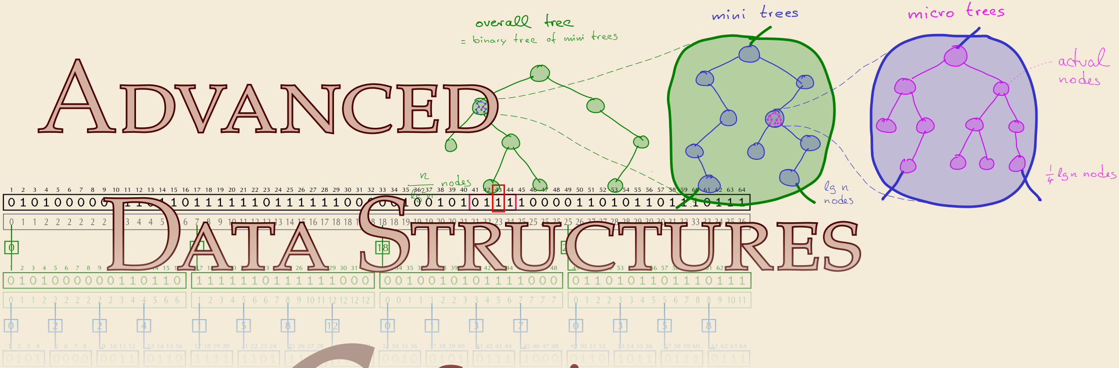


ADVANCED DATA STRUCTURES

6

Succinct Data Structures

Advanced Data Structures · Summer 2026

Prof. Dr. Sebastian Wild

CS650 (Summer 2026)
Philipps-Universität Marburg
version `2026-07-06 16:07`

Outline

6 Succinct Data Structures

6.1 A Motivating Example

Computing over compressed data

- traditionally:
  - compression for **archiving** data, minimize size of representation
  - computation/analysis: minimize time; use extra data structures
  - ⇝ always decompress data first!
  - reaches limits of fast memory for large datasets

Computing over compressed data

- traditionally:
  - compression for **archiving** data, minimize size of representation
  - computation/analysis: minimize time; use extra data structures
  - ⇝ always decompress data first!
  - reaches limits of fast memory for large datasets

- Approach in space-efficient data structures:
  - Represent data in compressed form
  - Augment with **small** index data structures to enable fast queries **directly on compressed representation**
  - *succinct* = $(1 + o(1)) \cdot$ information-theoretic lower bound

$n = |U|$ $\lg n$ bits needed to distinguish objects in U

Computing over compressed data

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$T[0..n) =$	a	b	a	n	a	n	a	a	n	d	a	n	a	p	p	l	e

Computing over compressed data

$T[0..n) =$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
a	b	a	n	a	n	a	a	n	d	a	n	a	p	p	l	e

Huffman Code

`a` ↦ `0`

`b` ↦ `11100`

`d` ↦ `11101`

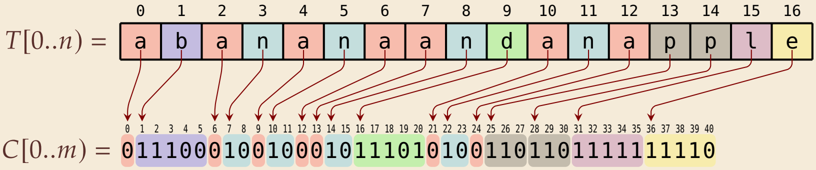
`e` ↦ `11110`

`l` ↦ `11111`

`n` ↦ `10`

`p` ↦ `110`

Computing over compressed data



$T[0..n) =$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
a	b	a	n	a	n	a	a	n	d	a	n	a	p	p	l	e

$C[0..m) =$ `0 11100 0 10 0 10 0 0 10 11101 0 10 0 110 110 11111 11110`

Huffman Code

a ↦ 0
b ↦ 11100
d ↦ 11101
e ↦ 11110
l ↦ 11111
n ↦ 10
p ↦ 110

Computing over compressed data

$T[0..n) =$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
a	b	a	n	a	n	a	a	n	d	a	n	a	p	p	l	e

$C[0..m) =$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
0	1	1	1	0	0	0	1	0	0	1	0	0	0	1	0	1	1	1	0	1	0	1	0	0	1	1	0	1	1	0	1	1	1	1	1	1	1	1	1	0

a b

- Can decode from beginning ⇝ fine for storage / transmission

Huffman Code

`a` ↦ `0`
`b` ↦ `11100`
`d` ↦ `11101`
`e` ↦ `11110`
`l` ↦ `11111`
`n` ↦ `10`
`p` ↦ `110`

Computing over compressed data

$T[0..n) =$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
a	b	a	n	a	n	a	a	n	d	a	n	a	p	p	l	e

Huffman Code

- a ↦ 0
- b ↦ 11100
- d ↦ 11101
- e ↦ 11110
- l ↦ 11111
- n ↦ 10
- p ↦ 110

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40

$C[0..m) =$ `01110001001000101110101001101101111111110`

- Can decode from beginning ⇝ fine for storage / transmission

But how to read $T[9]$ without full decoding? How to know where its codeword starts?

Computing over compressed data

$T[0..n) =$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
a	b	a	n	a	n	a	a	n	d	a	n	a	p	p	l	e

$C[0..m) =$ `01110001001000101110101001101101111111110`

Huffman Code

a ↦ 0
b ↦ 11100
d ↦ 11101
e ↦ 11110
l ↦ 11111
n ↦ 10
p ↦ 110

- Can decode from beginning $\rightsquigarrow$ fine for storage / transmission

But how to read $T[9]$ *without full decoding? How to know where its codeword starts?*

- We don't. But we can store it!
  - Naive way: Store starting index for each char

$\rightsquigarrow$ n numbers in $[n]$ $\rightsquigarrow$ $n \lg n$ bits.

Much more than the (compressed) text!

Computing over compressed data

$T[0..\overset{m}{\cancel{n}}) =$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
a	b	a	n	a	n	a	a	n	d	a	n	a	p	p	l	e

Huffman Code

a ↦ 0
b ↦ 11100
d ↦ 11101
e ↦ 11110
l ↦ 11111
n ↦ 10
p ↦ 110

$C[0..\overset{n}{\cancel{m}}) =$ `01110001001000101110101001101101111111110` (positions 0–40)

1100001101

↳ select(9)

- Can decode from beginning ⇝ fine for storage / transmission

But how to read $T[9]$ without full decoding? How to know where its codeword starts?

- We don't. But we can store it!
    - Naive way: Store starting index for each char

⇝ n numbers in $[n]$ ⇝ $n \lg n$ bits.

Much more than the (compressed) text!

entropy-compressed bitvector

Clever **succinct index** supports random access in constant time with $o(n)$ extra bits!

Dynamic Bitvectors

Recall: Dynamic rank/select lower bounds (Fredman-Saks)

$\Omega\left(\frac{\lg n}{\lg \lg n}\right)$ cell probes

Dynamic Bitvectors

Recall: Dynamic rank/select lower bounds (Fredman-Saks)

Note: Can use bitvector to represent n numbers from universe $[0..u)$

- $\rightsquigarrow$ Cannot have dynamic bitvectors **and** fast rank/select queries, $o\left(\frac{\lg n}{\lg \lg n}\right)$.
- $\rightsquigarrow$ *Focus here on static bitvectors.*

6.2 Bitvectors

Bitvector Representation in C++

- `std::vector<char/int>`, `bool[]`
  - easy to access
  - fastest read and write access
  - ≥ 1 **byte** per bit

Bitvector Representation in C++

- `std::vector<char/int>`, `bool[]`
    - easy to access
    - fastest read and write access
    - ≥ 1 **byte** per bit
- `std::vector<`~~`bool`~~ bool`>`
    - special overload
    - *may* use only 1 bit each
    - (with my GNU g++ it used same space as `vector<char>`)

Bitvector Representation in C++

- `std::vector<char/int>`, `bool[]`
    - easy to access
    - fastest read and write access
    - ≥ 1 **byte** per bit
- `std::vector<bool>`
    - special overload
    - *may* use only 1 bit each
    - (with my GNU g++ it used same space as `vector<char>`)
- `std::vector<uint64_t>`
    - store 64 bits in one 8 byte integer
    - use bit tricks to access and write individual bits

Bitvector Representation in C++

- `std::vector<char/int>`, `bool[]`
  - easy to access
  - fastest read and write access
  - ≥ 1 **byte** per bit
- `std::vector<bool>`
  - special overload
  - *may* use only 1 bit each
  - (with my GNU g++ it used same space as `vector<char>`)
- `std::vector<uint64_t>`
  - store 64 bits in one 8 byte integer
  - use bit tricks to access and write individual bits

$\rightsquigarrow$ in practice: `sdsl::bit_vector` (based on packing bits to words)

- from external library SDSL Lite
- `https://github.com/simongog/sdsl-lite`

Rank & Select on Bitvectors

- $B[0..n)$ static array of n bits (Boolean array).
  - easy to store using n bits of space

Rank & Select on Bitvectors

- $B[0..n)$ static array of n bits (Boolean array).
  - easy to store using n bits of space
- $B.\texttt{rank}(i) = \#\,\texttt{1}\text{s in } B[0..i)$ (first $i \in [0..n]$ positions) (=prefix sum)
- $$\begin{aligned} B.\texttt{select}_{1}(r) &= \min\left\{j : B.\texttt{rank}(j+1) \ge r\right\} \cup \{n\} \\ &= \text{index of } r\text{th } \texttt{1} \text{ in } B,\ (r = 1, 2, \ldots) \end{aligned}$$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	1	0	1	0	0	0	0	0	0	1	1	0	1	1	0	1	1	1	1	1	1	0	1	1	1	1	1	1	0	0	0

Rank & Select on Bitvectors

- $B[0..n)$ static array of n bits (Boolean array).
  - easy to store using n bits of space
- $B.\mathtt{rank}(i) = \#\,\mathtt{1}\text{s in } B[0..i)$ (first $i \in [0..n]$ positions) (=prefix sum)
- $$\begin{aligned} B.\mathtt{select}_1(r) &= \min\left\{j : B.\mathtt{rank}(j+1) \ge r\right\} \cup \{n\} \\ &= \text{index of } r\text{th } \mathtt{1} \text{ in } B,\ (r = 1, 2, \ldots) \end{aligned}$$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	1	0	1	0	0	0	0	0	0	1	1	0	1	1	0	1	1	1	1	1	1	0	1	1	1	1	1	1	0	0	0

- $\mathtt{rank}(12) = \mathtt{rank}(13) = 4$

Rank & Select on Bitvectors

- $B[0..n)$ static array of n bits (Boolean array).
    - easy to store using n bits of space
- $B.\mathtt{rank}(i) = \#\,\mathtt{1}\mathrm{s}$ in $B[0..i)$ (first $i \in [0..n]$ positions)   (=prefix sum)
- $$\begin{aligned} B.\mathtt{select_1}(r) &= \min\bigl\{j : B.\mathtt{rank}(j+1) \ge r\bigr\} \cup \{n\} \\ &= \text{index of } r\text{th } \mathtt{1} \text{ in } B, \ (r = 1, 2, \ldots) \end{aligned}$$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	1	0	1	0	0	0	0	0	0	1	1	0	1	1	0	1	1	1	1	1	1	0	1	1	1	1	1	1	0	0	0

