# L5: Rank & Select Data Structures

**Problems defined:**

initial input: $a =$ (one byte)

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 |

↪ a bit array (conceptual) * typically packed as int's array. (most typical in practice)

Maintain a data structure to support:

1. rank (i) : count the # of 1's up to position i
2. select (y) : find the pos (i.e. index) of the y-th 1.

Concrete Ex: $b = [0, 1, 0, 1, 0]$ rank(2) => 1, select(1) = 1

Goals: fast & compact < theory land / in practice.

**Focus on rank for now:**

- Strawman soln #1: What should we keep to allow answering rank quickly?
  - → naively count: $\Theta(n)$ time
  - → Keep an aux array of size n, storing the "answers".
  - → Note: prefix sum array $b\_aux = [0, 0, 1, 1, 2]$
  - → Space (in bits): $n \times$ (size of a word) $= O(n \times \log n)$ in bits — theory land
    ↪ $n \cdot 64$ in practice (64-bit machine)
- Can we do better?

L0

| 0 | 5 | 11 | 14 | 19 | 24 |
|---|---|---|---|---|---|
| 11011100 | 10111011 | 11000100 | 11010101 | 11100110 | 11110100 |

L1

O(log n)-bit numbers

→ split into blocks of b bits each

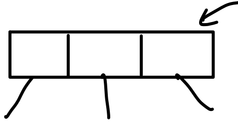
→ store prefix sums at the start of each block

→ To answer rank (i) < read L0 p.s. / count 1's in that block upto i.

space: $\left(\frac{n}{b}\right) \cdot \log n$ bits

time: $O(b)$.

- Optimization:



log n bits each (for top level)

Use strawman instead of counting → (log b)-sized words × b slots

space: $\left(\frac{n}{b}\right) \log n + \left(\frac{n}{b}\right) \cdot b \log b$ (L0)

$= \frac{n}{b} \log n + n \cdot \log b$

Using $b = \log n \Rightarrow O(n \log \log n)$ space & $O(1)$ time

- Remarks: Recursively use this idea
  $\Rightarrow n \log^* n$ space & $\log^* n$ time.

## New idea #1: The Four Russians Strategy.

→ Suppose the block size is small, say $b=3$. There are only $2^b$ variations

→ Build a table & look it up    count(blk, index) → 0..b

| | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
|---|---|---|---|---|---|---|---|---|
| *Index* 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| *Index* 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| *Index* 2 | 0 | 0 | 1 | 1 | 1 | 1 | 2 | 2 |
| *Index* 3 | 0 | 1 | 1 | 2 | 1 | 2 | 2 | 3 |

Table space usage:

$O(\underbrace{2^b}_{\text{variations}} \times \underbrace{b}_{\text{index pos}} \times \underbrace{\log b}_{\text{output size}})$

Using $b = \frac{1}{2}\cdot\log_2 n \Rightarrow O(n^{1/2}\cdot \lg n \cdot \lg\lg n)$
$o(n^{1/2+\varepsilon})$

How?

L0: prefix sum @ block level

| 0 | 5 | 11 | 14 | 19 | 24 |
|---|---|---|---|---|---|
| 11011100 | 10111011 | 11000100 | 11010101 | 11100110 | 11110100 |

$\frac{1}{2}\lg n = b$ bits each

+ 4Rus table

Space: $\frac{n}{b}\cdot\log n + o(n^{1/2+\varepsilon})$

$= \frac{n}{\frac{1}{2}\log n}\cdot\log n + o(n^{1/2+\varepsilon})$

$= O(n)$.

Time: $O(1)$.

**Reality** → large bit arrays → overall perf. ~ cache misses

→ pipelining ⇒ independent ops are cheap ($\geq 4$ at a time on recent procs)

→ Optimize for cache misses ≫ branches ≫ arith./logic ops
100 ns    5 ns    $< \frac{1}{4}$ ns

32-bits / 64 bits reg.

