Succinct Data Structures

Part One

Motivation

- Some data sets are downright gigantic.
  - The human genome uses 3 billion base pairs.
  - Google gets billions of search queries a day.
  - Census data for some countries runs to billions of entries.
- Simply loading the data sets into memory – let alone storing them in fancy data structures – pushes up on system limits.
- ***Goal:*** Store our data using as few bits as possible while still being able to answer interesting questions about that data.

Outline for Today

- ***The Binary Rank Problem***
  - Prefix sums on bitvectors.
- ***Solving Binary Rank***
  - And learning about how to save bits along the way.
- ***Jacobson's Succinct Rank Structure***
  - A surprisingly space-efficient data structure for binary ranking.

Binary Ranking

Binary Ranking

- The ***binary ranking problem*** is the following:

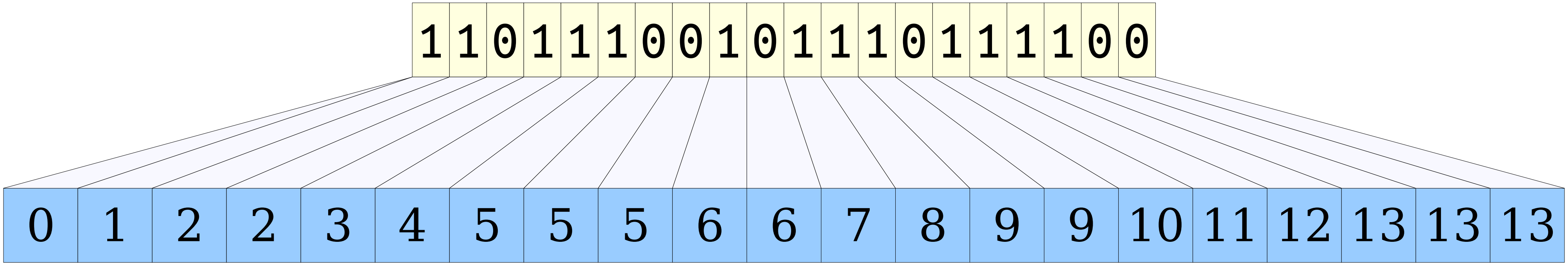
Given a list of n bits and an index i, return the sum of all the bits up to position i in the list.

- It's basically the problem of computing prefix sums in bitvectors.

1	1	0	1	1	1	0	0	1	0	1	1	1	0	1	1	1	1	0	0

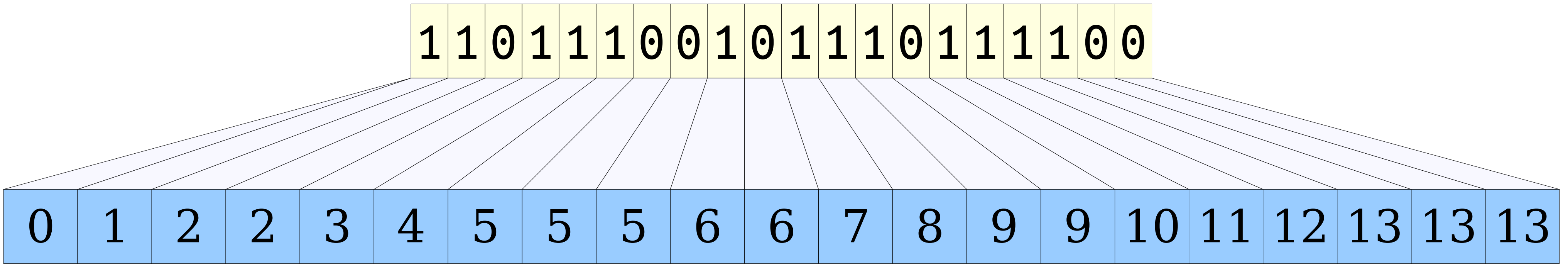
Binary Ranking

- Let's imagine we want to be able to answer rank queries in time O(1).
- We could do this by writing down the prefix sums for all positions in an array, then just looking up the answer in a table.
- ***Question:*** How much space does this use?



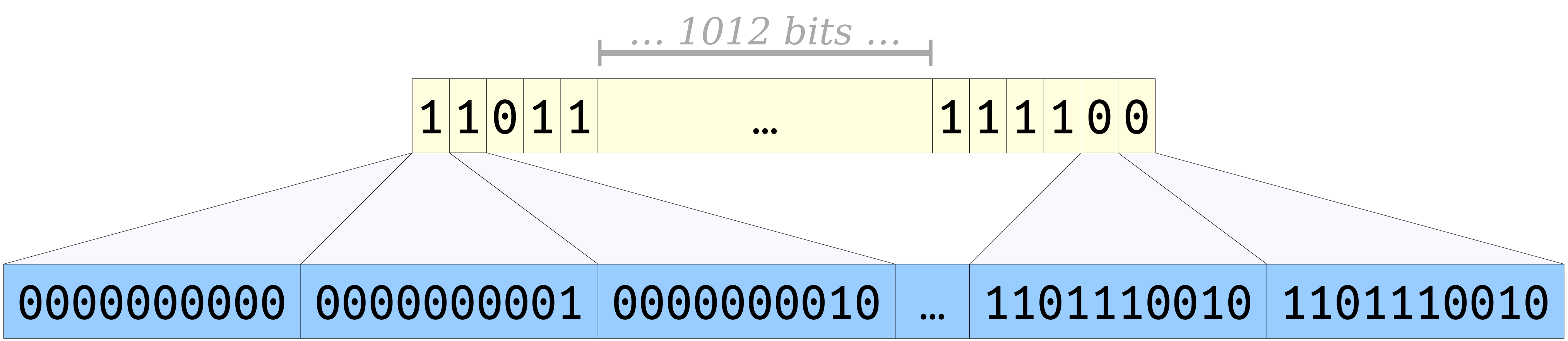
Binary Ranking

- It sure looks like this uses $\Theta(n)$ space.
- But what do we mean by “space” here?
  - Integers usually are represented by machine words.
  - We assume each machine word has ***w*** bits in it (e.g. $w = 32$, $w = 64$, etc.), for a constant w known to us.
- Space: $\boldsymbol{\Theta(nw)}$ bits. This leaves a lot to be desired.
  - On a 64-bit machine, this is a 64x blowup in memory!
- Can we do better?