- $\mathtt{rank}(12) = \mathtt{rank}(13) = 4$
- $\mathtt{select_1}(3) = 10$

Rank & Select on Bitvectors

- $B[0..n)$ static array of n bits (Boolean array).
  - easy to store using n bits of space
- $B.\texttt{rank}(i)$ = # 1s in $B[0..i)$ (first $i \in [0..n]$ positions) (=prefix sum)
- $$\begin{aligned} B.\texttt{select}_1(r) &= \min\bigl\{j : B.\texttt{rank}(j+1) \ge r\bigr\} \cup \{n\} \\ &= \text{index of } r\text{th } \texttt{1} \text{ in } B, \ (r = 1, 2, \ldots) \end{aligned}$$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	1	0	1	0	0	0	0	0	0	1	1	0	1	1	0	1	1	1	1	1	1	0	1	1	1	1	1	1	0	0	0

- $\texttt{rank}(12) = \texttt{rank}(13) = 4$
- $\texttt{select}_1(3) = 10$
- $\texttt{select}_1(4) = 11$

Rank & Select on Bitvectors

- $B[0..n)$ static array of n bits (Boolean array).
  - easy to store using n bits of space
- $B.\mathtt{rank}(i) = \#\,\mathtt{1}\text{s in } B[0..i)$ (first $i \in [0..n]$ positions) (=prefix sum)
- $$\begin{aligned} B.\mathtt{select}_1(r) &= \min\bigl\{j : B.\mathtt{rank}(j+1) \ge r\bigr\} \cup \{n\} \\ &= \text{index of } r\text{th } \mathtt{1} \text{ in } B, \ (r = 1, 2, \ldots) \end{aligned}$$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	1	0	1	0	0	0	0	0	0	1	1	0	1	1	0	1	1	1	1	1	1	0	1	1	1	1	1	1	0	0	0

- $\mathtt{rank}(12) = \mathtt{rank}(13) = 4$
- $\mathtt{select}_1(3) = 10$
- $\mathtt{select}_1(4) = 11$

$\rightsquigarrow$ $B[i] \;=\; B.\mathtt{rank}(i+1) - B.\mathtt{rank}(i)$ (no need to discuss access separately)

Rank & Select on Bitvectors

- $B[0..n)$ static array of n bits (Boolean array).
  - easy to store using n bits of space
- $B.\mathtt{rank}(i) = \#\,\mathtt{1}\text{s in } B[0..i)$ (first $i \in [0..n]$ positions) (=prefix sum)
- $$\begin{aligned} B.\mathtt{select}_1(r) &= \min\{j : B.\mathtt{rank}(j+1) \ge r\} \cup \{n\} \\ &= \text{index of } r\text{th } \mathtt{1} \text{ in } B,\ (r = 1, 2, \ldots) \end{aligned}$$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	1	0	1	0	0	0	0	0	0	1	1	0	1	1	0	1	1	1	1	1	1	0	1	1	1	1	1	1	0	0	0

- $\mathtt{rank}(12) = \mathtt{rank}(13) = 4$
- $\mathtt{select}_1(3) = 10$
- $\mathtt{select}_1(4) = 11$

$\rightsquigarrow$ $B[i] \;=\; B.\mathtt{rank}(i+1) - B.\mathtt{rank}(i)$ (no need to discuss access separately)

Goal: Support rank and select on bitvector using $n + \underline{o(n)}$ bits of space.

Versatile Bitvectors

- **Set of integers**
  - $S \subset [0..u) \quad \rightsquigarrow \quad B[0..u)$ with $B[x] = 1 \iff x \in S$
  - $\rightsquigarrow$ $B.\mathtt{select}(B.\mathtt{rank}(x+1)) \;=\; S.\mathtt{predecessor}(x)$
  - $\rightsquigarrow$ efficient intersection, union, etc. via bitwise logical operations
  - used, e. g., for **“Roaring Bitmaps”**: one bitvector / integer set per attribute
- **Unary encoded integers**
  - integers $x_1, \ldots, x_k \in \mathbb{N}_{\ge 0} \quad \rightsquigarrow \quad B = \mathtt{1}^{x_1}\mathtt{0}\,\mathtt{1}^{x_2}\mathtt{0}\,\ldots\,\mathtt{1}^{x_k}\mathtt{0}$
  - $x_1 + \cdots + x_j \;=\; B.\mathtt{rank}\big(B.\mathtt{select}_{\mathbf{0}}(j)\big)$
  - x_j as difference of prefix sums
- **Sparse array index** $n \lg u$ $n + k \cdot \lg u + o(u)$
  - suppose $A[0..n)$ is mostly 0, but with some k indices having non-zero values
  - $\rightsquigarrow$ Store in $\underline{V[0..k)}$ all non-zero values compactly, in $B[0..n)$ store $\underline{B[i] = 1} \iff A[i] \ne 0$
  - $\rightsquigarrow$ Simulate $A[x] \;=\; \begin{cases} 0 & \text{if } B[x] = 0 \\ V\big[B.\mathtt{rank}[x]\big] & \text{else} \end{cases}$
- **Note:** In all these cases, rather natural to expect skewed frequencies of `1`s and `0`s.

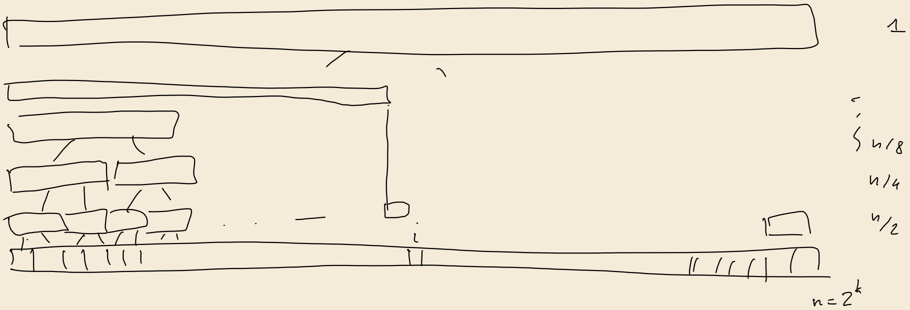
6.3 Supporting Rank

Rank – Simple Ideas

- can answer rank by linear scan . . . $O(n)$ time

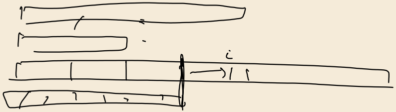
Rank – Simple Ideas

- can answer rank by linear scan . . . $O(n)$ time
- can maintain a ***segment tree*** with subrange sums
  - ⇝ $O(\lg n)$ query time, but also $n \lg n$ bits of extra space!
  - can support updates (bit flips)
- can store all answers ⇝ $O(1)$ time, but $n \lg n$ extra space



Rank – Simple Ideas

- can answer rank by linear scan . . . $O(n)$ time
- can maintain a ***segment tree*** with subrange sums
  - ⇝ $O(\lg n)$ query time, but also $n \lg n$ bits of extra space!
  - can support updates (bit flips)
- can store all answers ⇝ $O(1)$ time, but $n \lg n$ extra space
- hybrid: **subsampled segment tree**
  - store subrange sums only for blocks of $\lg n$ bits
  - within block use linear scan
  - entire blocks covered by segment tree ⇝ $O(\log n)$ query time
  - space: $\Theta\left(\frac{n}{\lg n} \cdot \lg n\right) = \Theta(n)$ bits of extra space
  - ⇝ a bit better, but still neither succinct, nor constant time!
    (but dynamic!)



Rank – Simple Ideas

- can answer rank by linear scan . . . $O(n)$ time
- can maintain a ***segment tree*** with subrange sums
  - $\rightsquigarrow$ $O(\lg n)$ query time, but also $n \lg n$ bits of extra space!
  - can support updates (bit flips)
- can store all answers $\rightsquigarrow$ $O(1)$ time, but $n \lg n$ extra space
- hybrid: **subsampled segment tree**
  - store subrange sums only for blocks of $\lg n$ bits
  - within block use linear scan
  - entire blocks covered by segment tree $\rightsquigarrow$ $O(\log n)$ query time
  - space: $\Theta\left(\frac{n}{\lg n} \cdot \lg n\right) = \Theta(n)$ bits of extra space
  - $\rightsquigarrow$ a bit better, but still neither succinct, nor constant time!
    (but dynamic!)
  - Could apply this on larger block sizes and recursively to reduce space, but time remains.

Rank – Simple Ideas

- can answer rank by linear scan . . . $O(n)$ time
- can maintain a ***segment tree*** with subrange sums
  - ⇝ $O(\lg n)$ query time, but also $n \lg n$ bits of extra space!
  - can support updates (bit flips)
- can store all answers ⇝ $O(1)$ time, but $n \lg n$ extra space
- hybrid: **subsampled segment tree**
  - store subrange sums only for blocks of $\lg n$ bits
  - within block use linear scan
  - entire blocks covered by segment tree ⇝ $O(\log n)$ query time
  - space: $\Theta\left(\frac{n}{\lg n} \cdot \lg n\right) = \Theta(n)$ bits of extra space
  - ⇝ a bit better, but still neither succinct, nor constant time!
    (but dynamic!)
  - Could apply this on larger block sizes and recursively to reduce space, but time remains.

How can we do better?

Rank Index for Bitvector

- Apart from B, we store:

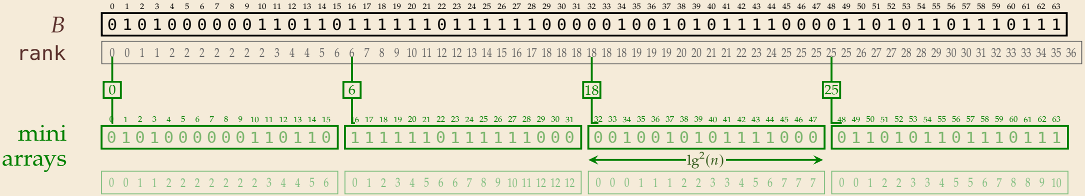
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
B	0	1	0	1	0	0	0	0	0	0	1	1	0	1	1	0	1	1	1	1	1	1	0	1	1	1	1	1	1	0	0	0

	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
B	0	0	1	0	0	1	0	1	0	1	1	1	1	0	0	0	0	1	1	0	1	0	1	1	0	1	1	1	0	1	1	1

Rank Index for Bitvector

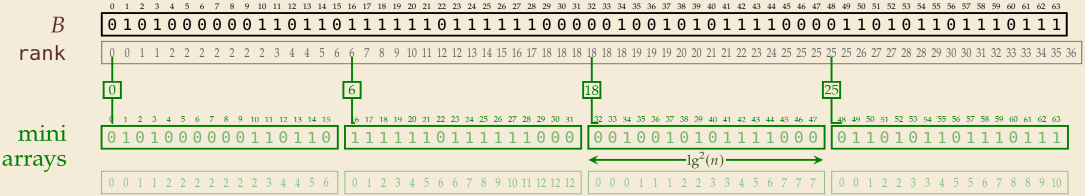
- Apart from B, we store:
  1. Rank of first pos. of each **mini array** of $L = \lg^2(n)$ bits

$$\#\text{mini arrays} = \left\lceil \frac{n}{L} \right\rceil = O\left(\frac{n}{\lg^2 n}\right) \rightarrow \text{can store } O(\lg n) \text{ bits each}$$



Rank Index for Bitvector

- Apart from B, we store:
  1. Rank of first pos. of each **mini array** of $L = \lg^2(n)$ bits $\rightsquigarrow \frac{n}{L} \cdot \lg(n) = \frac{n}{\lg n} = o(n)$ bits

