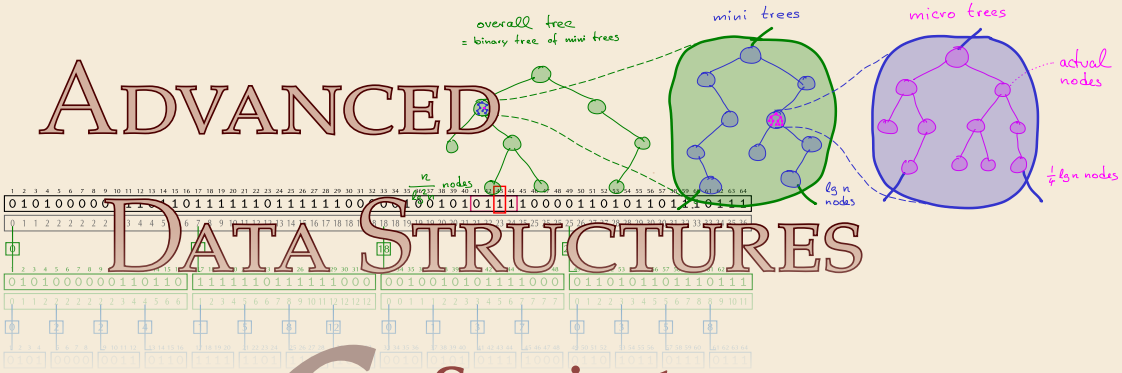


Advanced Data Structures

6 Succinct Data Structures

Advanced Data Structures · Summer 2026

Prof. Dr. Sebastian Wild

CS650 (Summer 2026)
Philipps-Universität Marburg
version **2026-06-29 16:06**

Outline

6 Succinct Data Structures

6.1 A Motivating Example
6.2 Bitvectors
6.3 Supporting Rank
6.4 Supporting Select
6.5 Compressed Bitvectors
6.6 Sequences
6.7 Trees
6.8 Graphs

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

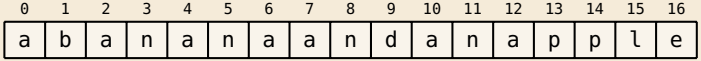
- traditionally:
  - compression for **archiving** data, minimize size of representation
  - computation/analysis: minimize time; use extra data structures
  - ⇝ always decompress data first!
  - reaches limits of fast memory for large datasets

- 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

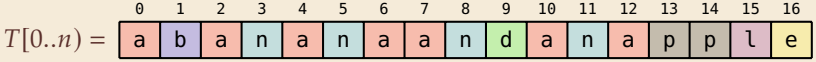
$n = |\mathcal{U}|$ $\quad$ $\lg n$ bits needed to distinguish objects in $\mathcal{U}$

Computing over compressed data

$T[0..n) =$



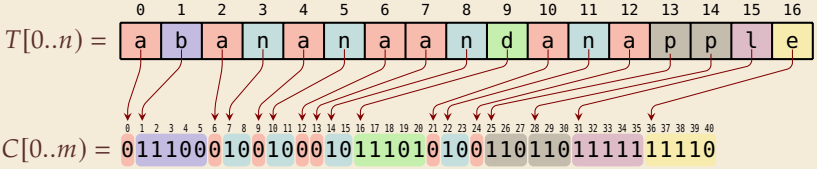
Computing over compressed data



Huffman Code

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

Computing over compressed data



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

c 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

$C[0..m) =$

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

Huffman Code

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

- 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

Huffman Code

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

$C[0..m) =$

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

- 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!

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

$C[0..m) =$ 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?

