aaaaaaa
```

```
___  a
```

```
_  b
```

```
d, l, r
```

```
l  d, l
```

```
d  l
```

```
d, l
```

```
r
```

```
```
Example: Rank

Consider asking for the rank of this “a”

how many _ , a, b occur up to (and including) this one? Count 0’s at this level — 8

So that maps to 8th character at this level
Example: Rank

Consider asking for the rank of this “a”

So that maps to 8th character at this level

Count 0’s at this level — 7
Example: Rank

Consider asking for the rank of this “a”

So that maps to 7th character at this level
Example: Rank

Consider asking for the rank of this “a”

So that maps to 5th character at this level

Count 1’s at this level — 5
Example: Rank

Consider asking for the rank of this “a”

This is the 5th ‘a’

---

$\text{Rank}_a(S, 11) = 5$

If we are 1-indexing
Example: Rank

This procedure turns rank for any character in the alphabet into $\lg \sigma$ rank calculations over bitvectors.

We can answer rank queries for an arbitrary character in $\lg \sigma \cdot O(1) = O(\lg \sigma)$ time. For small, constant alphabets, through the magic of Big-O, this is constant time. :)

---

### Tree Representation

```
  alabar_a_la_alabarda
    01000100010001000110
     __a,b__
       / \      /   \       /     \
      aaba_a_a_aabaa lrrlldrd
        00100000000100 010010
         __a   \    ___  d,l,r
          /     /     /    / \
         a     b     bb    rr
           \     \    \    \_____      ___
            \    \    \    \   d           d
             \    \    \    \  l          l
              \    \    \    \a          _a
               \    \    \    \__    b
                \    \    \    \__    \_a
                 \    \    \    \__    a
                  \    \    \    ___    ___
                   ___ aaaaaaaaaa ___ llll
                       ___ llll
```

---
Example: Select

Select the 3rd “l” (at what index does it occur?)
Example: Select

Select the 3rd “l” (at what index does it occur?)

Where does this “l” go at this level? It’s the 3rd 1

Here, we start at the bottom of the tree and work up.
Example: Select

Select the 3rd “l” (at what index does it occur?)

Where does this “l” go at this level? It’s the 3rd 0

Here, we start at the bottom of the tree and work up.
Example: Select

Select the 3rd “l” (at what index does it occur?)

Where does this “l” go at this level? It’s the 4th 1

Here, we start at the bottom of the tree and work up.
Example: Select

This procedure turns select for any character in the alphabet into $\lg \sigma$ select calculations over bitvectors.

We can answer select queries for an arbitrary character in $\lg \sigma \cdot O(1) = O(\lg \sigma)$ time. For small, constant alphabets, through the magic of Big-O, this is constant time. :)

The diagram illustrates the process with a binary tree structure, where each node represents a character or a substring, and the leaves represent the bitvectors for selecting characters.
Succinct Data Structures

We have only scratched the surface on what is possible with rank & select and succinct data structures in general.

However, we’ll assume familiarity with rank and select moving forward as we talk about data structures in Comp Bio that use them.

Gonzalo Navarro alone publishes 14-24 papers / year in this field :):

2019 (14+10)  2018 (14)
2017 (21)      2016 (19)

A google search on Gonzalo, and succinct data structures will send you down a wonderful rabbit-hole; I recommend you try it!
Some practical advice

Succinct data structure papers tend to be quite theoretical (go figure!).

Luckily, there is a go-to library for implementation of these ideas.
Some practical advice

Succinct data structure papers tend to be quite theoretical (go figure!).

Luckily, there is a *go-to* library for implementation of these ideas.

**Succinct Data Structure Library 2.0**

Provides a modern, modular C++ implementation of many different succinct data structures.
Some practical advice

Create an FM-index over some text?