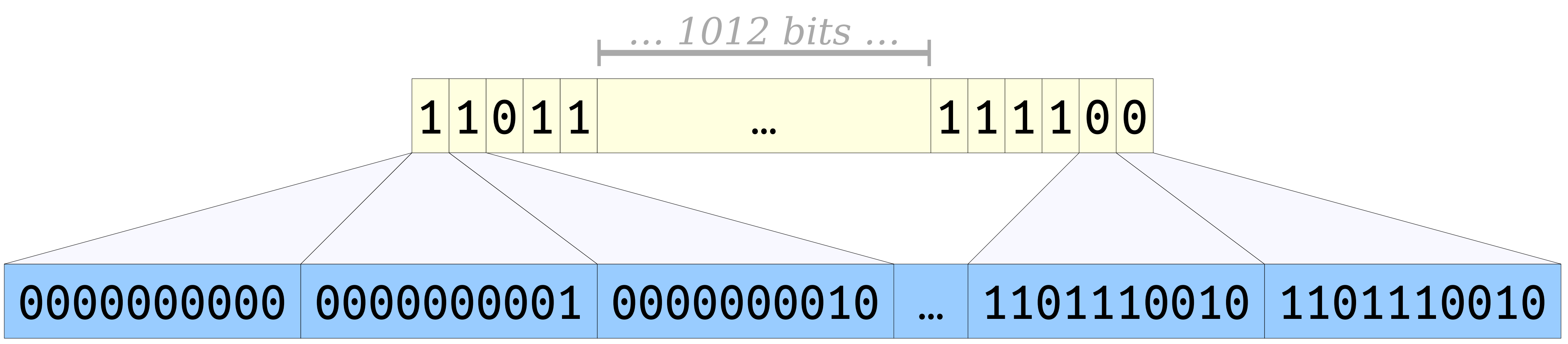
Counting Bits

- Let's suppose we have an array of $1023 = 2^{10} - 1$ bits.
- The prefix sum at each point would be an integer between 0 and 1023, inclusive.
- We only need 10 bits to represent such a prefix sum.
- ***Idea:*** Allocate an array of $10n$ bits, interpreted as an array of n 10-bit numbers.
- This reduces our space usage down to $10n$. It's better than before, but still 10× bigger than the original array.



Counting Bits

- If we maintain an array of prefix sums for an array of n bits, each individual prefix sum is a value between 0 and n, inclusive.
- There are $n+1$ possibilities for what those numbers can be, so each integer requires lg $(n+1)$ bits.
  - We could use fewer bits by using shorter integers for earlier values, but that won't necessarily asymptotically improve space usage.
- Our solution therefore uses O(n log n) bits, but allows for rank queries in time O(1).
- Can we do better?



Counting Bits

- We'll say that a solution to binary ranking is a **⟨*s*(*n*), *q*(*n*)⟩** solution if
  - its space usage is $s(n)$, and
  - queries take time $q(n)$.
- We currently have a ⟨O($n \log n$), O(1)⟩ solution to binary ranking.
- ***Question:*** Can we do better?

	Bits Needed	Query Time
Prefix Sum Array	O($n \log n$)	O(1)

Counting Bits

- We are currently using O(n log n) bits of storage space: O(n) numbers, each of which is O(log n) bits long.
- To improve on this, we could either
  - reduce how many numbers we're storing, or
  - reduce how many bits each number uses.
- ***Question:*** What might that look like?

	Bits Needed	Query Time
Prefix Sum Array	O(n log n)	O(1)

Improving Space Usage

- Split the input array of bits into blocks of b bits each. Then, only store prefix sums at the start of each block.
- To compute the prefix sum at index k:
  - Compute $i = \lfloor k/b \rfloor$, the index of the block containing k.
  - Write down the precomputed prefix sum for block i.
  - Run a linear scan to compute the sum of the first $k \bmod b$ bits of block i.
  - Add these numbers together.

rank(36) = 22

(block 4)

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

Improving Space Usage

- Total space usage: **O((*n* log *n*) / *b*)**.
  - We're storing Θ(*n* / *b*) numbers.
  - Each number needs O(log *n*) bits.
- Query time: **O(*b*)**.
  - We may have to scan Θ(*b*) bits.
- There is no "optimal" choice of *b* here.
  - Increasing *b* decreases memory usage but increases query time.
  - Decreasing *b* decreases query time but increases memory usage.
- We'll therefore leave *b* as a free parameter that whoever is using our data structure can tune.

O(log *n*)-bit numbers

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

The Story So Far

- Earlier, we said there were two strategies we could use to reduce space:
  - Store fewer numbers.
  - Use fewer bits per number.
- Our blocking approach hits this first point. What about the second?

	Bits Needed	Query Time
Prefix Sum Array	$O(n \log n)$	$O(1)$
Partial Prefix Sum Array	$O\left(\frac{n \log n}{b}\right)$	$O(b)$

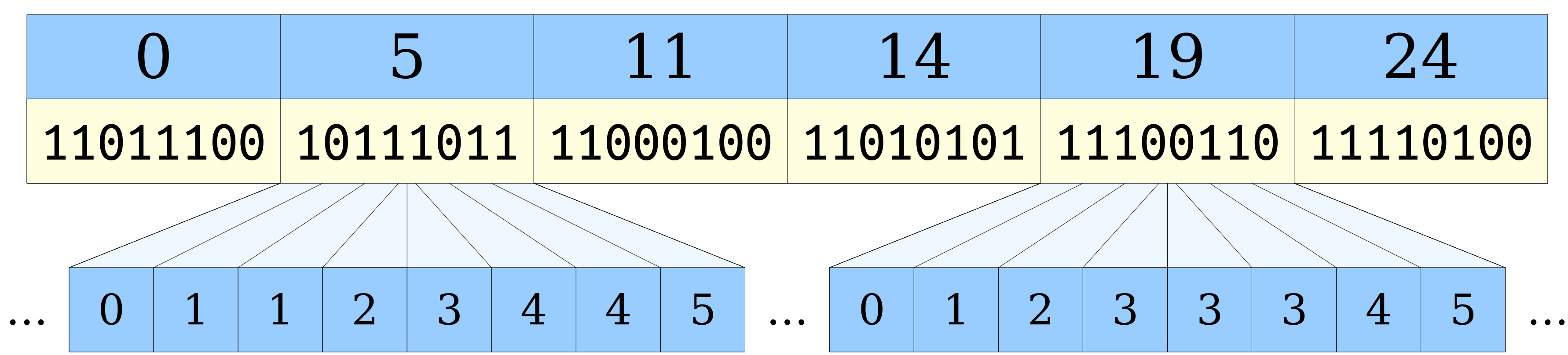
Combining Things Together

- The "slow" step in our query is the linear scan across the bits of a block. Can we speed things up?
- That linear scan is essentially a rank query on an array of b bits.
- ***Idea:*** Rather than use a linear scan there, use our existing ⟨Θ(n log n), O(1)⟩ solution at a per-block level.

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

Combining Things Together

- Instead of one single top-level array, maintain two parallel arrays.
  - The top-level array stores the bit sum up until the start of each block.
  - The second-level array can be thought of as an “array of arrays,” with one array per block, holding answers to rank queries purely within the block.
- There isn’t room in the slides to draw out the full second array; hopefully you can infer from the picture what the remaining entries would be.