- 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)

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

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<byte>` (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<byte>`
  - 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<byte>`
  - 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

⇝ in practice: `sdsl::bit_vector` (based on packing bits to words)

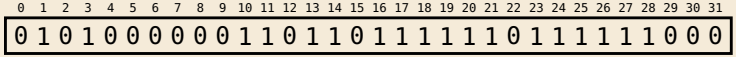
- from external library SDSL Lite
- `https://github.com/simongog/sdsl-lite`

Rank & Select on Bitvectors

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

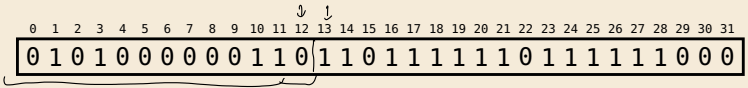
Rank & Select on Bitvectors

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



Rank & Select on Bitvectors

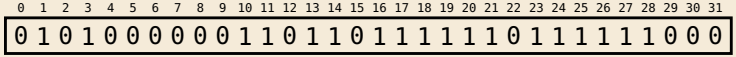
- $B[0..n)$ static array of n bits (Boolean array).
  - easy to store using n bits of space
- $B.\texttt{rank}(i) = \#\,1\text{s in } B[0..i)$ (first $i \in [0..n]$ positions) (=prefix sum)
- $B.\texttt{select}_1(r) = \min\{j : B.\texttt{rank}(j+1) \geq r\} \cup \{n\}$
  $= \text{index of } r\text{th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



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

Rank & Select on Bitvectors

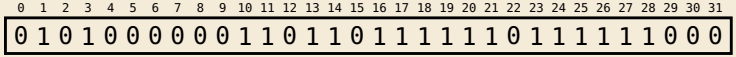
- $B[0..n)$ static array of n bits (Boolean array).
  - easy to store using n bits of space
- $B.\texttt{rank}(i) = \#\,1\text{s in } B[0..i)$ (first $i \in [0..n]$ positions) (=prefix sum)
- $B.\texttt{select}_1(r) = \min\{j : B.\texttt{rank}(j+1) \geq r\} \cup \{n\}$
  $= \text{index of } r\text{th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



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

Rank & Select on Bitvectors

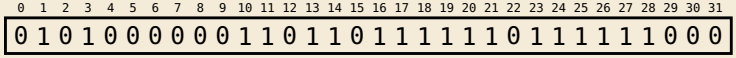
- $B[0..n)$ static array of n bits (Boolean array).
  - easy to store using n bits of space
- $B.\texttt{rank}(i) = \#\,1\text{s in } B[0..i)$ (first $i \in [0..n]$ positions) (=prefix sum)
- $B.\texttt{select}_1(r) = \min\{j : B.\texttt{rank}(j+1) \geq r\} \cup \{n\}$
  $= \text{index of } r\text{th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



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

Rank & Select on Bitvectors

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

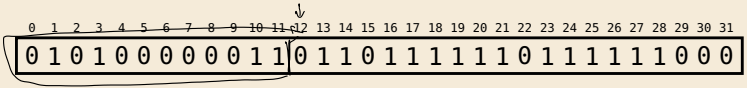


- $\texttt{rank}(12) = \texttt{rank}(13) = 4$
- $\texttt{select}_1(3) = 10$
- $\texttt{select}_1(4) = 11$

$\rightsquigarrow$ $B[i] = B.\texttt{rank}(i+1) - B.\texttt{rank}(i)$ (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
- $B.\texttt{rank}(i) = \#\,1\text{s in } B[0..i)$ (first $i \in [0..n]$ positions) (=prefix sum)
- $B.\texttt{select}_1(r) = \min\{j : B.\texttt{rank}(j+1) \geq r\} \cup \{n\}$
  $= \text{index of } r\text{th } 1 \text{ in } B, (r = 1, 2, \ldots)$