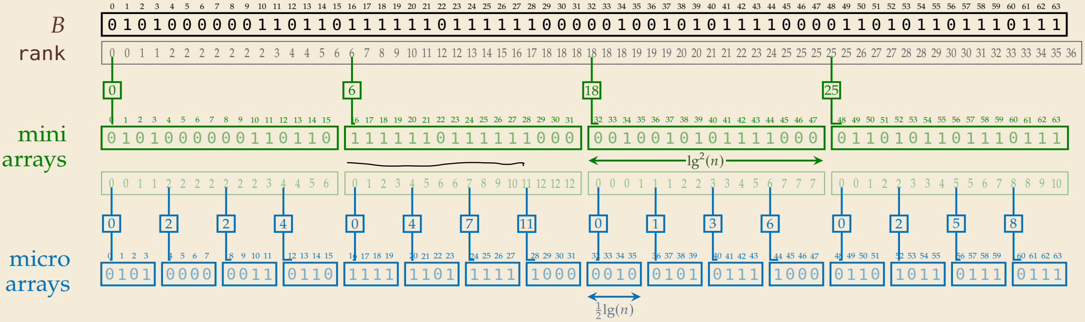


Rank Index for Bitvector

- Apart from B, we store:
  1. Rank of first pos. of each **mini array** of $L = \lg^2(n)$ bits $\rightsquigarrow \frac{n}{L} \cdot \lg(n) = \frac{n}{\lg n} = o(n)$ bits
  2. Rank of first pos. in **micro array** of $\ell = \frac{1}{2}\lg(n)$ bits *relative* to its mini array

#micro arrays $O\left(\frac{n}{\lg n}\right)$

#1s in mini array $\leq L$



0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63

B 0 1 0 1 0 0 0 0 0 0 1 1 0 1 1 0 1 1 1 1 1 1 0 1 1 1 1 1 1 0 0 0 0 0 1 0 0 1 0 1 0 1 1 1 1 0 0 0 0 1 1 0 1 0 1 1 0 1 1 1 0 1 1 1

rank 0 0 1 1 2 2 2 2 2 2 2 3 4 4 5 6 6 7 8 9 10 11 12 12 13 14 15 16 17 18 18 18 18 18 18 19 19 19 20 20 21 21 22 23 24 25 25 25 25 25 26 27 27 28 28 29 30 30 31 32 33 33 34 35 36

0 6 18 25

mini arrays

0 1 0 1 0 0 0 0 0 0 1 1 0 1 1 0 | 1 1 1 1 1 1 0 1 1 1 1 1 1 0 0 0 | 0 0 1 0 0 1 0 1 0 1 1 1 1 0 0 0 | 0 1 1 0 1 0 1 1 0 1 1 1 0 1 1 1

$\lg^2(n)$

0 0 1 1 2 2 2 2 2 2 2 3 4 4 5 6 | 0 1 2 3 4 5 6 6 7 8 9 10 11 12 12 12 | 0 0 0 1 1 1 2 2 3 3 4 5 6 7 7 7 | 0 0 1 2 2 3 3 4 5 5 6 7 8 8 9 10

0 2 2 4 0 4 7 11 0 1 3 6 0 2 5 8

micro arrays

0101 0000 0011 0110 1111 1101 1111 1000 0010 0101 0111 1000 0110 1011 0111 0111

$\frac{1}{2}\lg(n)$

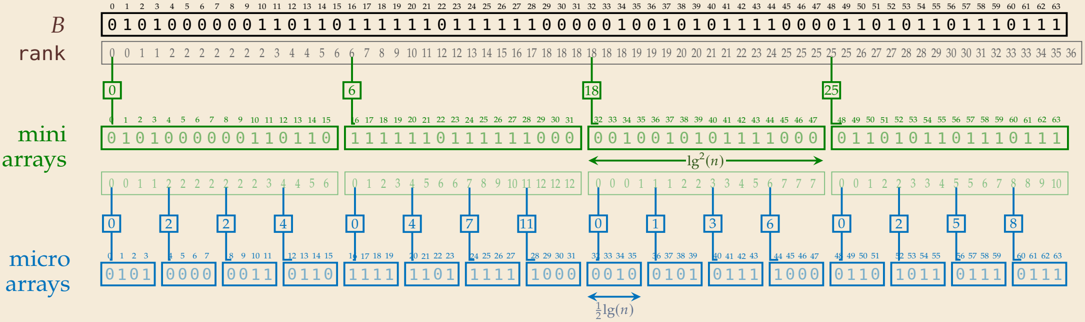
Rank Index for Bitvector

- Apart from B, we store:

  1. Rank of first pos. of each **mini array** of $L = \lg^2(n)$ bits $\rightsquigarrow$ $\frac{n}{L} \cdot \lg(n) = \frac{n}{\lg n} = o(n)$ bits

  2. Rank of first pos. in **micro array** of $\ell = \frac{1}{2}\lg(n)$ bits *relative* to its mini array

     $\rightsquigarrow$ ranks are numbers in $[0..L]$ $\rightsquigarrow$ $\lg L = 2 \lg\lg n$ bits suffice for each



Rank Index for Bitvector

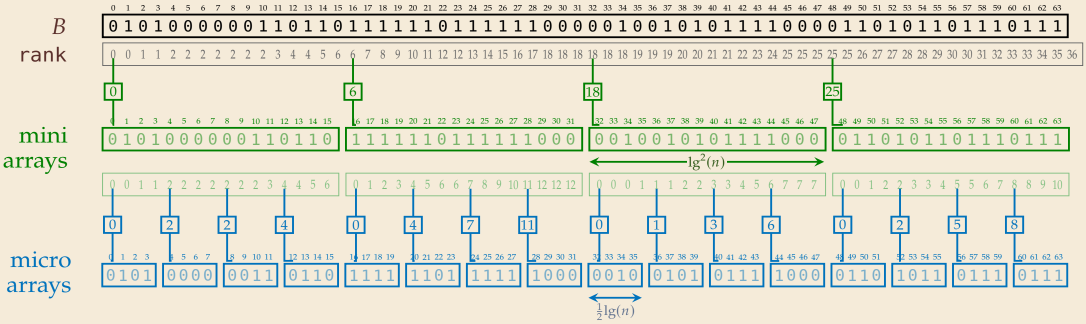
- Apart from B, we store:

1. Rank of first pos. of each **mini array** of $L = \lg^2(n)$ bits $\rightsquigarrow \frac{n}{L} \cdot \lg(n) = \frac{n}{\lg n} = o(n)$ bits

2. Rank of first pos. in **micro array** of $\ell = \frac{1}{2}\lg(n)$ bits *relative* to its mini array

   $\rightsquigarrow$ ranks are numbers in $[0..L]$ $\rightsquigarrow$ $\lg L = 2\lg\lg n$ bits suffice for each

   $\rightsquigarrow \frac{n}{\ell}\lg L = \frac{n}{\frac{1}{2}\lg n} \cdot 2\lg\lg n = \frac{4n\lg\lg n}{\lg n} = o(n)$ bits



B: 0 1 0 1 0 0 0 0 0 0 1 1 0 1 1 0 1 1 1 1 1 1 0 1 1 1 1 1 1 0 0 0 0 0 1 0 0 1 0 1 0 1 1 1 1 0 0 0 0 1 1 0 1 0 1 1 0 1 1 1 0 1 1 1

`rank`: 0 0 1 1 2 2 2 2 2 2 2 3 4 4 5 6 6 7 8 9 10 11 12 12 13 14 15 16 17 18 18 18 18 18 18 19 19 19 20 20 21 21 22 23 24 25 25 25 25 25 26 27 27 28 28 29 30 30 31 32 33 33 34 35 36

mini arrays: 0 | 6 | 18 | 25

0 1 0 1 0 0 0 0 0 0 1 1 0 1 1 0 | 1 1 1 1 1 1 0 1 1 1 1 1 1 0 0 0 | 0 0 1 0 0 1 0 1 0 1 1 1 1 0 0 0 | 0 1 1 0 1 0 1 1 0 1 1 1 0 1 1 1

$\lg^2(n)$

0 0 1 1 2 2 2 2 2 2 2 3 4 4 5 6 | 0 1 2 3 4 5 6 6 7 8 9 10 11 12 12 12 | 0 0 0 1 1 1 2 2 3 3 4 5 6 7 7 7 | 0 0 1 2 2 3 3 4 5 5 6 7 8 8 9 10

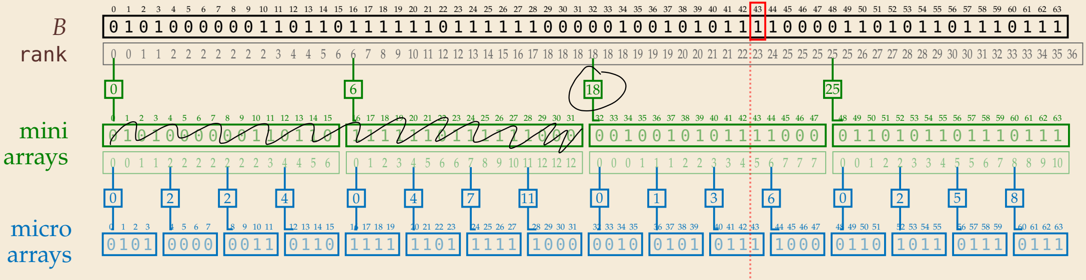
micro arrays: 0 2 2 4 | 0 4 7 11 | 0 1 3 6 | 0 2 5 8

0101 0000 0011 0110 | 1111 1101 1111 1000 | 0010 0101 0111 1000 | 0110 1011 0111 0111

$\frac{1}{2}\lg(n)$

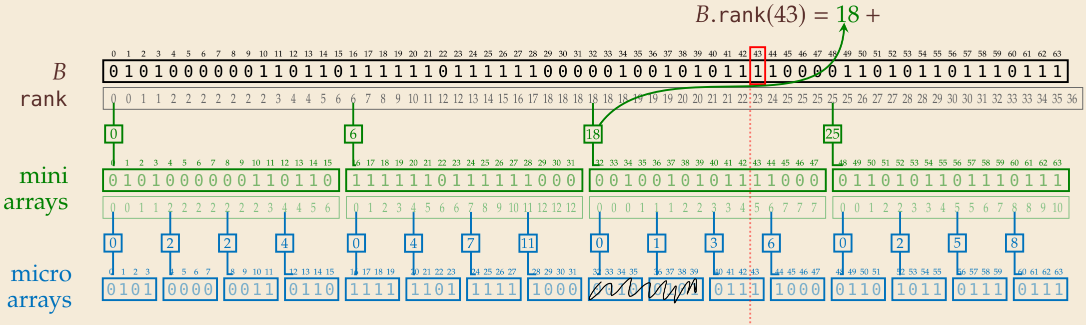
How to find rank

$B.\mathtt{rank}(43) =$



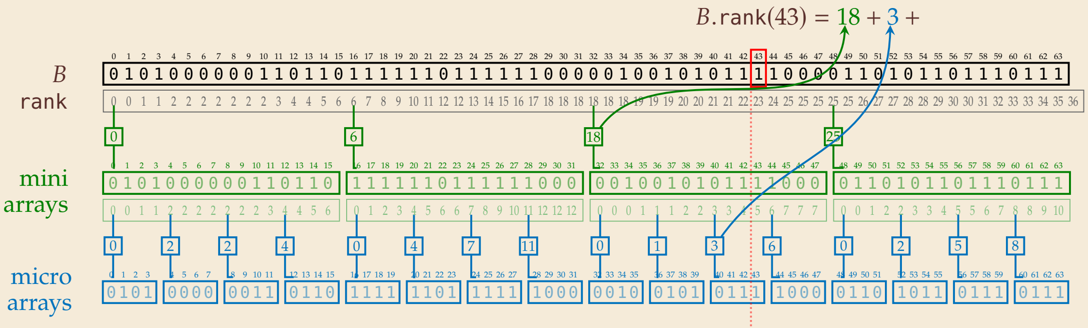
- How to compute $B.\mathtt{rank}(i)$?