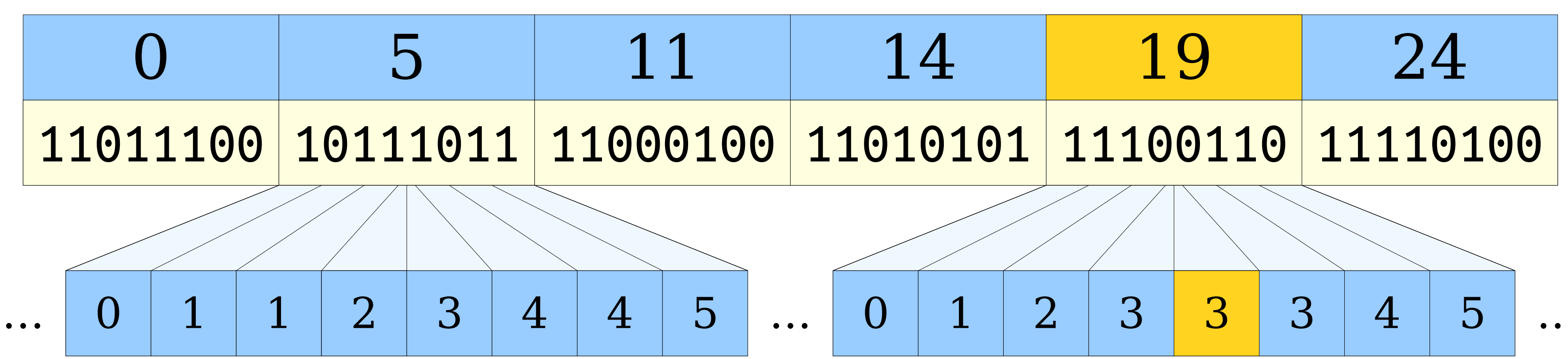


Combining Things Together

- To answer a rank query at index k:
  - Compute $i = \lfloor k/b \rfloor$, the index of the block where the query ends.
  - Look up the ith entry of the top-level table.
  - Look up the (k mod b)th entry of the second-level table's section for block i.
  - Return the sum of those numbers.
- Query cost: **O(1)**.

rank(36) = 22

(block 4)

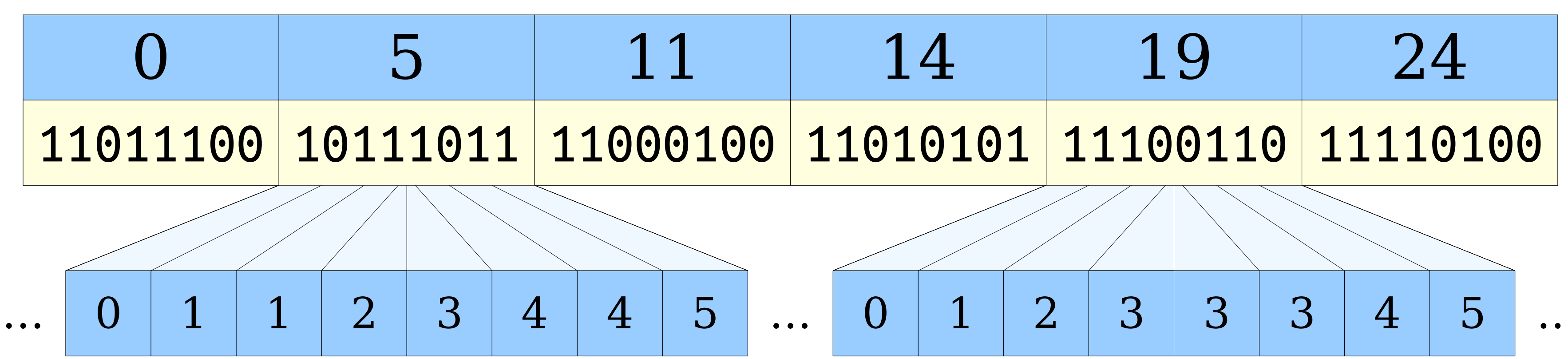


Combining Things Together

- How much memory does this use?

Answer at

https://cs166.stanford.edu/pollev



Intuiting $O\left(\frac{n \log n}{b} + n \log b\right)$

- As b increases:
  - We use less space ***storing partial prefix sums*** at the start of each block, since there are fewer blocks.
  - Each block has more bits, so the ***sums within each block*** require more bits.
- As b decreases:
  - We use more space ***storing partial prefix sums*** at the start of each block, since there are more blocks.
  - Each block has fewer bits, so the ***sums within each block*** requires fewer bits.
- ***Question:*** What choice of b minimizes the above quantity?

Optimizing $O\left(\frac{n \log n}{b} + n \log b\right)$

- Start by taking the derivative:

$$\frac{d}{db}\left(\frac{n \log n}{b} + n \log b\right) = \frac{-n \log n}{b^2} + \frac{n}{b}$$

- Setting equal to zero and solving:

$$\begin{aligned} \frac{-n \log n}{b^2} + \frac{n}{b} &= 0 \\ -\log n + b &= 0 \\ b &= \log n \end{aligned}$$

- Asymptotically optimal choice is ***b* = Θ(log *n*)**, giving space usage **O(*n* log log *n*)**.

The Story So Far

- Our new approach is more space-efficient than our original approach, and works nicely in practice.
  - $\lg \lg 2^{64} = 6$.
- ***Question:*** Can we do better?

	Bits Needed	Query Time
Prefix Sum Array	$O(n \log n)$	$O(1)$
Partial Prefix Sum Array	$O\left(\frac{n \log n}{b}\right)$	$O(b)$
Two-Level Prefix Sums	$O(n \log \log n)$	$O(1)$

Feedback Loops

- Think back to how we arrived at our $\Theta(n \log \log n)$-space solution.
  - We split our array apart into blocks of size b.
  - We stored the prefix sums at the start of each block.
  - We used our $\Theta(n \log n)$-space solution for each block.
- More generally, for that last step, we could have used *any* rank structure we wanted.

0	5	11	14	19	24
11011100	10111011	11000100	11010101	11100110	11110100
Block-Level Rank	Block-Level Rank	Block-Level Rank	Block-Level Rank	Block-Level Rank	Block-Level Rank

Feedback Loops

- Last time, we used our ⟨O(n log n), O(1)⟩ structure per block. It was the best approach we had available.
- But we now have a ⟨O(n log log n), O(1)⟩ structure available, which uses asymptotically fewer bits!
- What happens if we use that one within each block?

0	5	11	14	19	24
11011100	10111011	11000100	11010101	11100110	11110100
O(b lg lg b) Space	O(b lg lg b) Space	O(b lg lg b) Space	O(b lg lg b) Space	O(b lg lg b) Space	O(b lg lg b) Space

Feedback Loops

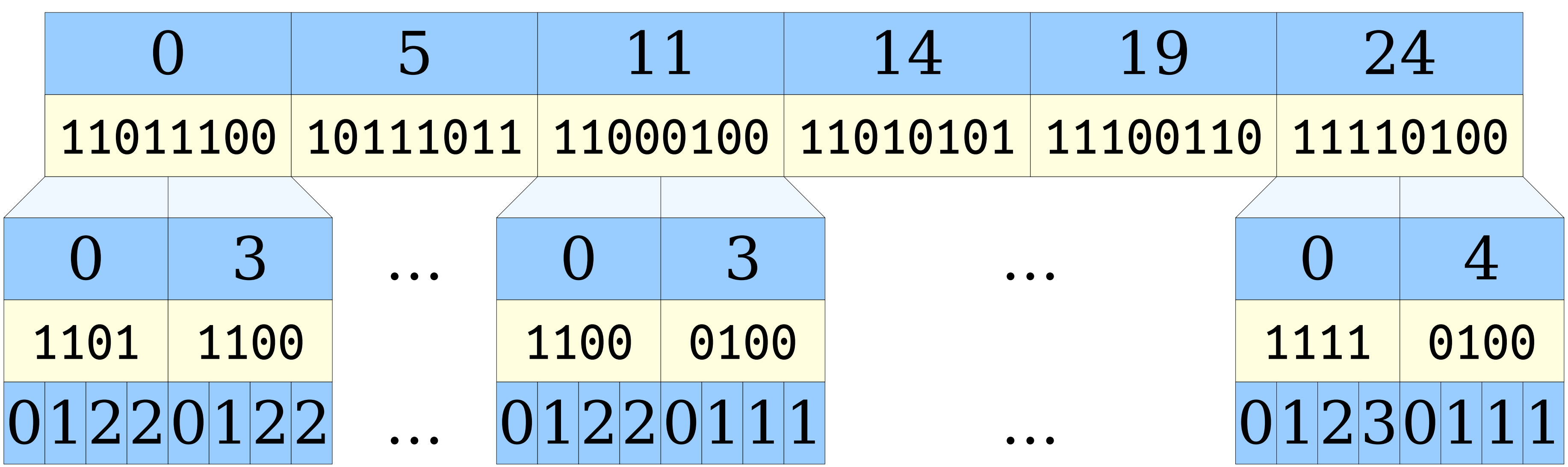
- Split the input apart into blocks of size b.
- Store the prefix sum at the start of each block.
- Use our ⟨O(n log log n), O(1)⟩ solution within each block.
- Compute the overall rank of an index k by combining these answers together.