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

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

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

Versatile Bitvectors

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

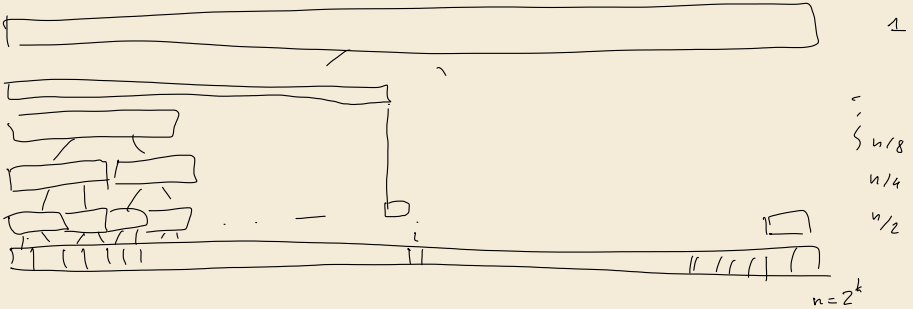
6.3 Supporting Rank

Rank – Simple Ideas

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

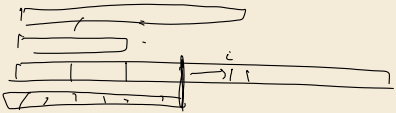
Rank – Simple Ideas

- can answer rank by linear scan . . . $O(n)$ time
- can maintain a ***segment tree*** with subrange sums
  - $\rightsquigarrow$ $O(\lg n)$ 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



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!)



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.

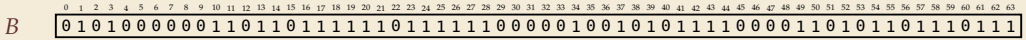
Rank – Simple Ideas

- can answer rank by linear scan ... $O(n)$ time
- can maintain a ***segment tree*** with subrange sums
  - $\rightsquigarrow$ $O(\lg n)$ 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.

How can we do better?

Rank Index for Bitvector

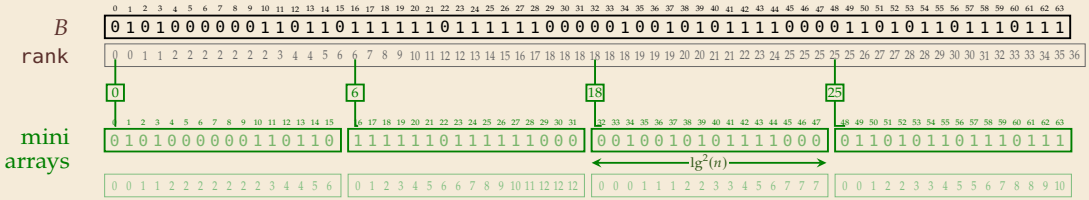
- Apart from B, we store:



Rank Index for Bitvector

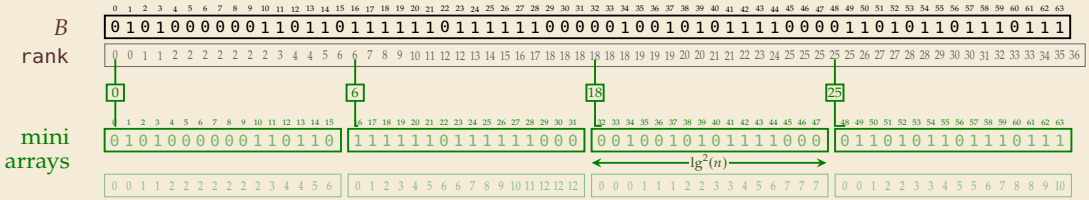
- Apart from B, we store:
  1. Rank of first pos. of each **mini array** of $L = \lg^2(n)$ bits