```cpp
#include <sds/suffix_arrays.hpp>
#include <string>
#include <iostream>

string index_file = string(argv[1]) + index_suffix;
csa_wt<wt_huff<rrr_vector<127>>, 512, 1024> fm_index;

if (!load_from_file(fm_index, index_file)) {
    ifstream in(argv[1]);
    if (!in) {
        return 1;
    }
    cout << "No index " << index_file << " located. Building index now." << endl;
    construct(fm_index, argv[1], 1); // generate index
    store_to_file(fm_index, index_file); // save it
}
Perform rank queries over a bit vector?

```cpp
#include <iostream>
#include <sdsl/bit_vectors.hpp>

#include <iostream> using namespace std;
#include <sdsl/bit_vectors.hpp>

int main()
{
    using namespace s;
    using namespace sds;

    int main()
    {
        bit_vector b = {0,1,0,1,1,0,1,0,1,1};
        size_t zeros = rank_support_v<0>(&b)(b.size());
        bit_vector::select_0_type b_sel(&b);

        for (size_t i=1; i <= zeros; ++i) {
            cout << b_sel(i) << " ";
        }
        for (size_t i=0; i<=b.size(); i+= b.size()/4)
        {
            cout << "(" << i << " , " << b_rank(i) << ") " ;
            cout << endl;
        }
    }
}
Some practical advice

How about select_0

```cpp
#include <iostream>
#include <sds/sl/bit_vectors.hpp>

using namespace std;
using namespace sds;

int main()
{
    bit_vector b = {0,1,0,1,1,1,0,0,0,1,1};
    size_t zeros = rank_support_v<0>(&b)(b.size());
    bit_vector::select_0_type b_sel(&b);