How to find rank



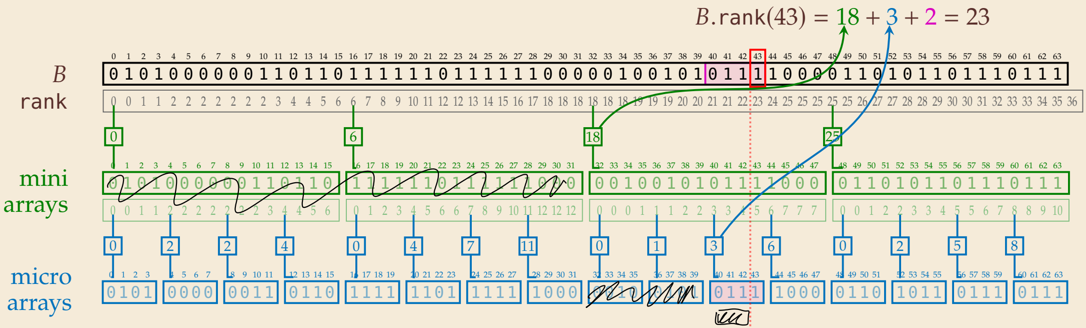
- How to compute *B*.`rank`(*i*)?
    - find rank up to element's mini array

How to find rank



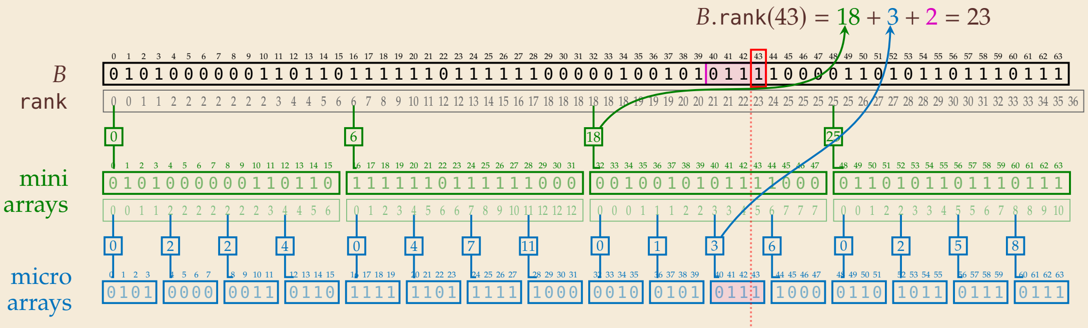
- How to compute $B.\texttt{rank}(i)$?
  - find rank up to element's mini array
  - add rank up to element's micro array (mini-array local)

How to find rank


$B.\mathtt{rank}(43) = 18 + 3 + 2 = 23$

- How to compute $B.\mathtt{rank}(i)$?
  - find rank up to element's mini array
  - add rank up to element's micro array (mini-array local)
  - add micro-array-local rank of position

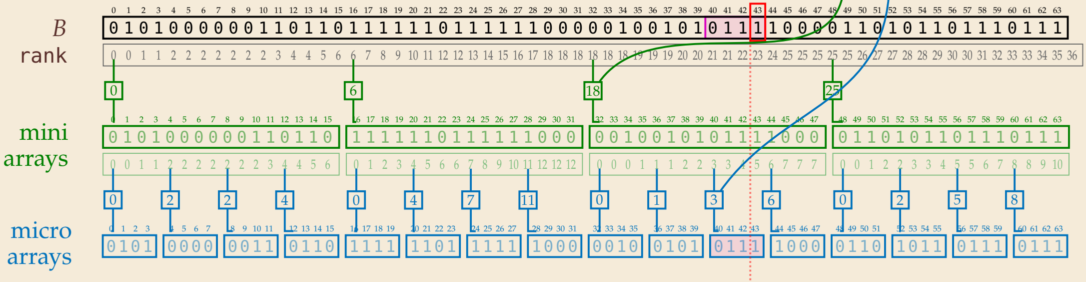
How to find rank


$B.\texttt{rank}(43) = 18 + 3 + 2 = 23$

- How to compute $B.\texttt{rank}(i)$?
  - find rank up to element's mini array
  - add rank up to element's micro array (mini-array local)
  - add micro-array-local rank of position
    - can either do this naively by scanning $\frac{1}{2}\lg n$ bits $\rightsquigarrow$ $O(\log n)$ time

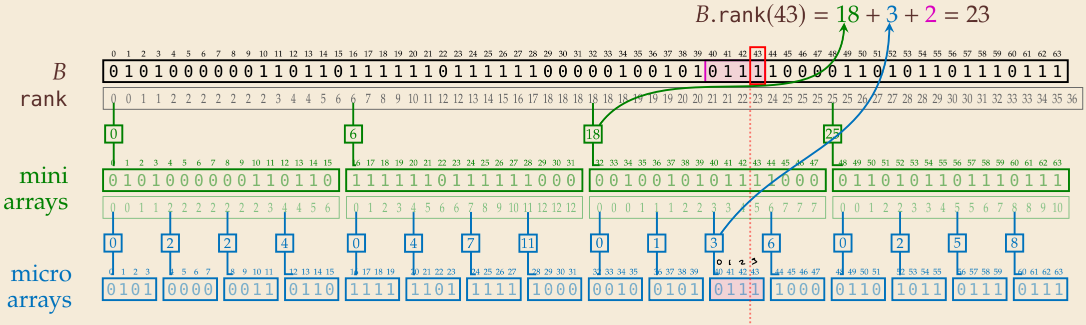
How to find rank

$B.\texttt{rank}(43) = 18 + 3 + 2 = 23$



- How to compute $B.\texttt{rank}(i)$?
  - find rank up to element's mini array
  - add rank up to element's micro array (mini-array local)
  - add micro-array-local rank of position
    - can either do this naively by scanning $\frac{1}{2}\lg n$ bits $\rightsquigarrow$ $O(\log n)$ time
    - or use bit marks and pop-count instructions on CPUs $\rightsquigarrow$ $O(1)$ time

How to find rank



$B.\texttt{rank}(43) = 18 + 3 + 2 = 23$

- How to compute $B.\texttt{rank}(i)$?
  - find rank up to element's mini array
  - add rank up to element's micro array (mini-array local)
  - add micro-array-local rank of position
    - can either do this naively by scanning $\frac{1}{2}\lg n$ bits $\rightsquigarrow$ $O(\log n)$ time
    - or use bit marks and pop-count instructions on CPUs $\rightsquigarrow$ $O(1)$ time
    - or use exhaustive *lookup table!* $\frac{1}{2}\lg n$ bits $\rightsquigarrow$ only $2^{1/2 \lg n} = \sqrt{n}$ different micro arrays

Exhaustive Tabulation

Classic word-RAM technique: bootstrap on small cases with *exhaustive tabulation*

- **Idea 1:** on word-RAM, can use integers (and hence small bit sequences) as array index
- **Idea 2:** for mirco subproblems on tiny bit seqeuences, cannot have many *different* micro subproblems

$\rightsquigarrow$ Precompute all micro subproblems in a lookup table, indexed by the subproblem's bitpattern

$\rightsquigarrow$ $O(1)$ time for micro subproblem (after preprocessing)

Back to Rank!

Recall: Our task is to compute the local rank in a micro array of $\ell = \frac{1}{2}\lg n$ bits.

Back to Rank!

Recall: Our task is to compute the local rank in a micro array of $\ell = \frac{1}{2} \lg n$ bits.

Preprecessing: Build table for all possible queries for all *possible* micro arrays

$$\mu R[0..2^{\ell})[0..\ell) \;=$$

idx	**micro array**	rank(0)	rank(1)	rank(2)	rank(3)	rank(4)
0	`0000`	0	0	0	0	0
1	`0001`	0	0	0	0	1
2	`0010`	0	0	0	1	1
3	`0011`	0	0	0	1	2
4	`0100`	0	0	1	1	1
5	`0101`	0	0	1	1	2
6	`0110`	0	0	1	2	2
7	`0111`	0	0	1	2	3
8	`1000`	0	1	1	1	1
9	`1001`	0	1	1	1	2
10	`1010`	0	1	1	2	2
11	`1011`	0	1	1	2	3
12	`1100`	0	1	2	2	2
13	`1101`	0	1	2	2	3
14	`1110`	0	1	3	3	3
15	`1111`	0	1	2	3	4

Back to Rank!

Recall: Our task is to compute the local rank in a micro array of $\ell = \frac{1}{2}\lg n$ bits.

Preprecessing: Build table for all possible queries for all *possible* micro arrays

$$\mu R[0..2^{\ell})[0..\ell) \;=$$

idx	micro array	rank(0)	rank(1)	rank(2)	rank(3)	rank(4)
0	`0000`	0	0	0	0	0
1	`0001`	0	0	0	0	1
2	`0010`	0	0	0	1	1
3	`0011`	0	0	0	1	2
4	`0100`	0	0	1	1	1
5	`0101`	0	0	1	1	2
6	`0110`	0	0	1	2	2
7	`0111`	0	0	1	2	3
8	`1000`	0	1	1	1	1
9	`1001`	0	1	1	1	2
10	`1010`	0	1	1	2	2
11	`1011`	0	1	1	2	3
12	`1100`	0	1	2	2	2
13	`1101`	0	1	2	2	3
14	`1110`	0	1	3	3	3
15	`1111`	0	1	2	3	4

Space: $\underset{\#\text{rows}}{2^{\ell}} \cdot \underset{\#\text{col}}{\ell} \cdot \lg \ell \;=\; \sqrt{n} \cdot \operatorname{polylog}(n) \;=\; o(n)$

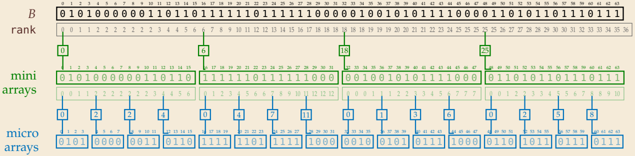
Back to Rank!

Recall: Our task is to compute the local rank in a micro array of $\ell = \frac{1}{2}\lg n$ bits.

Preprecessing: Build table for all possible queries for all *possible* micro arrays

$$\mu R[0..2^{\ell})[0..\ell) \;=$$

idx	micro array	rank(0)	rank(1)	rank(2)	rank(3)	rank(4)
0	0000	0	0	0	0	0
1	0001	0	0	0	0	1
2	0010	0	0	0	1	1
3	0011	0	0	0	1	2
4	0100	0	0	1	1	1
5	0101	0	0	1	1	2
6	0110	0	0	1	2	2
7	0111	0	0	1	2	3
8	1000	0	1	1	1	1
9	1001	0	1	1	1	2
10	1010	0	1	1	2	2
11	1011	0	1	1	2	3
12	1100	0	1	2	2	2
13	1101	0	1	2	2	3
14	1110	0	1	3	3	3
15	1111	0	1	2	3	4



- Note: #micro arrays $= \dfrac{n}{\frac{1}{2}\lg n} \gg$

  #*different* micro arrays $= 2^{1/2 \lg n} = \sqrt{n}$

Space: $2^{\ell} \cdot \ell \cdot \lg \ell \;=\; \sqrt{n} \cdot \mathrm{polylog}(n) \;=\; o(n)$

Back to Rank!