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



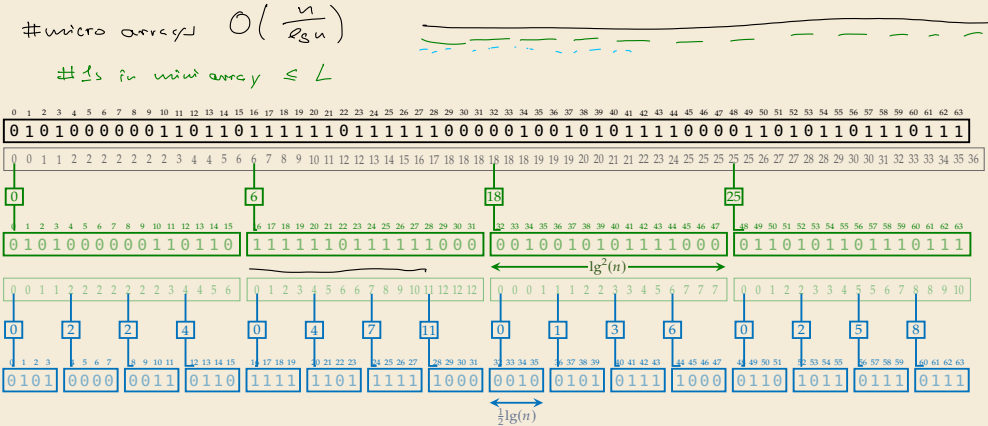
Rank Index for Bitvector

- 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



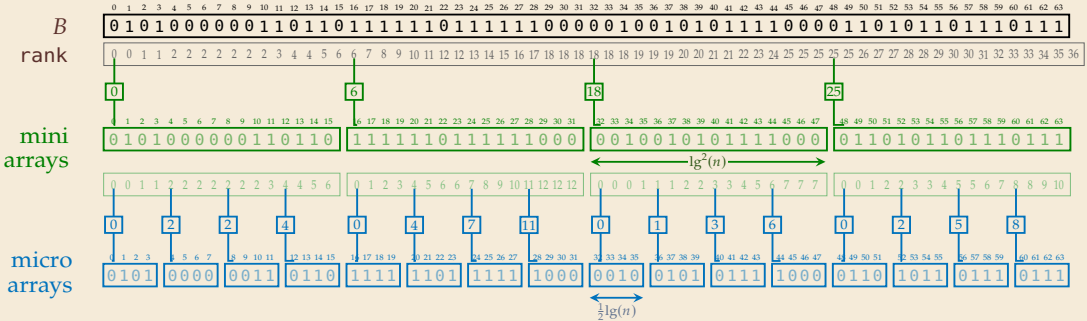
Rank Index for Bitvector

- 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



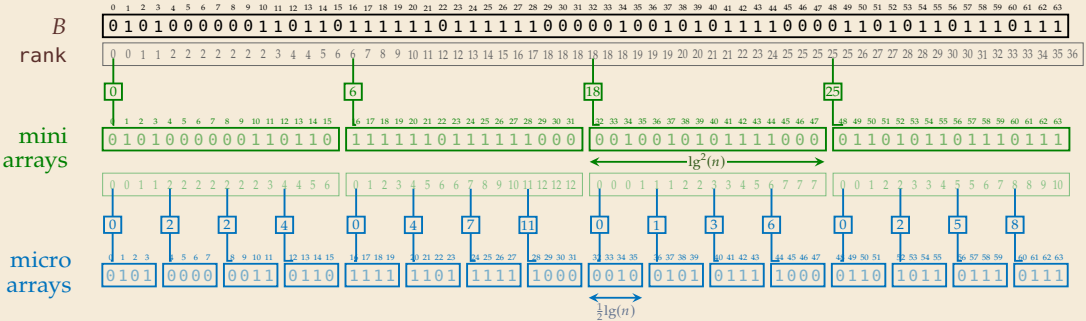
Rank Index for Bitvector

- 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