→ popcnt (x86) ~ 1 cycle, only using 1 "port"
Nehalem + later ↗ (4 cycles latency)

popcnt-ing $10^8$ blocks →

| Size (bits) | Time (seconds) | # of cache misses |
|---|---|---|
| 64 | 0.13 | 1 |
| 128 | 0.19 | 1 |
| 256 | 0.30 | 1 |
| 512 | 0.50 | 1 |
| 1024 | 0.99 | 2 |
| 2048 | 2.01 | 4 |

[poppy] (Zhou - Andersen - Kaminsky '2013)

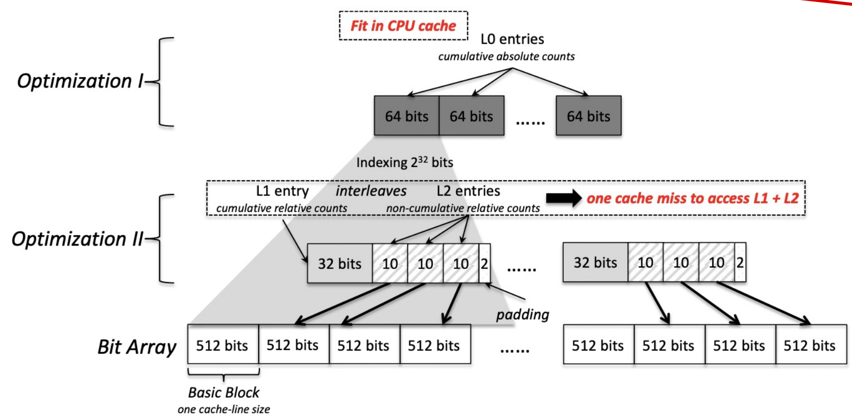
popcnt-ing 64M array 300×

4Rus (above)

| Method | Time (ms) |
|---|---|
| Precomputed table (byte-wise) | 729.0 |
| `popcnt` instruction | 191.7 |
| SSE2 | 336.0 |
| SSSE3 | 237.7 |
| Broadword programming | 798.9 |

SIMD

sysbill style




- Block: 512 bits (one cache line on their mach.)
- L0: $2^{32}$ bits

L1: prefix sum trick (w: 32 bits)

L2: count of 1s in block

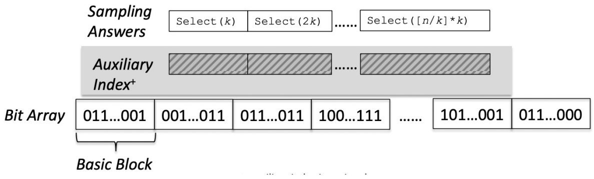
- L0 + L1 parallel lookups

# Select

→ Could binary search (using fast "rank") – $O(\log n)$ time.

· How to "cache" useful info?

→ general trick:



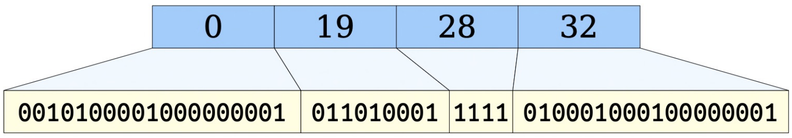

to answer select(y)

→ ask select$\left(\lfloor y/k \rfloor \cdot k\right)$

→ linear scan from that point on.

Optimization: Note: some top-level pieces have a big gap

→ write down the positions of 1's in aux.




$\frac{n}{c} \cdot \log n$ space : $c^2$ time

→ Extension of 4Rus: $O(n\sqrt{\log n})$ space & $O(1)$ time

→ Poppy: see paper

→ Open(?): Practical select in small const $O(n)$ space & $O(1)$ time.

Activity: Write code to count the # of 1's in a randomly generated array of len 256M

1. Count bits directly
2. Use popcnt

Compare the differences.