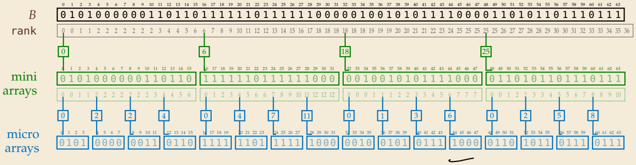
Recall: Our task is to compute the local rank in a micro array of $\ell = \frac{1}{2} \lg n$ bits.

Prepecessing: Build table for all possible queries for all *possible* micro arrays

$\mu R[0..2^{\ell})[0..\ell) =$

idx	micro array	rank(0)	rank(1)	rank(2)	rank(3)	rank(4)
0	0000	0	0	0	0	0
1	0001	0	0	0	0	1
2	0010	0	0	0	1	1
3	0011	0	0	0	1	2
4	0100	0	0	1	1	1
5	0101	0	0	1	1	2
6	0110	0	0	1	2	2
7	0111	0	0	1	2	3
8	1000	0	1	1	1	1
9	1001	0	1	1	1	2
10	1010	0	1	1	2	2
11	1011	0	1	1	2	3
12	1100	0	1	2	2	2
13	1101	0	1	2	2	3
14	1110	0	1	3	3	3
15	1111	0	1	2	3	4

Space: $2^{\ell} \cdot \ell \cdot \lg \ell = \sqrt{n} \cdot \mathrm{polylog}(n) = o(n)$



- Note: #micro arrays $= \frac{n}{\frac{1}{2}\lg n} \gg$

  #different micro arrays $= 2^{1/2 \lg n} = \sqrt{n}$

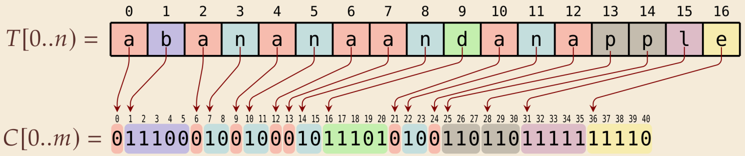
- To access micro-array rank(i)
  1. Read micro array *type* (bits), here from B
  2. Treat *type* as binary number $\rightsquigarrow$ idx t
  3. Return $\mu R[t][i]$

Succinct Rank – Result

We have shown:

Theorem 6.1 (Succinct bitvector with rank)

Given a bitvector $B[0..n)$, we can construct in $O(n)$ time an index data structure using $O\left(n\frac{\lg\lg n}{\lg n}\right) = o(n)$ bits of space that can answer $B.\mathtt{rank}(i)$ queries in $O(1)$.
The index requires oracle access to B at query time, i. e., in $O(1)$ time we must be able to read $\Theta(\log n)$ consecutive bits of B. ◀



S[0..m) 1 1 0000 1 1 0 1 1 0 1 1 1 0 1 0 0 0 0 0 1 1 0 1 1 0 0 1 0 0 1 0 0 0 0 1 0 0 0 0

S. select (r)

recompute S from C even on demand

Four Russians?

The *exhaustive-tabulation technique* is often called “Four Russians trick” in the literature . . .

- The algorithmic technique was published 1970 by
  V. L. Arlazarov, E. A. Dinitz, M. A. Kronrod, and I. A. Faradžev

  📕 **Arlazarov, Dinitz, Kronrod, Faradžev**: *On economical construction of the transitive closure of a directed graph*, Dokl. Akad. Nauk SSSR **1970**

  inofficial translation: `https://minorfree.github.io/FourRussian/`

- all worked in Moscow at that time . . . but Soviet ≠ Russian; Dinitz emigrated to Isreal

  (Arlazarov and Kronrod are Russian)

Four Russians?

The *exhaustive-tabulation technique* is often called “Four Russians trick” in the literature . . .

- The algorithmic technique was published 1970 by
  V. L. Arlazarov, E. A. Dinitz, M. A. Kronrod, and I. A. Faradžev

  Arlazarov, Dinitz, Kronrod, Faradžev: *On economical construction of the transitive closure of a directed graph*, Dokl. Akad. Nauk SSSR **1970**

  inofficial translation: `https://minorfree.github.io/FourRussian/`

- all worked in Moscow at that time . . . but Soviet $\neq$ Russian; Dinitz emigrated to Isreal

  (Arlazarov and Kronrod are Russian)

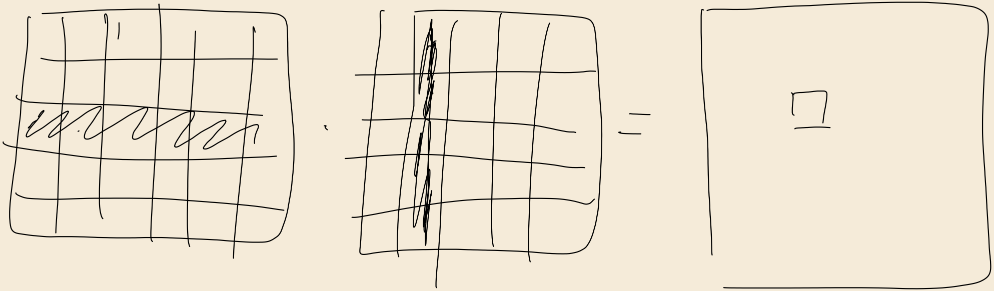
- American authors coined the othering term “Method of Four Russians”
  . . . name in widespread use

The Original Application: Boolean Matrix Multiplication

Suppose we want to multiply two $n \times n$ Boolean matrices $C = A \cdot B$.

We divide A, B, and C into $\ell \times \ell$ *micro matrices*.

$\rightsquigarrow$ C consists of $\left(\frac{n}{\ell}\right)^2$ micro matrices, each of which is the sum of $\frac{n}{\ell}$ micro-matrix products.



The Original Application: Boolean Matrix Multiplication

Suppose we want to multiply two $n \times n$ Boolean matrices $C = A \cdot B$.

We divide A, B, and C into $\ell \times \ell$ *micro matrices*.

⇝ C consists of $\left(\frac{n}{\ell}\right)^2$ micro matrices, each of which is the sum of $\frac{n}{\ell}$ micro-matrix products.

The number of *different* possible micro matrix products is $L = 2^{\ell^2} \cdot 2^{\ell^2}$.
If we pick $\ell = \frac{1}{4}\sqrt{\lg n}$, we have only $L = 2^{2\ell^2} = \sqrt{n}$ different products.

⇝ ***Exhaustive Tabulation:*** Can *precompute* all $\sqrt{n}$ *possible* micro-matrix sums/products!

The Original Application: Boolean Matrix Multiplication

Suppose we want to multiply two $n \times n$ Boolean matrices $C = A \cdot B$.

We divide A, B, and C into $\ell \times \ell$ *micro matrices*.

$\rightsquigarrow$ C consists of $\left(\frac{n}{\ell}\right)^2$ micro matrices, each of which is the sum of $\frac{n}{\ell}$ micro-matrix products.

The number of *different* possible micro matrix products is $L = 2^{\ell^2} \cdot 2^{\ell^2}$.
If we pick $\ell = \frac{1}{4}\sqrt{\lg n}$, we have only $L = 2^{2\ell^2} = \sqrt{n}$ different products.

$\rightsquigarrow$ ***Exhaustive Tabulation:*** Can *precompute* all $\sqrt{n}$ *possible* micro-matrix sums/products!

For two micro matrices a and b, we store $a \cdot b$ at the offset $a_{1,1} \ldots a_{\ell,\ell} b_{1,1} \ldots b_{\ell,\ell}$, where we interpret this bitstring as a binary number.
On a word RAM, we can use this as indirect memory access in $O(1)$ time.

$\rightsquigarrow$ Any micro matrix sum/product takes $O(1)$ time
after a total of $O(\sqrt{n} \cdot \log^{3/2} n)$ preprocessing.

The Original Application: Boolean Matrix Multiplication

Suppose we want to multiply two $n \times n$ Boolean matrices $C = A \cdot B$.

We divide A, B, and C into $\ell \times \ell$ *micro matrices*.

$\rightsquigarrow$ C consists of $\left(\frac{n}{\ell}\right)^2$ micro matrices, each of which is the sum of $\frac{n}{\ell}$ micro-matrix products.

The number of *different* possible micro matrix products is $L = 2^{\ell^2} \cdot 2^{\ell^2}$.
If we pick $\ell = \frac{1}{4}\sqrt{\lg n}$, we have only $L = 2^{2\ell^2} = \sqrt{n}$ different products.

$\rightsquigarrow$ ***Exhaustive Tabulation:*** Can *precompute* all $\sqrt{n}$ *possible* micro-matrix sums/products!

For two micro matrices a and b, we store $a \cdot b$ at the offset $a_{1,1} \ldots a_{\ell,\ell} b_{1,1} \ldots b_{\ell,\ell}$, where we interpret this bitstring as a binary number.
On a word RAM, we can use this as indirect memory access in $O(1)$ time.

$\rightsquigarrow$ Any micro matrix sum/product takes $O(1)$ time
after a total of $O(\sqrt{n} \cdot \log^{3/2} n)$ preprocessing.

The total time to compute one micro matrix in C is thus $O(\frac{n}{\ell})$.
So the total time to compute C is $O(n^3/\ell^3) = O(n^3/\log^{3/2} n)$.

The Original Application: Boolean Matrix Multiplication

Suppose we want to multiply two $n \times n$ Boolean matrices $C = A \cdot B$.

We divide A, B, and C into $\ell \times \ell$ *micro matrices*.

$\leadsto$ C consists of $\left(\frac{n}{\ell}\right)^2$ micro matrices, each of which is the sum of $\frac{n}{\ell}$ micro-matrix products.

The number of *different* possible micro matrix products is $L = 2^{\ell^2} \cdot 2^{\ell^2}$.
If we pick $\ell = \frac{1}{4}\sqrt{\lg n}$, we have only $L = 2^{2\ell^2} = \sqrt{n}$ different products.

$\leadsto$ ***Exhaustive Tabulation:*** Can *precompute* all $\sqrt{n}$ *possible* micro-matrix sums/products!

For two micro matrices a and b, we store $a \cdot b$ at the offset $a_{1,1} \ldots a_{\ell,\ell} b_{1,1} \ldots b_{\ell,\ell}$, where we interpret this bitstring as a binary number.
On a word RAM, we can use this as indirect memory access in $O(1)$ time.

$\leadsto$ Any micro matrix sum/product takes $O(1)$ time
after a total of $O(\sqrt{n} \cdot \log^{3/2} n)$ preprocessing.

The total time to compute one micro matrix in C is thus $O(\frac{n}{\ell})$.
So the total time to compute C is $O(n^3/\ell^3) = O(n^3/\log^{3/2} n)$.

Note: By taking $n \times \ell$ resp. $\ell \times n$ *"micro strips"* instead of squares, we can choose
$\ell = \Theta(\log n)$ and obtain final time $\underline{O(n^3/\log^2 n)}$.