    for (size_t i=1; i <= zeros; ++i) {
        cout << b_sel(i) << " ";
    }
    cout << endl;
}
```
Some practical advice
You get the idea! An incredibly powerful library, at your fingertips.

### sdl Cheat Sheet

#### Data structures

The library code is in the `sdl` namespace. Either import the namespace in your program (using `namespace sdl;`) or qualify all identifiers by a `sdl::*`. Each section corresponds to a header file. The file is hyperlinked as part of the section heading.

We have two types of data structures in `sdl`. Self-contained and support structures. A support object can extend a self-contained object (e.g. add functionality), but requires access to a. Support structures contain the subsupport in their class names.

#### Integer Vectors (IV)

The core of the library is the class `int_vector<w>`. Parameter w corresponds to the fixed length of each element in bits. For `w = 8, 16, 32, 64`, the length is fixed during compile-time and the vectors correspond to `std::vector<int_t<w>>`. If `w = 0` (default) the length can be set during runtime. Constructor: `int_vector<w>(n, x, c, d)`, with `n` equals size, `x` default integer value, `c` width of integer (has no effect for `w > 0`).

**Public methods:** `operator[]()`, `size<w>()`, `begin<w>()`, `end<w>()`.

#### Manipulating `int_vector<w>` v

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>v[w]=x</td>
<td>Set entry v[w] to x.</td>
</tr>
<tr>
<td>v.width()</td>
<td>Set width to t, if t &gt; 0.</td>
</tr>
<tr>
<td>v.realize()</td>
<td>Realize v to n elements.</td>
</tr>
</tbody>
</table>

**Useful methods in namespace `sdl::util`:**

- `set_to_zero(v, k)` Set `v[0:k]` to 0 for each i.
- `set_to_one(v, k)` Set `v[0:k]` to 1 for each i.
- `set_random_bits(v, s)` Set elements to random bits.
- `mod(v, m)` Set `v[0,m)` mod `m` for each i.
- `bit_compress(v)` Gets `x` max width, `v` and `t = log2(x+1)` packs the entries in `t` bits.\[ \text{Compressed Integer Vectors (CIV)} \]

For a vector v, `vc_vector` stores the self-defining coded delta `(v[i+1] - v[i])`. Fast random access is achieved by sampling values v at rate t_dens. Available coders are `coder::elias_delta`, `coder::elias_gamma`, and `coder::fibonacci`.

Samples `vc_vector` stores each v[1] as self-defining codeword. Samples `dc_vector` stores each value z the least (t, b − 1) significant bits plus a bit which is set if z > 2^b − 1. In the latter case, the process is repeated with x = x/2^b − 1.

#### Bitvectors (BV)

Representations for a bitvector of length n with m set bits. Class `bit_vector` plain bitvector.

- `bit_vector()` Interleaved bitvector.
- `interleaved_bitvector()` Interleaved bitvector.
- `sparse_bitvector()` Sparse bitvector.
- `hybrid_bitvector()` Hybrid bitvector.

Every bitvector equals `int_vector<w>` and is therefore dynamic. **Public methods:** `operator[](i)`, `size<w>()`, `begin<w>()`, `end<w>()`.

#### Rank Supports (RS)

RIs add rank functionality to BV. Methods rank() and `operator[](i)` return the number of set bits in `i` in the prefix `[0:i]` of the supported BV for `i ∈ [0:n]`.

**Public methods:** `operator[](i)`, `size<w>()`, `begin<w>()`, `end<w>()`.

#### Select Supports (SLS)

SLSs add select functionality to BV. Let m be the number of set bits in BV. Methods select() and `operator[](i)` return the position of the i-th set bit in BV for `i ∈ [0:m]`.

**Public methods:** `operator[](i)`.

#### Compressed Integers (CI)

Compressed integer vectors use CIs or WIs to represent the suffix arrays (SA), its inverse (ISA), BWT, Ψ, and Λ. CSAs can be built over byte and integer alphabets.

**Public methods:** `operator[](i)`, `size<w>()`, `begin<w>()`, `end<w>()`.

#### Suffix Arrays (CSA=IV+WT)

Compressed suffix arrays use CIs or WIs to represent the suffix arrays (SA), its inverse (ISA), BWT, Ψ, and Λ. CSAs can be built over byte and integer alphabets.

**Public methods:** `operator[](i)`, `size<w>()`, `begin<w>()`, `end<w>()`.

#### Longest Common Prefix (LCP) Arrays

Wavetrees represent sequences over byte or integer alphabets of size σ and consist of a tree of BVs. Rank and select on the sequences is reduced to rank and select on BVs, and the runtime is multiplied by a factor in [log₂ log₉].

**Public methods:** `operator[](i)`.

#### Balanced Parentheses Supports (BPS)

We represent a sequence of parentheses as a bit vector. An opening/closing parenthesis corresponds to 1/0.

**Public methods:** `operator[](i)`.
Some practical advice

You get the idea! An incredibly powerful library, at your fingertips.

Suffix Trees (CST:=CSA+LCP+BPS)

A CST can be parametrized by any combination of CSA, LCP, and BPS. The operation of each part can still be accessed through member variables. The additional operations are listed below. CSTs can be built for byte or integer alphabets.

Class Description
cst_sada
Represents a node as position in BPS. Navigational
operations are fast (they are directly transcribed
in BPS operations on the DFS-BPS). Space:
4+n(1+C[CSA]+LCP) bits.
cst_sct3
Represents nodes as intervals. Fast construction,
but slower navigational operations. Space: 3n +
2o(n)+C[CSA]+LCP) bits.

Public type: node_type. In the following let v and w be nodes
and i, d, l, b, r be integers.

Public methods: size(o), nodes(c), root(o), begin(l), end(c),
begin_bottom_up(s), end_bottom_up, size(v), is_leaf(v),
degree(v), depth(v), node_depth(v), edge(v, d), lbd(v),
rb(v), id(v), inv_id(v), sm(v), select_leaf(o), node(lb,