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

O(b lg lg b) Space	O(b lg lg b) Space	O(b lg lg b) Space	O(b lg lg b) Space	O(b lg lg b) Space	O(b lg lg b) Space

Feedback Loops

- The actual data structure consists of three arrays:
  - A top-level array of prefix sums before each b-bit block.
  - A second-level array of prefix sums before each $(\log b)$-bit "miniblock."
  - A third-level array with prefix sums before each bit of each "miniblock."
- We group these tables into three arrays, one per level, to avoid storing pointers.

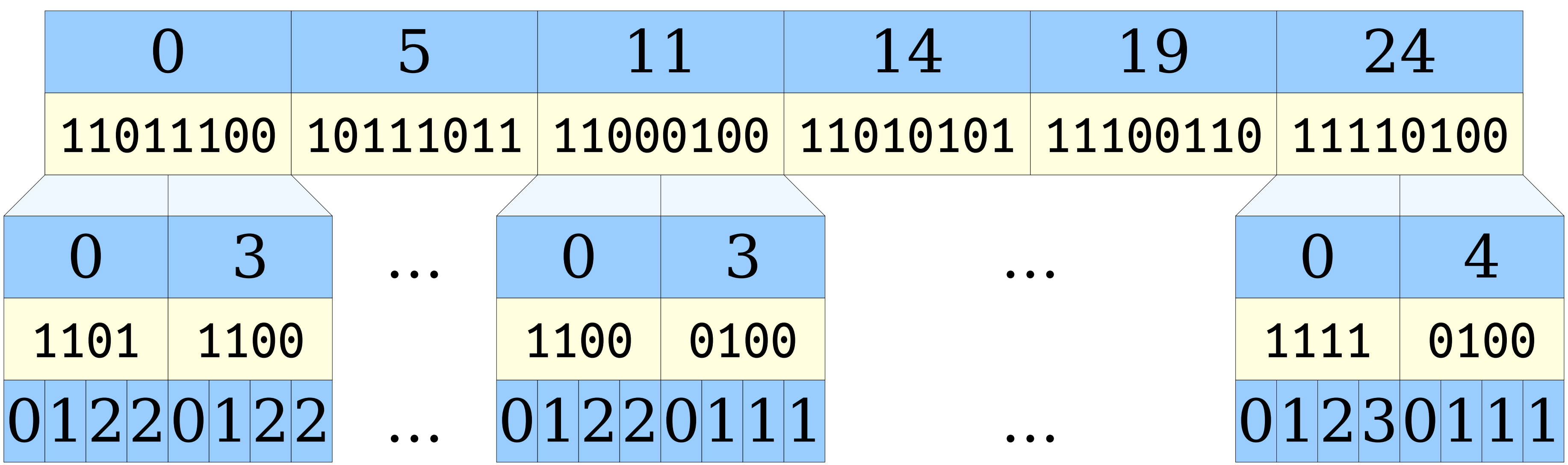


Feedback Loops

- How much memory does this structure use, and what's the query cost?