6.4 Supporting Select

Select Indices – Simple Ideas

$$\text{Recall:}\quad B.\mathtt{select_1}(r) \;=\; \min\left\{j : B.\mathtt{rank}(j+1) \ge r\right\} \cup \{n\}$$
$$\qquad\qquad\qquad\qquad\;\; = \;\text{index of } r\text{th } \mathtt{1} \text{ in } B,\ (r = 1, 2, \ldots)$$

Select Indices – Simple Ideas

Recall: $$\begin{aligned} B.\mathtt{select}_1(r) &= \min\left\{j : \underbrace{B.\mathtt{rank}(j+1)} \geq r\right\} \cup \{n\} \\ &= \text{index of } r\text{th } \mathtt{1} \text{ in } B,\ (r = 1, 2, \ldots) \end{aligned}$$

$\rightsquigarrow$ in $O(\log n)$ calls to `rank`, we can answer `select` without extra space

- Not a terrible option if we only do few `select` queries

Select Indices – Simple Ideas

Recall: $$\begin{aligned} B.\mathtt{select}_1(r) &= \min\{j : B.\mathtt{rank}(j+1) \ge r\} \cup \{n\} \\ &= \text{index of } r\text{th } \mathtt{1} \text{ in } B,\ (r = 1, 2, \ldots) \end{aligned}$$

⇝ in $O(\log n)$ calls to `rank`, we can answer `select` without extra space

- Not a terrible option if we only do few `select` queries

- Trivial $O(1)$ solution: store all answers

  ⇝ m indices if $B[0..n)$ contains m `1`s ⇝ $m \lg n$ bits

  - can be sensible solution if m is small

Select Indices – Simple Ideas

Recall: $B.\texttt{select}_1(r) = \min\{j : B.\texttt{rank}(j+1) \ge r\} \cup \{n\}$

$\phantom{B.\texttt{select}_1(r)} = \text{index of } r\text{th } \texttt{1} \text{ in } B, \ (r = 1, 2, \ldots)$

⇝ in $O(\log n)$ calls to `rank`, we can answer `select` without extra space

- Not a terrible option if we only do few `select` queries

- Trivial $O(1)$ solution: store all answers
  - ⇝ m indices if $B[0..n)$ contains m `1`s ⇝ $m \lg n$ bits
  - can be sensible solution if m is small

$m = \Theta(n)$

- hybrid solution by subsampling:
  - explicitly store every kth `select` answer, binary search in between
  - ⇝ $\frac{m}{k} \lg n$ bits
  - worst case of $O(\log n)$ remains

Select Indices – Simple Ideas

Recall: $B.\texttt{select}_1(r) = \min\Big\{j : B.\texttt{rank}(j+1) \ge r\Big\} \cup \{n\}$

$\phantom{B.\texttt{select}_1(r)} = $ index of rth `1` in B, $(r = 1, 2, \ldots)$

$\rightsquigarrow$ in $O(\log n)$ calls to `rank`, we can answer `select` without extra space

- Not a terrible option if we only do few `select` queries

- Trivial $O(1)$ solution: store all answers
  - $\rightsquigarrow$ m indices if $B[0..n)$ contains m `1`s $\rightsquigarrow$ $m \lg n$ bits
  - can be sensible solution if m is small
- hybrid solution by subsampling:
  - explicitly store every kth `select` answer, binary search in between
  - $\rightsquigarrow$ $\frac{m}{k} \lg n$ bits
  - worst case of $O(\log n)$ remains

Can we do better?

Select Index

Theorem 6.2 (Succinct bitvector with select)

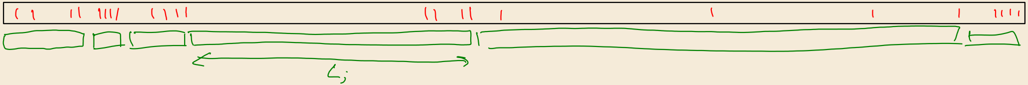
Given a bitvector $B[0..n)$, we can construct in $O(n)$ time an index data structure using $O\left(n\frac{\lg\lg n}{\sqrt{\lg n}}\right) = o(n)$ bits of space that can answer $B.\texttt{select}(r)$ queries in $O(1)$.

The index requires oracle access to B at query time, i. e., in $O(1)$ time we must be able to read $\Theta(\log n)$ consecutive bits of B.

try as w/ rank?

chunk id : $\lg(\#\text{chunks}) = \lg\left(\frac{n}{L}\right) = \lg n - \lg L$

Proof:



Split bitvector into miniblocks of K ones each

$\boxed{K = \lg^2 n}$

$\Rightarrow \#\text{miniblocks} = O\left(\frac{n}{K}\right)$

r th one is in $\left\lfloor \frac{r}{K} \right\rfloor$ th mini block.

Case distinction

- Long miniblock $L_j > \lg^4(n)$

  ⇒ have $\le \frac{n}{\lg^4(n)}$ such long miniblocks each w/ K ones

  ⇒ store indices of K ones explicitly

  $$\frac{n}{\lg^4(n)} \cdot \lg^2(n) \cdot \lg(n) = O\left(\frac{n}{\log(n)}\right) = o(n)$$

- Dense miniblock ($L_j \le \lg^4(n)$) store starting index of mini block

  → can write mini block-local index w/ $O(\lg \lg n)$ bits still too much

recursively subdivide into micro blocks w/ k ones

$$\boxed{k = \sqrt{\lg n}}$$

micro block: $\left\lfloor \frac{r'}{k} \right\rfloor$ $r' = r - \left\lfloor \frac{r}{K} \right\rfloor$

second-level case distinction

- <u>long micro block</u> $\ell_j > \frac{1}{2}\lg n$

  only $\leq \frac{2n}{\lg n}$ such long micro blocks

  $\Rightarrow$ write down indices of ones

$$\frac{2n}{\lg n} \cdot k \cdot O(\lg\lg n) = O\left(\frac{n \lg\lg(n)}{\sqrt{\lg(n)}}\right) = o(n)$$

- <u>dense micro block</u> store mini-block-local starting index

  $\ell_j \leq \frac{1}{2}\lg n$ $\Rightarrow$ can use exhaustive table lookup

  to answer micro-block-local select

Remark: a more complicated index achieves rank & select with $O\left(n \frac{\lg\lg n}{\lg n}\right)$ bits of extra space.

also: for $O(1)$ rank/select in indexing model we need $\Omega\left(n \frac{\lg\lg n}{\lg n}\right)$ bits of extra space.

6.5 Compressed Bitvectors

Recall: Bitvector Applications

- Many applications of bitvectors naturally yield *unbalanced* bitvectors:
  - characteristic vector of set $S \subseteq [0..u)$ $\rightsquigarrow$ n ones among u bits
  - prefix sums via unary encoding $\rightsquigarrow$ #zeros = #numbers; #ones = total sum
  - sparse array index $\rightsquigarrow$ only useful of many fewer ones than entries

Recall: Bitvector Applications

- Many applications of bitvectors naturally yield *unbalanced* bitvectors:
  - characteristic vector of set $S \subseteq [0..u)$ $\rightsquigarrow$ n ones among u bits
  - prefix sums via unary encoding $\rightsquigarrow$ #zeros = #numbers; #ones = total sum
  - sparse array index $\rightsquigarrow$ only useful of many fewer ones than entries

  Consider $B[0..n)$ with m 1-bits
  - w.l.o.g. assume $m \le \frac{n}{2}$ (other case symmetric)

- Number of different bitvectors $B[0..n)$ with m ones: $\binom{n}{m}$ $\quad m = \frac{n}{2} \quad \lg\binom{n}{\frac{n}{2}} \approx n$

$\rightsquigarrow$ Should only use $\lg\binom{n}{m} \le m\lg\left(\frac{n}{m}\right) + 1.4427m$ bits

$m = 0$ / $m = \frac{n}{2}$

$0 \quad \longrightarrow \quad n$

$m = 1$

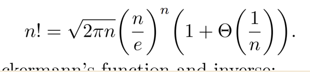
$\lg n$

true always but $m = \Theta(n)$ not tight

interesting range $m = o(n)$

$m = \sqrt{n} \quad \rightarrow \quad \frac{1}{2}\sqrt{n} \cdot \lg n + O(\sqrt{n})$

$$\lg\binom{n}{m} = \lg n! - \lg m! - \lg(n-m)!$$



$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+\Theta\left(\frac{1}{n}\right)\right).$$

ckermann's function and inverse:

$$= n\lg n \cancel{- n\lg e} - (m\lg m \cancel{- m\lg e})$$

$$\lg(n!) = n\lg\left(\frac{n}{e}\right) \pm O(\lg n)$$

$$- ((n-m)\lg(n-n) \cancel{- (n-m)\lg e})$$

$$\pm O(\log n)$$

$$= n\underbrace{\left(\frac{m}{n}\lg\left(\frac{n}{m}\right) + \frac{m-n}{n}\lg\left(\frac{n}{m-n}\right)\right)} \pm O(\lg n)$$

$$p = \frac{m}{n} \qquad \frac{m-n}{n} = 1-p$$

space for strings:

exactly m 1s out of n

$\approx$ iid bits with $B(\frac{m}{n})$

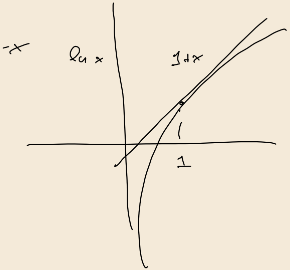
$$= n \cdot \underset{H(p)}{\underset{\shortparallel}{H\left(\frac{m}{n}, \frac{m-n}{n}\right)}} \pm O(\log n)$$

zeroth-order empirical entropy of B[0..n)

$$n\left(\frac{m}{n}\lg\left(\frac{n}{m}\right)+\underbrace{\frac{m-n}{n}\lg\left(\frac{n}{\underbrace{m-n}_{\approx 1}}\right)}_{\approx 0}\right)=m\lg\left(\frac{n}{m}\right)+\frac{n-m}{\ln 2}\ln\left(1+\frac{m}{n-m}\right)$$

$$\le m\lg\left(\frac{n}{m}\right)+\frac{\cancel{n-m}}{\underset{\underset{1.4427}{\|}}{\ln 2}}\cdot\frac{m}{\cancel{n-m}}$$

$\ln(x) \le 1-x$



Coding Tricks

Recall: bitvector w/ rank & select consisted of

indices mini level
indices micro level $\Big\}$ $O\left(n \frac{\lg\lg n}{\sqrt{\lg n}}\right)$ bits

lookup table $O(n^{1-\varepsilon})$

replace only that → bitvector itself n bits

↑

only needed to get access to $\Theta(\lg n)$ consecutive bits to access lookup table.

conceptually: B stored previously as ℓ-bit chunks $\ell = \frac{1}{2}\lg n$

instead of plain bits, store per chunk: (m_i, o_i)

m_i = # 1s in chunk

o_i = "offset" among $\binom{\ell}{m_i}$ possible chunks

m_i needs $\lg \ell$ bits

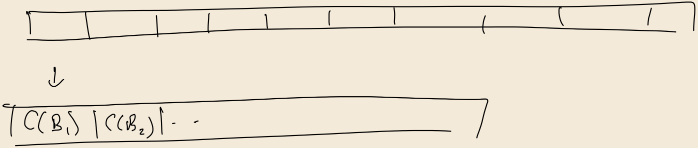
o_i needs $\lg \binom{\ell}{m_i} \approx m_i \lg\left(\frac{\ell}{m_i}\right)$

$\Rightarrow$ space for all chunks asymptotic to $\lg \binom{n}{m}$!