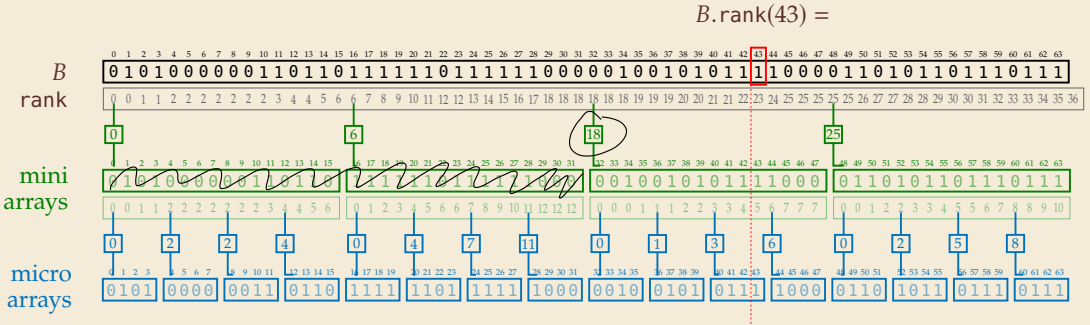


Rank Index for Bitvector

- 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

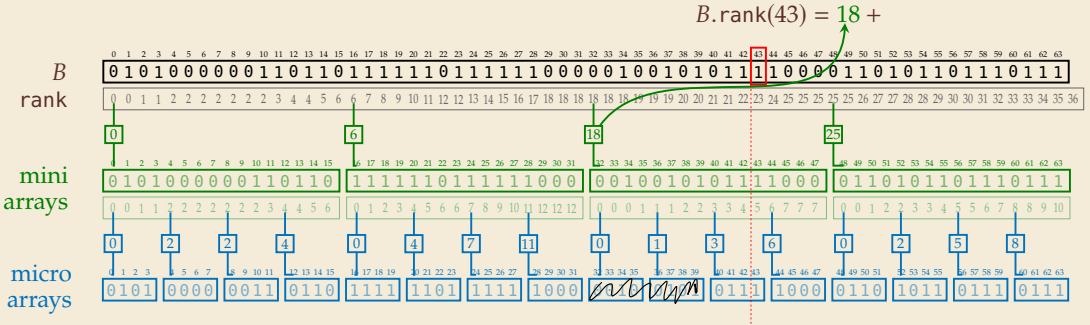


How to find rank



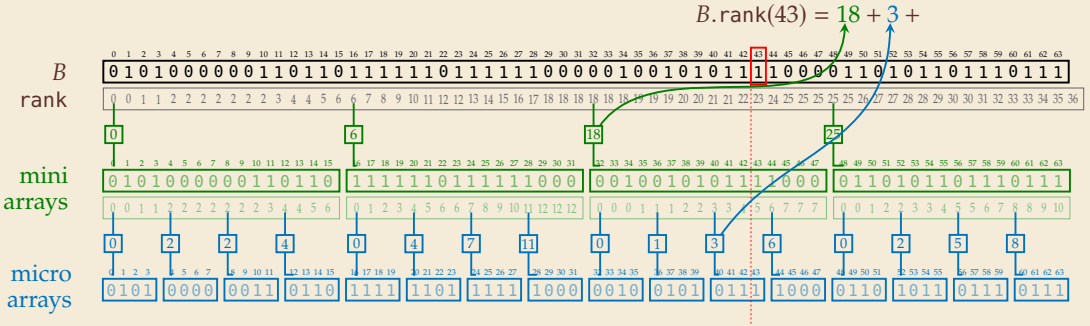
- How to compute $B.\mathtt{rank}(i)$?

How to find rank



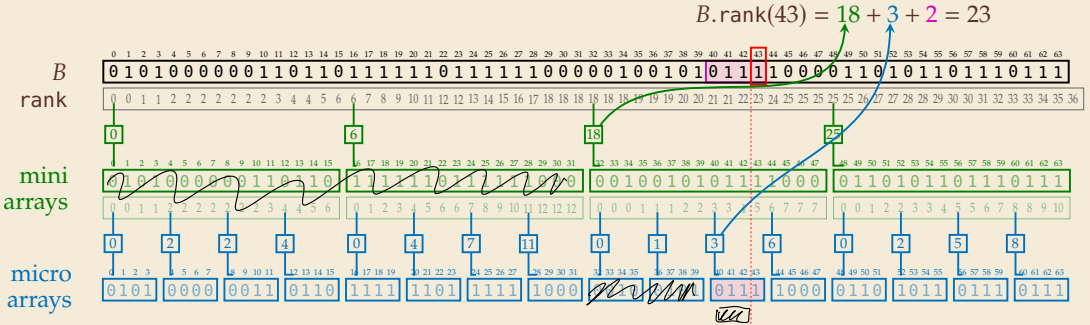
- How to compute $B.\texttt{rank}(i)$?
  - find rank up to element's mini array