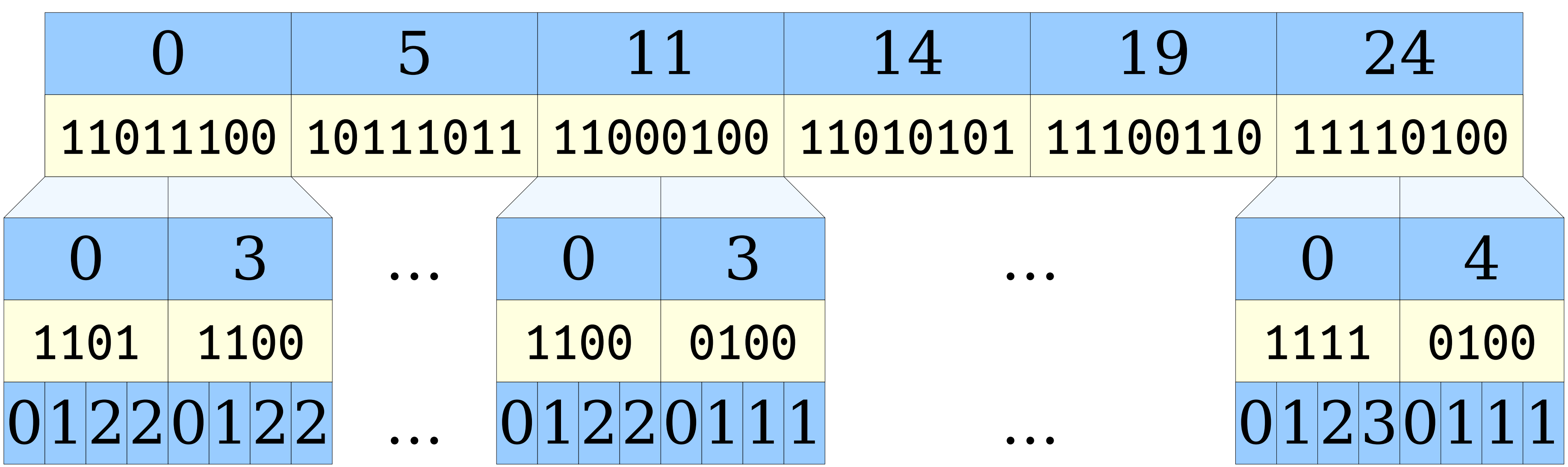
Answer at

https://cs166.stanford.edu/pollev



Feedback Loops

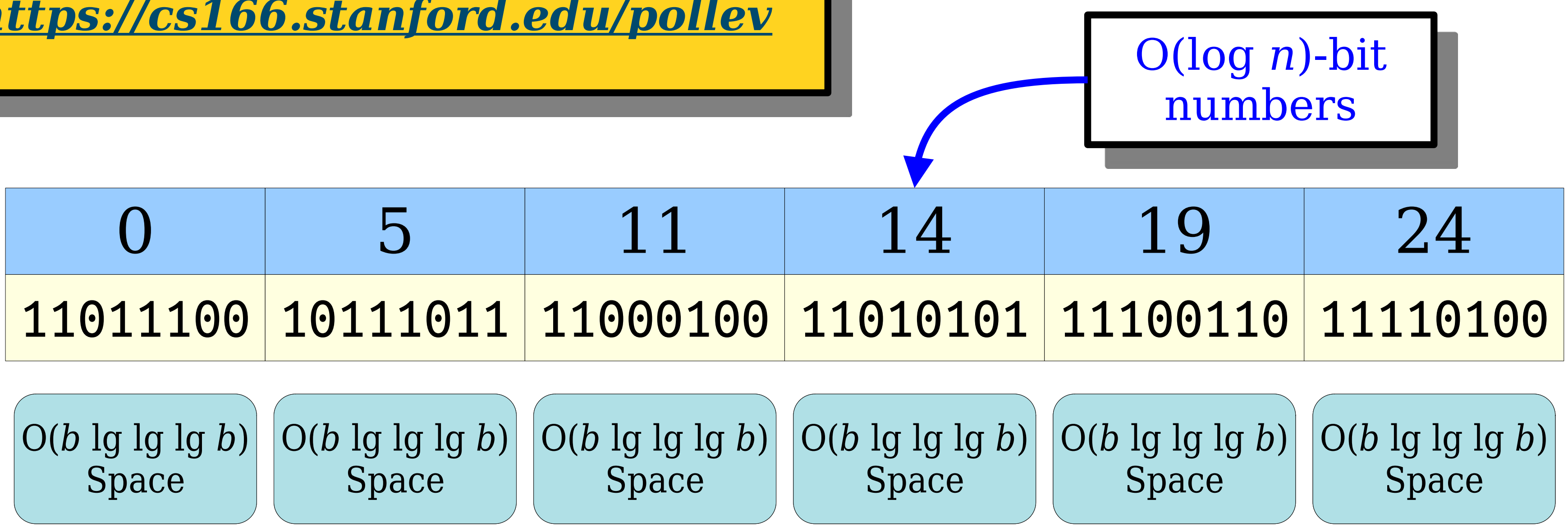
- ***Claim:*** The choice of b that asymptotically minimizes $\Theta((n \log n) / b + n \log \log b)$ is given by $b = \Theta(\log n)$.
- We now have an $\langle O(n \log \log \log n), O(1) \rangle$ solution for ranking!



Feedback Loops

- As you might expect, we can feed this solution back into itself to come up with a $\langle\Theta(n \log \log \log \log n), O(1)\rangle$ solution to ranking.
- More generally, let $\log^{(k)} n$ denote the logarithm function iterated k times.
- ***Question:*** Does this solution allow us to get a $\langle\Theta(n \log^{(k)} n), O(1)\rangle$ solution for all choices of k?

Answer at
https://cs166.stanford.edu/pollev



Counting Layers

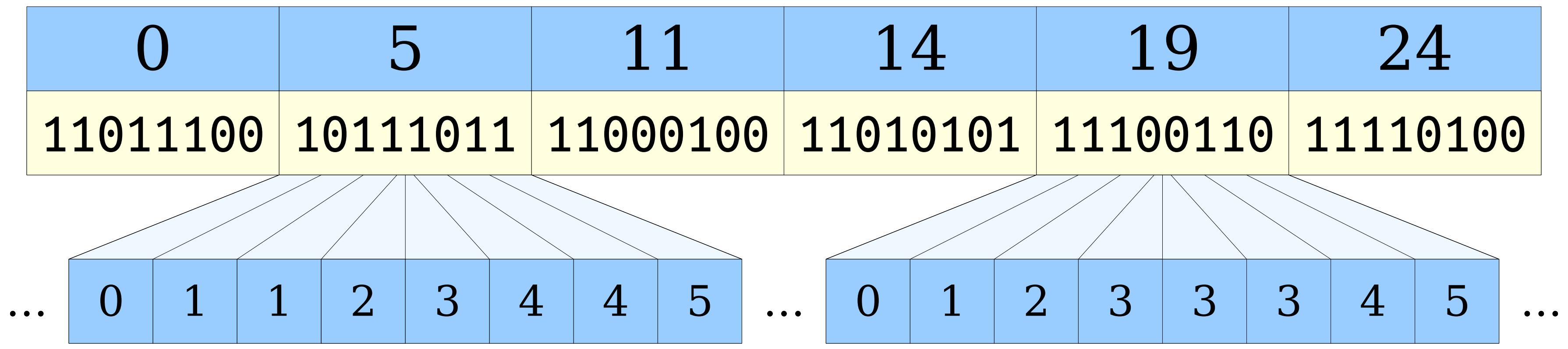
- Our ⟨O(n log$^{(1)}$ n), O(1)⟩ solution to ranking uses a single array of integers to store prefix sums.

0	1	2	2	3	4	5	…	25	26	27	28	29	29	29

`1101110010111011110001001101010111100110111101111100`

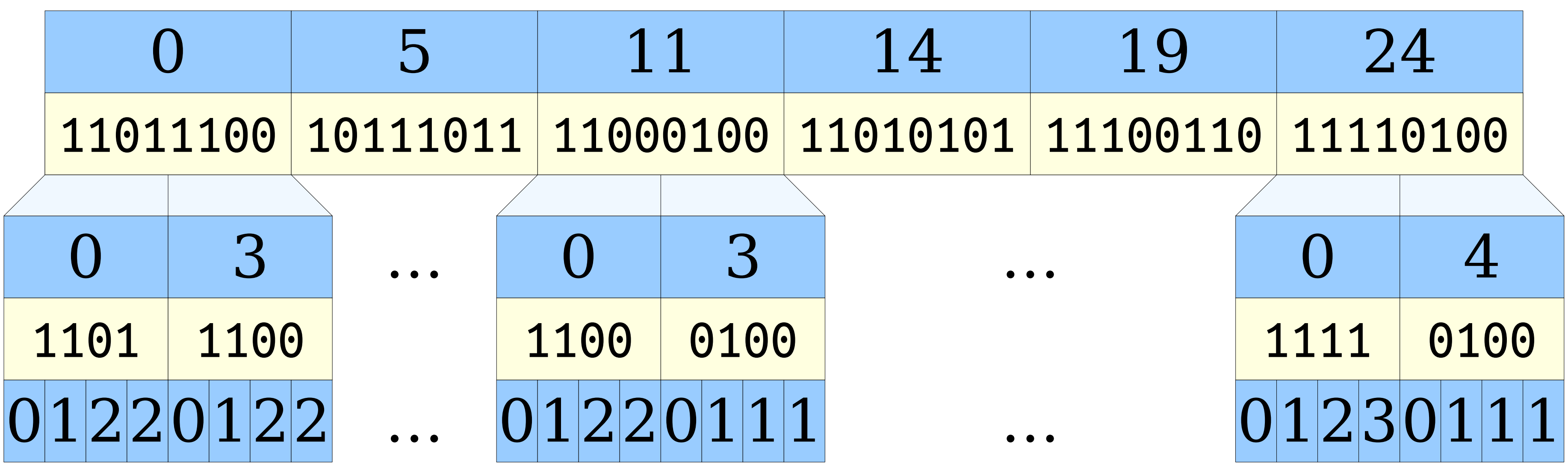
Counting Layers

- Our ⟨O($n \log^{(2)} n$), O(1)⟩ solution to ranking uses two prefix arrays, one at the top level and one for the blocks.