$\ell = 5 \quad m_i = 2$

0	00011
1	00101
2	01001
3	10001
4	00110
5	01010
6	10010
7	01100
8	10100
9	11000

Take any code $C : \{0,1\}^\ell \to \{0,1\}^*$ invertible



lookup table for inverse mapping C^{-1}

c_1	$C^{-1}(c_1)$
c_2	$C^{-1}(c_2)$
c_3	$\vdots$
c_s	

Succinct Compressed Bitvectors

Theorem 6.3

does not need B at query time

Let $B[0..n)$ be a bitvector containing m one bits. In $O(n)$ time we can compute an encoding data structure using

$$\lg \binom{n}{m} + O\left(\frac{n \lg\lg n}{\lg^{1/2} n}\right)$$

dominates if m too small

bits that supports the following operations in $O(1)$ time:

1. $B[i]$: return the bit at index i
2. $B.\mathtt{rank}_\alpha(i)$ for $\alpha \in \{0, 1\}$
3. $B.\mathtt{select}_\alpha(r)$ for $\alpha \in \{0, 1\}$

Predecessor Strikes Again

Theorem 5.11 (Predecessor Lower Bound)

Given a **static** set of $S \subseteq [0..u)$ of n integers with $\ell = \lg u$ bits each, represented in a data structure using $S \geq n\ell$ bits of space. Set $a = \lg(S/n)$. Answering `predecessor` (a.k.a. `floor`) queries costs

$$\Omega\left(\min\left\{\log_w(n),\ \lg\left(\frac{\ell-\lg n}{a}\right),\ \frac{\lg(\frac{\ell}{a})}{\lg\left(\frac{a}{\lg n}\cdot\lg(\frac{\ell}{a})\right)},\ \frac{\lg(\frac{\ell}{a})}{\lg\left(\lg(\frac{\ell}{a})/\lg(\frac{\lg n}{a})\right)}\right\}\right)$$

cell probes on a word-RAM with word size w. ◀

Pătrașcu, Thorup: *Time-Space Trade-Offs for Predecessor Search*, STOC **2006**

Remarks

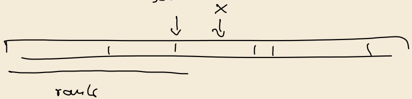
- Typical parameters:
  - space usage $S = n2^a$ is $\Theta(n)$ **words** of space ⇝ $a = \lg\lg n$ $a = \lg w$
  - $\ell \approx w = \lg u$
- ⇝ fusion trees optimal for huge w, i.e., when $\lg w = \Omega(\sqrt{\lg n}\lg\lg n)$
- ⇝ y-fast tries optimal for moderate w, i.e., any $w = O(\operatorname{polylog} n)$
- last two terms less intuitive . . .

Store a set $S \subseteq [u]$

$|S| = n$ as characteristic bitvector

$B[i] = [i \in S]$

$S.\text{pred}(x) = \text{select}(\text{rank}(x))$



① uncompressed bitvectors

$|B| = u$ $a = \lg\left(\frac{S}{n}\right) = \lg\left(\frac{u}{n}\right)$

l.b.: $\lg\left(\cancel{\frac{\lg u - \lg n}{\lg(\frac{u}{n})}}\right) = 0 = O(1)$ ✓

② compressed bitvectors

$m \ll n$

$\lg\binom{n}{m} \approx m\lg\left(\frac{n}{m}\right)$

$S = n\cdot\lg\left(\frac{u}{n}\right)$

$a = \lg\left(\lg\left(\frac{u}{n}\right)\right) \Rightarrow$ pred. l.b. $\geq \lg\left(\frac{\lg u - \lg n}{\lg\lg(\frac{u}{n})}\right) \sim \lg\lg\left(\frac{u}{n}\right)$

③ Largish regime for m $\underline{\omega\left(\frac{n}{\log n}\right) = m = o(n)}$

$\Rightarrow$ l.b. $= O(1)$

Overview: Monotone Minimal Perfect Hashing

- A function $h : S \to [m]$ is a *monoton minimal perfect hash function (MMPHF)* if
  - $h(x) \neq h(y)$ for all $x, y \in S$ (perfect)
  - $|S| = m$ (minimal)
  - $h(x) < h(y)$ iff $x < y$ for all $x, y \in S$ (monotone)

$\rightsquigarrow$ $h(x) = S.\texttt{rank}(x)$ for all $x \in S$

Note that $h(z)$ is completely arbitrary when $z \notin S$

Overview: Monotone Minimal Perfect Hashing

- A function $h : S \to [m]$ is a *monoton minimal perfect hash function (MMPHF)* if
  - $h(x) \neq h(y)$ for all $x, y \in S$ (perfect)
  - $|S| = m$ (minimal)
  - $h(x) < h(y)$ iff $x < y$ for all $x, y \in S$ (monotone)

⇝ $h(x) = S.\mathtt{rank}(x)$ for all $x \in S$

Note that $h(z)$ is completely arbitrary when $z \notin S$

- A compressed bitvector $B[0..u)$ with $m = |S|$ ones yields a MMPHF via $h(x) = B.\mathtt{rank}(x)$

Overview: Monotone Minimal Perfect Hashing

- A function $h : S \to [m]$ is a *monoton minimal perfect hash function (MMPHF)* if
  - $h(x) \neq h(y)$ for all $x, y \in S$ (perfect)
  - $|S| = m$ (minimal)
  - $h(x) < h(y)$ iff $x < y$ for all $x, y \in S$ (monotone)

$\rightsquigarrow$ $h(x) = S.\mathtt{rank}(x)$ for all $x \in S$

Note that $h(z)$ is completely arbitrary when $z \notin S$

only space efficient for m close to u

- A compressed bitvector $B[0..u)$ with $m = |S|$ ones yields a MMPHF via $h(x) = B.\mathtt{rank}(x)$
- Amazingly, MMPHF exist with $O(n \lg\lg\lg u)$ bits of space (using very different construction)

Belazzougui, Boldi, Pagh, Vigna: *Monotone Minimal Perfect Hashing: Searching a Sorted Table with O(1) Accesses*, SODA **2009**

Elias-Fano Encoding

$m << \frac{n}{\lg^c n}$ (can't get w.c. $O(1)$ query)

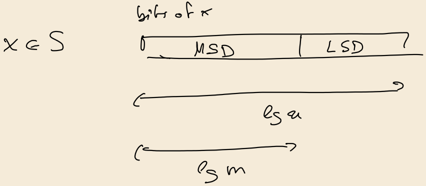
A practical heuristic for compressed bitvectors

Store $S \subseteq [0..u)$ with $|S| = m$ with $m \cdot \lg\left(\frac{u}{m}\right) + 2m + o(m)$ bits

and support

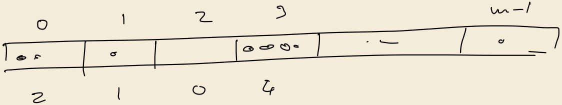
- selecting ith smallest elem. in S in $O(1)$
- rank/predecessor in w.c. $\min\left\{\lg\left(\frac{u}{m}\right), \lg m\right\}$ time

  and "average case" in $O(1)$ time



use MSD as "hash" of x

store MSD buckets as unary coded bucket counts



A 110 10 0 11110 ... 2m bits

add rank/select

store LSDs in sorted order in plain array

rank: bucket via rank on A

binary search in LSDs of bucket

$|\text{bucket}| \le m$

— " — $\le 2^{\lg(\frac{u}{m})} = \frac{u}{m}$

6.6 Sequences

Rank & Select on Sequences

Sequences = strings over Σ

- Represented as array $T[0..n)$

As for bitvectors, we want to support rank/select queries

#occurrences of c

- $\text{rank}_c(T[0..n), i) \;=\; \big|T[0..i)\big|_c$
  $=\;$ #c in first i characters of T
- $\text{select}_c(T[0..n), r)$
  $= \min\{j : |T[0..j]|_c \ge r\} \cup \{n\}$
  $=$ index of rth c in T, $(r = 1, 2, \ldots)$

▷ access T[i]

	0	1	2	3	4	5	6	7	8
$T[0..9)$	b	a	n	a	n	a	b	a	n

	0	1	2	3	4	5	6	7	8	9
$\text{rank}_{a}(T, i)$	0	0	1	1	2	2	3	3	4	4
$\text{rank}_{b}(T, i)$	0	1	1	1	1	1	1	2	2	2
$\text{rank}_{n}(T, i)$	0	0	0	1	1	2	2	2	2	3

$\text{select}_{a}(T, r)$	/	1	3	5	7	9	9	9	9	9
$\text{select}_{b}(T, r)$	/	0	6	9	9	9	9	9	9	9
$\text{select}_{n}(T, r)$	/	2	4	8	9	9	9	9	9	9

Rank & Select on Sequences

Sequences = strings over Σ

- Represented as array $T[0..n)$

	0	1	2	3	4	5	6	7	8
$T[0..9)$	b	a	n	a	n	a	b	a	n

As for bitvectors, we want to support rank/select queries

#occurrences of c

- $\text{rank}_c(T[0..n), i) \;=\; \big|T[0..i)\big|_c$
  $= \;\; \#c$ in first i characters of T
- $\text{select}_c(T[0..n), r)$
  $= \;\; \min\big\{j : |T[0..j]|_c \geq r\big\} \cup \{n\}$
  $= \;\;$ index of rth c in T, $(r = 1, 2, \ldots)$

	0	1	2	3	4	5	6	7	8	9
$\text{rank}_a(T, i)$	0	0	1	1	2	2	3	3	4	4
$\text{rank}_b(T, i)$	0	1	1	1	1	1	1	2	2	2
$\text{rank}_n(T, i)$	0	0	0	1	1	2	2	2	2	3

$\text{select}_a(T, r)$	/	1	3	5	7	9	9	9	9	9
$\text{select}_b(T, r)$	/	0	6	9	9	9	9	9	9	9
$\text{select}_n(T, r)$	/	2	4	8	9	9	9	9	9	9

Rank/select on Sequences again a versatile tool

- bioinformatics: compressed text indexes, e. g., FM Index
- Representing graphs

Wavelet Trees

The **Wavelet Trees** for a $T \in [0..n)$ over $\Sigma = [0..\sigma)$

- supports access to $T[i]$ in $O(\log \sigma)$ time,
- $\mathrm{rank}_c(T, i)$ and $\mathrm{select}_c(T, r)$ in $O(\log \sigma)$ time, and
- occupies $\sim n \lg \sigma$ *bits* of space.

$n \lg \sigma + o(n \lg \sigma)$

Compressed Wavelet Trees

using compressed bitvectors for wavelet tree nodes

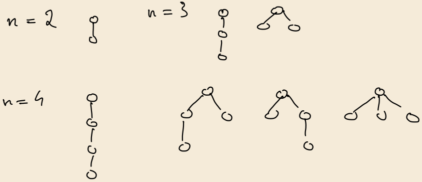
$n \cdot H_0(T)\,(1+o(1))$

- concatenate bitvectors for better l.o.t.
- compressed repr. picks up local low entropy

6.7 Trees

Ordinal Trees

- rooted tree with unbounded degrees
- children are ordered (left to right)
- store only shape of tree (no node labels)
  (store labels separately!)

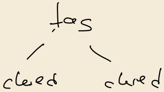


How many trees w/ n nodes?