rb), parent(v), sibling(v), lca(l, v, w), select_child(v, l),
child(v, l), children(v, l), sl(v), vl(v, w, c),
leftmost_leaf(v), rightmost_leaf(v)

Public members: csa, lcp.
The traversal example shows how to use the DFS-iterator.

Range Min/Max Query (RMQ)

A RMQ rmq can be used to determine the position of the minimum value1 in an arbitrary subrange [i, j] of an
preprocessed vector v. Operator operator(, j) returns
z = min {v[r] | r ∈ [i, j] ∧ v[r] ≤ v[k] ∀ k ∈ [i, j]}

Class Space
rmq_support_sparse_table n log2 n C(1)
rmq_suc_each
4n+2o(n) C(1)
rmq_suc_each
2n+3o(n) C(1)

Constructing data structures

Let o be a W.T., CSA, or CST-object. Object o is built with
construct(r, file, num_bytes=0) from a sequence stored in
file. File is interpreted dependent on the value of num_bytes:

Value File interpreted as
num_bytes=0 serialized int_vector
num_bytes=1 byte sequence of length util::file_size(file)
num_bytes=2 16-bit word sequence.
num_bytes=4 32-bit word sequence.
num_bytes=8 64-bit word sequence.
num_bytes=4 Parse decimal numbers.

Note: construct writes reads data to/from disk during
construction. Accessing disk for small instances is a
considerable overhead. construct_info(num_data_bytes=0)
will build o using only main memory. Have a look at this
beforehand tool for an example.

Configuring construction

The locations and names of the intermediate files can be
configured by a cache_object config. It is constructed by
cache_config(sdl, tmp_dir, id, map) where sdl is a boolean
variable which specifies if the intermediate files should be
deleted after construction, tmp_dir is a path to the directory
where the intermediate files should be stored, id is used as
part of the file names, and map contains a mapping of keys
(e.g. conf::KEY_BT, conf::KEY_SA, ...) to file paths.
The cache_config parameter extends the construction method
to construct(r, file, config, num_bytes).

The following methods (key is a key string, config represent
a cache_config object, and o a std object) should be handy
in customized construction processes:
cache_file_name(key, config)
cache_file_exists(key, config)
register_cache_file(key, config)
load_from_cache(key, config)
store_to_cache(key, config).

Resource requirements

Memory: The memory peak of CSA and CST construction
occurs during the SA construction, which is 5 times the tests
size for byte-arrays and inputs < 2 GB (see the figure
below for a 200 MB text) and 0 times for larger inputs. For
integer alphabets the construction takes about twice the space
of the resulting output.

Time: A CST construction processes at about 2 MB/s. The
Figure below shows the resource consumption during the
construction of a std::CST for 250 MB English text.
For a detailed description of the phases click on the figure.

Utility methods

More useful methods in the std::util namespace:

pid() Id of current process.
id() Get unique id inside the process.
basename(p) Get filename part of a path p.
dirname(p) Get directory part of a path p.
dimangle(e) Demangles output of typeid().name().
dimangle2(e) Simplifies output of dimangle. E.g. removes
sdsl::prefixes, ...
to_string(o) Transform object o to a string.
assign(o1, o2) Assign o1 to o2, or swap o1 and o2 if the objects
are of the same type.
clear(o) Set o to the empty object.

Measuring and Visualizing Space

size_in_bytes(o) returns the space used by an std object o.
Call writestructure_json_format(o, out) to get a detailed
space breakdown written in JSON format to stream out.
<JSON_FORMAT> will write a HTML page (like this),
which includes an interactive SVG-figure.

Methods on words

Class bite contains various fast methods on a 64-bit word z.
Here the most important ones.

Method Description
bite::cmp(x) Number of set bits in x.
bite::set(x, i) Position of i-th set bit, i ∈ [0, cmp(x)−1].
bite::low(x) Position of least significant set bit.
bite::hi(x) Position of most significant set bit.
Notes: Positions in x start at 0. lo and hi return 0 for x = 0.

Tests

A make test call in the test directory, downloads test inputs,
compiles tests, and executes them.

Benchmarks

Directory contains configurable benchmarks for
various data structures, like WTs, CSA’s/FM-indexes
(measuring time and space for operations count, locate, and
concatenate).

Debugging

You get the gdb command pr <int_vector> <int2> <int2>,
which displays the elements of an int_vector in the range
[int1, int2] by appending the field std::gdb to your
.gdbinit.

© Simon Gog
Cheat sheet template provided by Winston Chang
http://www.stdlib.org/~winston/latex/

Notes
1 select::type not defined for sd_vector.
2 It is also possible to rank 0 or the patterns 10 and 01.
3 It is also possible to select 0 or the patterns 10 and 01.
4 For PHS the bits are counted in the prefix [0,i].
5 Or maximum value; can be set by a template parameter.

This diagram was generated using the sample program
memory-visualization.cpp.

Reading and writing data

Importing data into std::structures

load_vector_from_file(v, file, num_bytes)
Load file into an int_vector v. Interpretation of file
depends on num_bytes, see method construct.

Store std::structures

Use store_to_file(o, file) to store an std object o to file.
Object o can also be serialized into a std::iostream object
by the call o.serialize().

Load std::structures

Use load_from_file(o, file) to load an std object o, which
is stored in file. Call o.load() reads o from
std::iostream object.