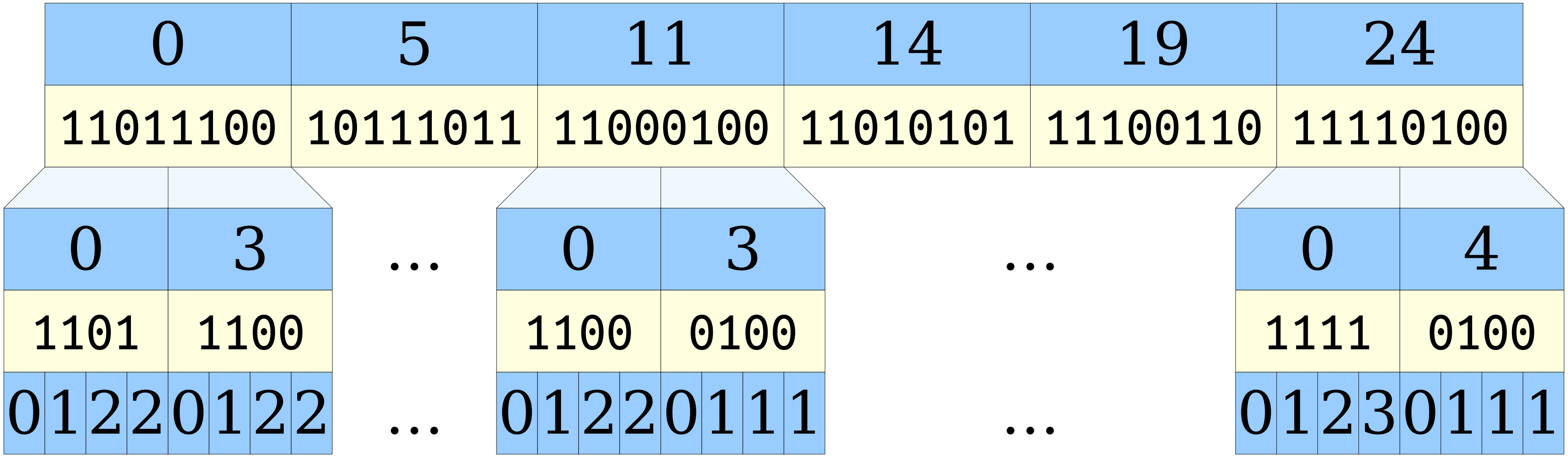
Counting Layers

- Our ⟨O(n log$^{(3)}$ n), O(1)⟩ solution to ranking uses three prefix arrays: one at the top level, one at the block level, and one for "miniblocks."



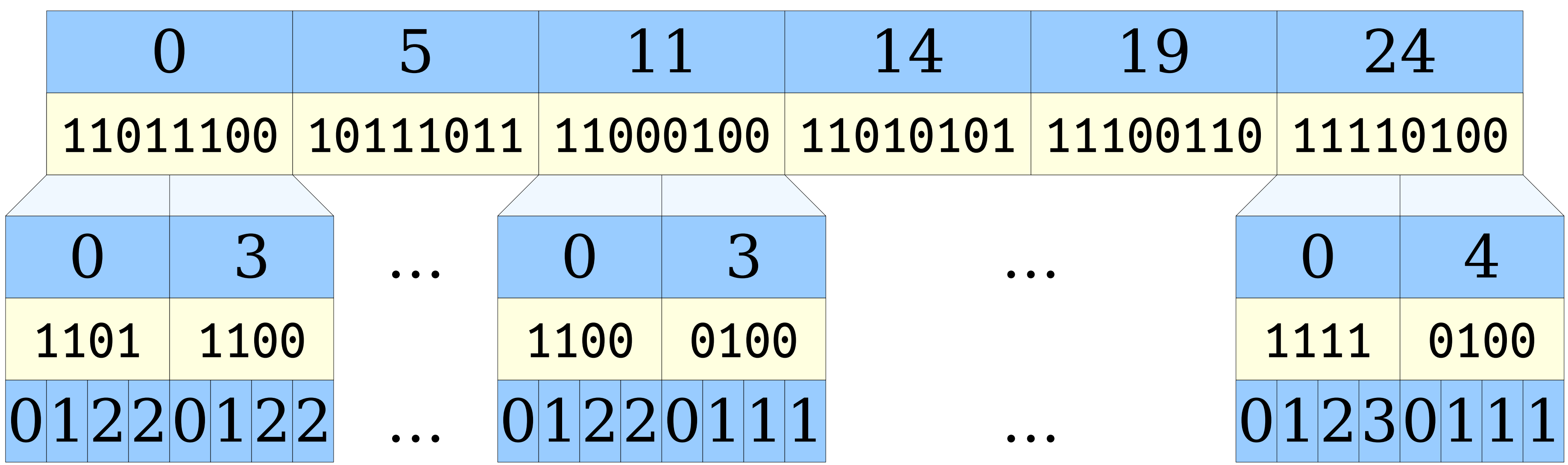
Counting Layers

- More generally, if we have k layers of arrays, we use $O(nk + n \log^{(k)} n)$ bits.
  - Each of the first k – 1 layers requires $O(n)$ bits. *(Why?)*
  - The last layer uses $O(n \log^{(k)} n)$ bits. *(Why?)*
- Our query time is $O(k)$, since we have k layers to navigate.



Counting Layers

- We now have a ⟨O($nk + n \log^{(k)} n$), O(k)⟩ solution for ranking.
- If k is a fixed constant, this is a ⟨O($n \log^{(k)} n$), O(1)⟩ solution to ranking.
- ***Question:*** What if we pick k in terms of n?



Intuiting O($nk + n \log^{(k)} n$)

- What's the impact of tuning k?
  - If k is too large, then we have ***too many layers of recursion*** and the recursive prefix sums use too much space.
  - If k is too small, then we have ***too few layers of recursion*** and the final array of numbers will be too big.
- There should be an optimal choice of k that balances these constraints. What is it?

Iterated Logarithms

- ***Intuition:*** The log function is incredibly effective at shrinking down large quantities.
  - Number of protons in the known universe: $\approx 2^{240}$.
  - $\log^{(0)} 2^{240}$ = 1,766,847,`[… 57 digits …]`,292,619,776
  - $\log^{(1)} 2^{240} = 240$
  - $\log^{(2)} 2^{240} \approx 7.91$
  - $\log^{(3)} 2^{240} \approx 2.98$
  - $\log^{(4)} 2^{240} \approx 1.58$
- More generally, for any natural number n, there is some minimum k for which $\log^{(k)} n \leq 2$.
- The ***iterated logarithm of n***, denoted **log* *n***, is the smallest choice of k that makes $\log^{(k)} n \leq 2$.
- Question to ponder: what's the smallest n where $\log^* n = 10$?

Iterated Logarithms

- For any choice of k, we have a

$$\langle O(nk + n \log^{(k)} n), O(k) \rangle$$

  solution to ranking.
- Pick $\boldsymbol{k = \log^* n}$. This gives us a

$$\boldsymbol{\langle O(n \log^* n), O(\log^* n) \rangle}$$

  solution to binary ranking.
- In practice, this is *essentially* a $\langle O(n), O(1) \rangle$ solution to ranking.
  - (If $n \leq 2^{64}$, then $\log^* n = 4$. So four layers of structure would always suffice.)