Example uses

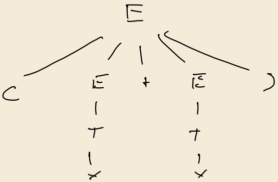
- structure of XML documents
- MSTs for routing
- representing parse trees of grammas (grammar-based compression)

<tag> <child> .. </child> <child> .. </child> </tag>

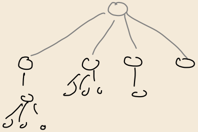


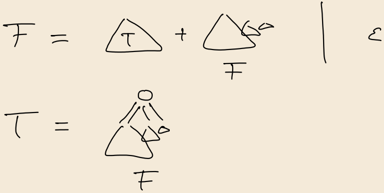
$E \to (E+E) \mid (E \bullet E) \mid T$

$T \to x$



count forests on n nodes





C_n = # forests w/ n nodes

$$C_n = \sum_{k=1}^{n} C_{k-1} \cdot C_{n-k}$$

$\uparrow$ # choices for first tree

$C_1 = 1$

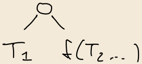
$$C_n = \frac{1}{n+1}\binom{2n}{n}$$

$$C_2 = \frac{1}{3}\binom{4}{2} = \frac{6}{3} = 2$$

C_n = # binary trees on n nodes

$T_1, T_2 \ldots$

$F \to$



Lower Bound

- **Fact:** The number of ordinal trees on $n + 1$ nodes is $C_n = \frac{1}{n+1}\binom{2n}{n}$ $\approx 4^n / n^{3/2}$

Lower Bound

- **Fact:** The number of ordinal trees on $n+1$ nodes is $C_n = \frac{1}{n+1}\binom{2n}{n}$

⇝ To represent an ordinal tree, we need $\lg C_n \sim 2n$ bits of space

2 bits per node!

Typical pointer-based representation uses $\Omega(\log n)$ bits per node

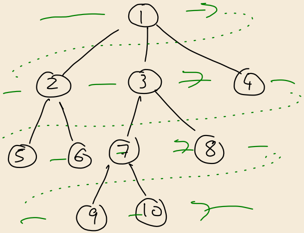
Operations on Trees

Operation	Meaning
`root()`	The root node
`fchild`(v)	The first child of node v, if it exists
`lchild`(v)	The last child of node v, if it exists
`nsibling`(v)	The next sibling of node v, if it exists
`psibling`(v)	The previous sibling of node v, if it exists
`parent`(v)	The parent of node v, if it exists
`isleaf`(v)	Whether v is a leaf node
`nodemap`(v)	A unique identifier in $[1, n]$ for v, called its *index*
`nodeselect`(i)	The node v with index i
`depth`(v)	Depth of node v; the depth of the root is 1
`height`(v)	Height of node v; the height of a leaf is 0
`deepestnode`(v)	A deepest node in the subtree of node v
`subtree`(v)	Number of nodes in the subtree of v, counting v
`isancestor`(u, v)	Whether u is an ancestor of v (or v itself)
`levelancestor`(v, d)	The ancestor at distance d from v ($d = 1$ is `parent`(v))
`preorder`(v)	Preorder value of v (first v, then each subtree)
`preorderselect`(i)	Node v with preorder i
`postorder`(v)	Postorder value of v (first each subtree, then v)
`postorderselect`(i)	Node v with postorder i
`children`(v)	Number of children of node v
`child`(v, t)	The tth child of node v, if it exists
`childrank`(v)	The t such that node v is the tth child of its parent
`lca`(u, v)	The lowest common ancestor of nodes u and v
`leafnum`(v)	The number of leaves in the subtree of node v
`leafrank`(v)	The number of leaves to the left of node v, plus 1
`leafselect`(i)	The ith left-to-right leaf node in the tree
`childrenlabeled`(v, l)	Number of children of node v via edges labeled l
`labeledchild`(v, l, t)	The tth left-to-right child of v by an edge labeled l
`childlabel`(v)	The label of the edge that leads to node v

Tree Covering based data structure can support all of these in $O(1)$ time and using $2n + o(n)$ bits

LOUDS = level-order unary degree sequence

simplest tree data structure w/ 2n bits, supports local navigation in $O(1)$



$$L \quad \underbrace{\overset{0}{1}\overset{1}{1}\overset{2}{1}\overset{3}{0}}_{1}\ \underbrace{\overset{4}{1}\overset{5}{1}\overset{6}{0}}_{2}\ \underbrace{\overset{7}{1}\overset{8}{1}\overset{9}{0}}_{3}\ \underbrace{\overset{10}{0}}_{4}\ \underbrace{\overset{11}{0}}_{5}\ \underbrace{\overset{12}{0}}_{6}\ \underbrace{\overset{13}{1}\overset{14}{1}\overset{15}{0}}_{7}\ \underbrace{\overset{16}{0}}_{8}\ \underbrace{\overset{17}{0}}_{9}\ \underbrace{\overset{18}{0}}_{10}$$

parent(v)

nodes $\overset{!}{=}$ starting position of their UDS

⑤ = 11

parent(⑧)

parent(⑥) = parent(12)

$\shortparallel$ ② $\shortparallel$ 4

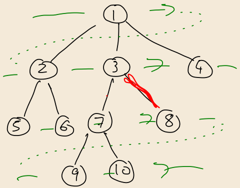
nodemap(v) = $L.\text{rank}_0(v) + 1$

nodemap(11) $\overset{!}{=}$ ⑤

nodeselect(ⓘ) = $L.\text{select}(i-1) + 1$

$L.\text{prev}_0(i) = L.\text{select}_0(L.\text{rank}_0(i))$

```
parent(v):
    $i := L.\text{rank}_0(v) + 1$
    $p := L.\text{select}_1(i-1)$
    $w := L.\text{prev}_0(p) + 1$
    return w.
```



$$L \quad \underbrace{\overset{0}{1}\overset{1}{1}\overset{2}{1}\overset{3}{0}}_{1}\ \underbrace{\overset{4}{1}\overset{5}{1}\overset{6}{0}}_{2}\ \underbrace{\overset{7}{1}\overset{8}{1}\overset{9}{0}}_{3}\ \underbrace{\overset{10}{0}}_{4}\ \Big|\ \underbrace{\overset{11}{0}}_{5}\ \underbrace{\overset{12}{0}}_{6}\ \underbrace{\overset{13}{1}\overset{14}{1}\overset{15}{0}}_{7}\ \underbrace{\overset{16}{0}}_{8}\ \underbrace{\overset{17}{0}}_{9}\ \underbrace{\overset{18}{0}}_{1}$$

In LOUDS: ith 1 and (i+1)st 0 bits form a parent-child edge

We can similarly build

fchild, lchild, child(v,t)

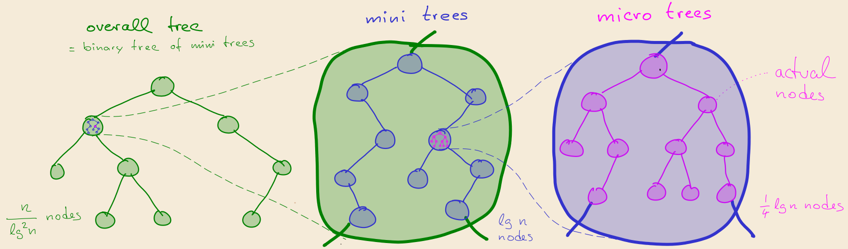
children(v) (degree)

Not clear how to support, e.g. subtree size

depth

...

Tree Covering



same idea as for bitvectors!

can store $O(\log n)$ bits per mini tree

for each micro tree can store $O(\log\log n)$ bits

exhaustive tabulation for operations inside micro trees

subtree-size(v)

depth(v): ✓

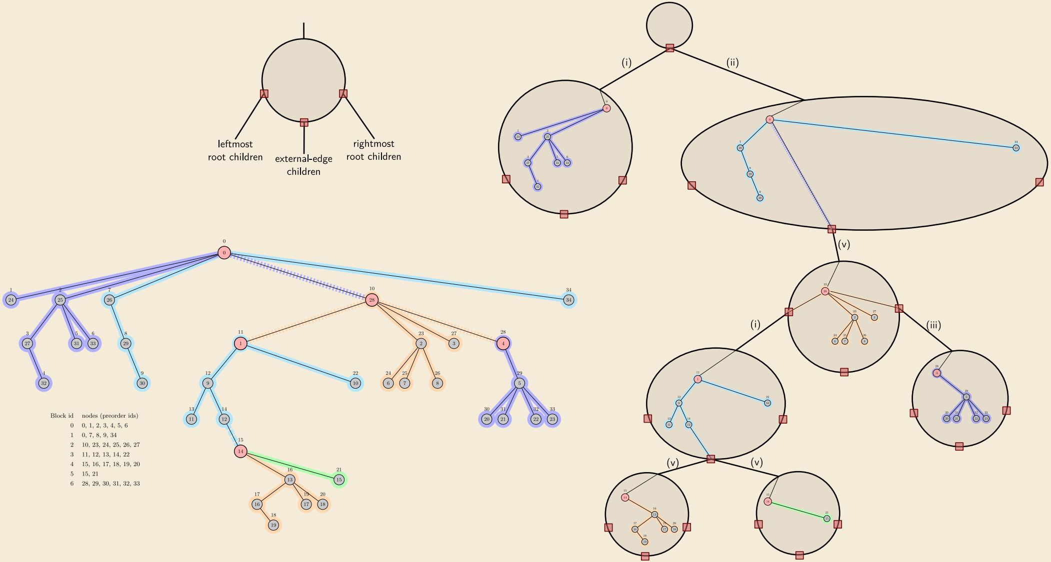
store depths of

mini tree roots (absolute)

micro tree roots (relative to mini)

within depth in table

TC: Farzan-Munro Decomposition



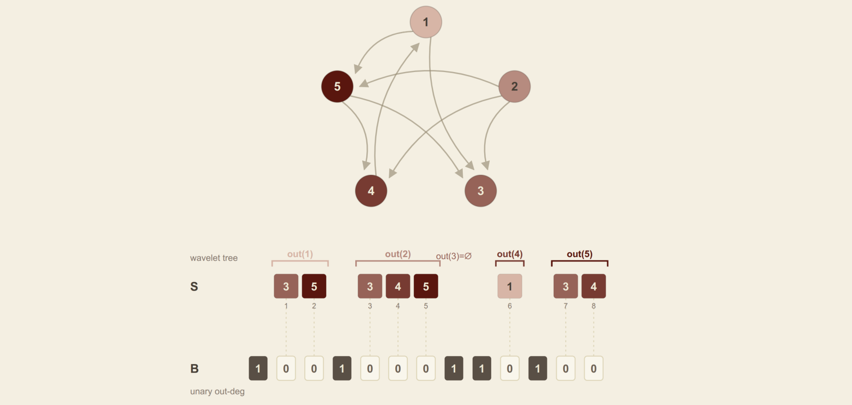
Block id	nodes (preorder ids)
0	0, 1, 2, 3, 4, 5, 6
1	0, 7, 8, 9, 34
2	10, 23, 24, 25, 26, 27
3	11, 12, 13, 14, 22
4	15, 16, 17, 18, 19, 20
5	15, 21
6	28, 29, 30, 31, 32, 33

6.8 Graphs

Adjacency Strings

A Directed Graph as a Wavelet Tree

The wavelet tree holds `S`, all out-neighbourhoods concatenated; the bitvector `B` holds the out-degrees in unary. Together they answer out- and in-neighbour queries symmetrically.



S		3	5		3	4	5			1		3	4
		1	2		3	4	5			6		7	8
B	1	0	0	1	0	0	0	1	1	0	1	0	0

Store graph as
- wavelet tree over concatenated out neighborhoods
- bitvector unary outdegree sequence

⇒ in- and out-going edges $O(\log n)$ time

space $m \log n + O(n)$ bits

(standard adjacency list uses $\Theta(m \log n + n \log n)$ bits