How to find rank



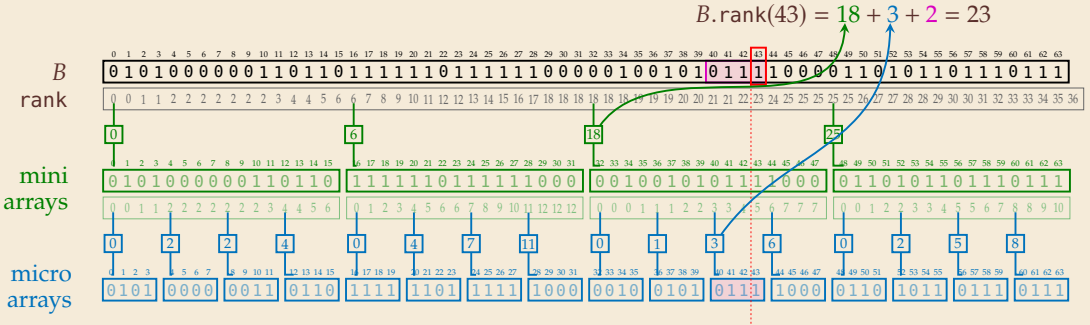
- How to compute $B.\texttt{rank}(i)$?
  - find rank up to element's mini array
  - add rank up to element's micro array (mini-array local)

How to find rank



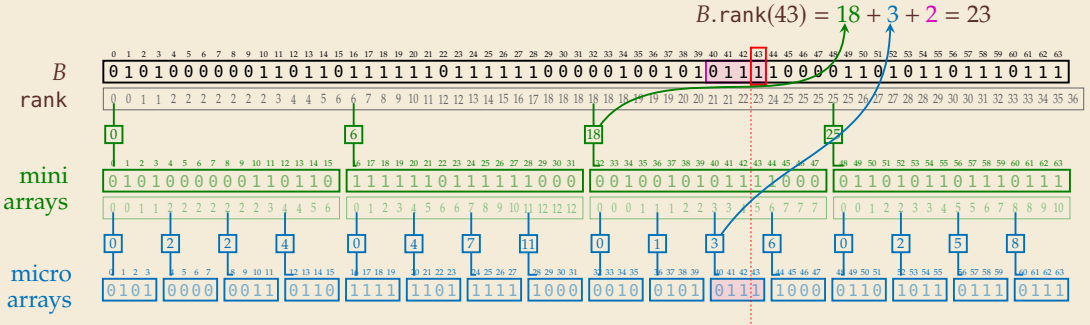
- How to compute $B.\mathtt{rank}(i)$?
  - find rank up to element's mini array
  - add rank up to element's micro array (mini-array local)
  - add micro-array-local rank of position