The Story So Far

- We have an (almost) linear-space solution to ranking.
- There's still more room for improvement.
  - Practically, we're still using ≈5n total bits.
  - Theoretically, we'd like to remove the log* n factor.
- Can we do better?

	Bits Needed	Query Time
Prefix Sum Array	O(n log n)	O(1)
Two-Level Prefix Sums	O(n log log n)	O(1)
Multilevel Prefix Sums	O(n log* n)	O(log* n)

Time-Out for Announcements!

Problem Set 1

- Problem Set 0 (Concept Refresher) was due today at 1:00PM.
  - Need more time? You can use up to two late days to extend the deadline by 24 or 48 hours.
- Problem Set 1 (***RMQ***) goes out today. It's due next Tuesday at 1:00PM.
  - You may work with a partner on this assignment if you'd like.
  - Play around with the RMQ structures from last week, and see what it's like to code them up!
- As always, ping us on EdStem or stop by office hours if you have questions!

Back to CS166!

An Alternative Approach

An Alternative Approach

- Our best approach so far involves the following idea:
  - Split the input array into smaller blocks.
  - Recursively build fast ranking structures per block.
- The recursion in that second step is where we get the O(log* n) query time from.
- ***Question:*** Can we avoid having to run the recursion in the last step?

An Alternative Approach

- When we set out to split our input apart into blocks, we left the choice of block size b unspecified.
- Later, we found that $b = \Theta(\log n)$ was the optimal choice.
  - This means that our blocks are *tiny* compared to the size of our input array.
- ***Key Intuition:*** These blocks are so small that there can't be "too many" distinct blocks.
- ***Question:*** Where have you seen this idea before?

The Four Russians Strategy

- As an example, imagine that we pick our block size as $b = 3$.
- There are only eight possible blocks:

`000 001 010 011 100 101 110 111`

- We could therefore build a table keyed on a combination of a block and an index in into the block:

	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

The Four Russians Strategy

- There are only 2^b possible blocks.
- There are $O(b)$ positions within a block.
- Each prefix sum within a block requires $O(\log b)$ bits to write out.
- Total space: $O(2^b \cdot b \cdot \log b)$.

	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

The Four Russians Strategy

- Total space: $O(2^b \cdot b \cdot \log b)$.
- Plugging in ***b* = ½ lg *n*** gives a space usage of

  $= O(2^{½ \lg n} \cdot \log n \cdot \log \log n)$

  $= O(n^{½} \log n \log \log n)$

  $= o(n^{⅔})$.
- This is *sublinear* space for sufficiently large *n*.

	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

The Four Russians Strategy

- Split the input apart into blocks of size ½ lg n.
- Compute the prefix sum to the start of each block.
  - This uses $O((n \log n) / \log n) = O(n)$ bits.
- Build a table of all possible rank queries on all possible blocks. This uses $o(n^{2/3})$ bits.
- Total space: **$O(n)$**.

0	2	5	6	8	10	13	13	14	16	18	20
110	111	001	011	101	111	000	100	110	101	101	110

	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

The Four Russians Strategy

- To perform a query for the rank sum up to index k:
  - Compute $i = \lfloor {}^{k}/_{b} \rfloor$, the index block k falls in.
  - Use the bits of block i as an index into the secondary table, then look up row k mod b.
  - Add the Four Russians table number to the ith entry of the top-level array.
- Query time: **O(1)**.

rank(17) = 12

(block 5)

0	2	5	6	8	10	13	13	14	16	18	20
110	111	001	011	101	111	000	100	110	101	101	110

	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

The Story So Far

- This new approach uses O(n) bits and can support queries in time O(1).
- It seems like there's no more room for improvement here – are we done?

	Bits Needed	Query Time
Prefix Sum Array	O(n log n)	O(1)
Multilevel Prefix Sums	O(n log* n)	O(log* n)
Four Russians	O(n)	O(1)

The Story So Far

- Our Four Russians approach uses $\Theta(n)$ extra bits beyond the bits in the original array. The actual number is actually

$$2n + o(n)$$

  because we need to store

  - n / (½ lg n) = $2n$ / lg n indices in the top-level table,
  - each index is lg (n + 1) bits long, and
  - we need $o(n)$ bits for the precomputed tables.

- This is a marked improvement over our original approach, but it still means we need at least twice as many bits as in the original array.
- ***Goal:*** Reduce the space usage *even further*.

The Story So Far

- The two space-efficient solutions we've developed so far are based on different ideas.
  - Multilevel Prefix Sums: subdivide the array into blocks, then recursively subdivide those blocks even further.
  - Four Russians: Once we reach blocks of size ½ lg n or smaller, precompute all possible answers to all possible queries.
- What happens if we combine these strategies together?

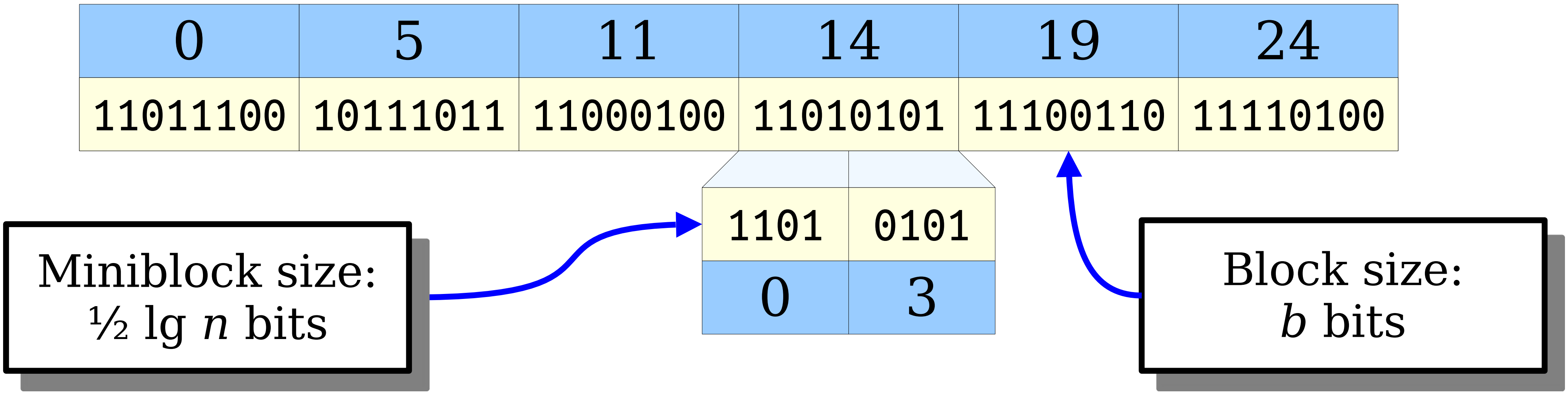
	Bits Needed	Query Time
Multilevel Prefix Sums	$O(n \log^* n)$	$O(\log^* n)$
Four Russians	$O(n)$	$O(1)$

The Combined Approach

- We begin with an array of n bits. We ultimately need to reduce the array size to ½ lg n to use the Four Russians approach.
- If we immediately subdivide into blocks of that size, we get our ⟨O(n), O(1)⟩ solution.
- ***Idea:*** Introduce some intermediate level of subdivision between the original array and the blocks of size ½ lg n.