How to find rank



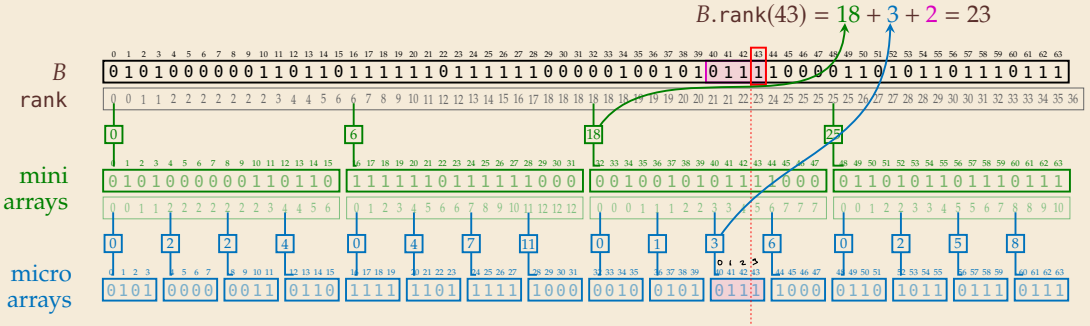
- How to compute $B.\mathtt{rank}(i)$?
  - find rank up to element's mini array
  - add rank up to element's micro array (mini-array local)
  - add micro-array-local rank of position
    - can either do this naively by scanning $\frac{1}{2}\lg n$ bits $\rightsquigarrow$ $O(\log n)$ time

How to find rank



- 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
    - 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



- 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
    - 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

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

⇝ $O(1)$ 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.

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

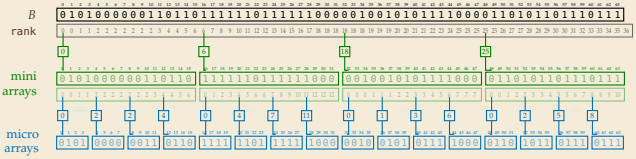
Back to Rank!

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

Preprocessing: 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: $\#\text{micro arrays} = \frac{n}{\frac{1}{2}\lg n} \gg$

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

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

Back to Rank!

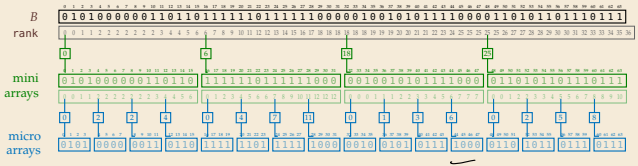
Recall: Our task is to compute the local rank in a micro array of $\ell = \frac{1}{2}\lg n$ bits.

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



- Note: $\#\text{micro arrays} = \frac{n}{\frac{1}{2}\lg n} \gg$ $\#\textit{different}$ 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 $\mu R[t][i]$

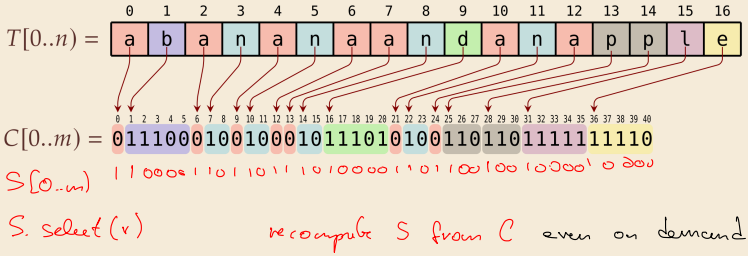
Succinct Rank – Result

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 $B.\texttt{rank}(i)$ 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.



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 ≠ Russian; Dinitz emigrated to Isreal

  (Arlazarov and Kronrod are Russian)

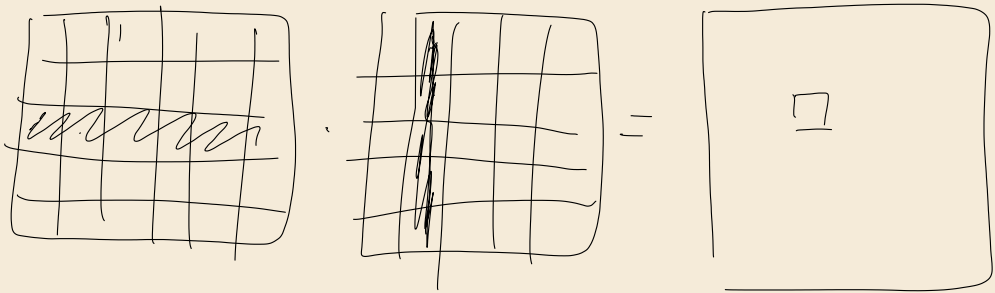
- American authors coined the othering term "Method of Four Russians"
  . . . name in widespread use

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*.

$\rightsquigarrow$ 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!

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 $a_{1,1} \dots a_{\ell,\ell} b_{1,1} \dots b_{\ell,\ell}$, 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 $a_{1,1} \dots a_{\ell,\ell} b_{1,1} \dots b_{\ell,\ell}$, 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*.

$\rightsquigarrow$ 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 $O(n^3/\ell^3) = O(n^3/\log^{3/2} n)$.

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 $O(n^3/\log^2 n)$.

6.4 Supporting Select

Select Indices – Simple Ideas

Recall: $B.\texttt{select}_1(r) = \min\{j : B.\texttt{rank}(j+1) \geq r\} \cup \{n\}$
$= \text{index of } r\text{th } 1 \text{ in } B, (r = 1, 2, \ldots)$

Select Indices – Simple Ideas

Recall: $B.\texttt{select}_1(r) = \min\{j : B.\texttt{rank}(j+1) \geq r\} \cup \{n\}$
$= \text{index of } \overbrace{r\text{th } 1 \text{ in } B}, (r = 1, 2, \ldots)$

$\rightsquigarrow$ 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

Select Indices – Simple Ideas

Recall: $B.\texttt{select}_1(r) = \min\{j : B.\texttt{rank}(j+1) \geq r\} \cup \{n\}$
$= \text{index of } r\text{th } \texttt{1} \text{ in } B, (r = 1, 2, \ldots)$

$\rightsquigarrow$ 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

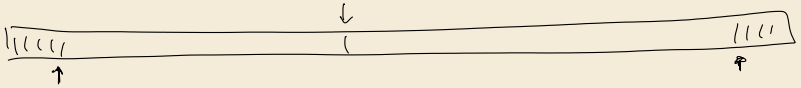
Select Indices – Simple Ideas

Recall: $B.\texttt{select}_1(r) = \min\{j : B.\texttt{rank}(j+1) \geq r\} \cup \{n\}$
$= \text{index of } r\text{th } \texttt{1} \text{ in } B, (r = 1, 2, \ldots)$

$\rightsquigarrow$ 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



Select Indices – Simple Ideas

Recall: $B.\texttt{select}_1(r) = \min\{j : B.\texttt{rank}(j+1) \geq r\} \cup \{n\}$
$= $ 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 1s $\rightsquigarrow$ $m \lg n$ 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

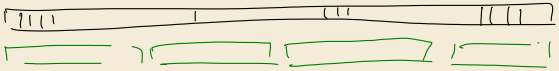
$O\left(n \frac{\lg \lg n}{\sqrt{\lg n}}\right)$

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 $O\left(n \frac{(\lg \lg n)^2}{\lg n}\right) = o(n)$ bits of space that can answer $B.\texttt{select}(r)$ queries in $O(1)$.

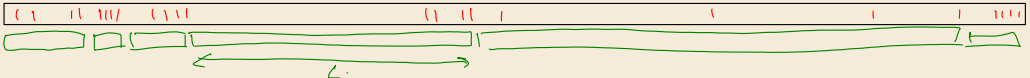
The index requires oracle access to B at query time, i. e., in $O(1)$ time we must be able to read $\Theta(\log n)$ consecutive bits of B. ◀

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

$K = \lg^2 n$

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

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

Case distinction

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

  $\Rightarrow$ have $\leq \frac{n}{\lg^4(n)}$ such long miniblocks each w/ k ones

  $\Rightarrow$ 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 $(L_j \leq \lg^4(n))$ store starting index of mini block

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

  recursively subdivide into micro blocks w/ k ones

  $k = \sqrt{\lg n}$

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

  second-level case distinction

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

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

  $\Rightarrow$ write down indices of ones

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

- dense micro block — 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