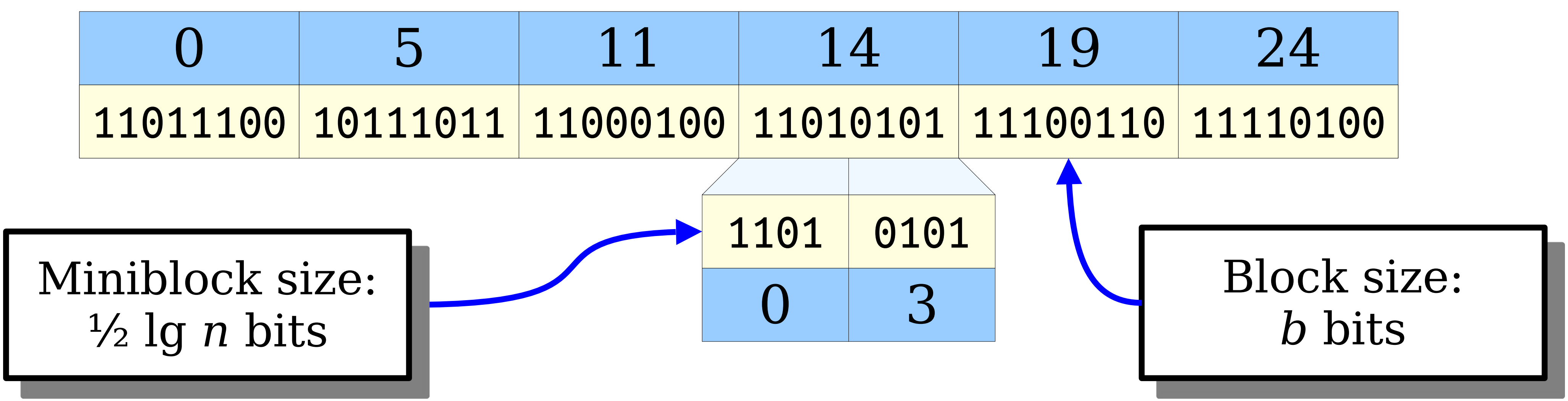
The Combined Approach

- Subdivide the array into Θ(n / b) blocks of size b.
- Write prefix sums of O(log n) bits at the start of each block.
- Subdivide each block into Θ(b / log n) miniblocks of size ½ lg n.
- Write prefix sums of O(log b) bits at the start of each miniblock.
- Precompute a table of all rank queries on all miniblocks (not shown), using $o(n^{2/3})$ bits.



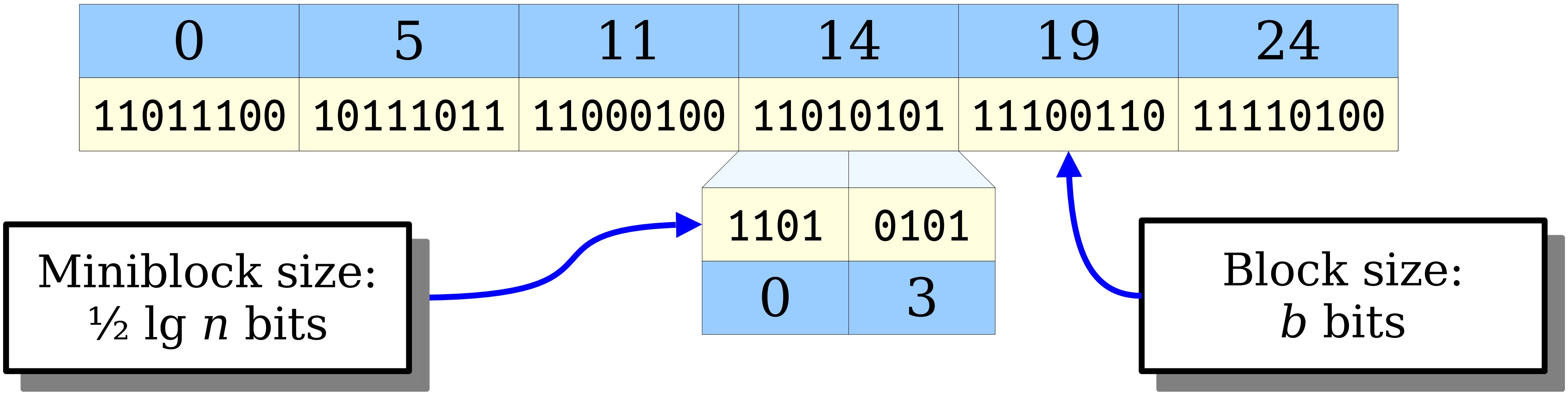
The Combined Approach

- To perform a query for the prefix sum at index k:
  - Compute $i = \lfloor k/b \rfloor$, the index of the block containing k. Write down the prefix sum at the start of block i in the top-level array.
  - Compute $j = \lfloor (k \bmod b) / (½ \lg n) \rfloor$, the index of the miniblock within block i containing k. Write down the prefix sum at the start of miniblock i in the second-level array.
  - Look up $(k \bmod b) \bmod ½ \lg n$ in the precomputed table for the miniblock to get the prefix sum within the miniblock.
  - Add these values together.
- Total query time: **O(1)**.



The Combined Approach

- Space for top-level array: O((n log n) / b).
- Space for the miniblocks: O((n log b) / log n)
  - O(n / log n) total miniblocks.
  - O(log b) bits per miniblock for a prefix sum.
- Space for the Four Russians table: $o(n^{2/3})$.
- Total space: **O((n log n) / b + (n log b) / log n) + $o(n^{2/3})$**.
- What's the optimal choice of b here?



Optimizing $O\left(\frac{n \log n}{b} + \frac{n \log b}{\log n}\right)$

- Start by taking the derivative:

$$\frac{d}{db}\left(\frac{n \log n}{b} + \frac{n \log b}{\log n}\right) = \frac{-n \log n}{b^2} + \frac{n}{b \log n}$$

- Setting equal to zero and solving:

$$\begin{aligned} \frac{-n \log n}{b^2} + \frac{n}{b \log n} &= 0 \\ -\log^2 n + b &= 0 \\ b &= \log^2 n \end{aligned}$$

- Asymptotically optimal space usage is when we pick $b = \Theta(\log^2 n)$.
- If we do that, our space usage is

$$O\left(\frac{n \log n}{b} + \frac{n \log b}{\log n}\right)$$

The Combined Approach

- We now have a solution that uses a *sublinear* number of auxiliary bits.
- The space usage for the original array, plus our structure, is $n + o(n)$. As n increases, we need proportionally fewer and fewer bits!

	Bits Needed	Query Time
Multilevel Prefix Sums	$O(n \log^* n)$	$O(\log^* n)$
Four Russians	$O(n)$	$O(1)$
Two-Level Four Russians (Jacobson's Structure)	$O\left(\frac{n \log \log n}{\log n}\right)$	$O(1)$

Succinct Data Structures

- A data structure is called ***succinct*** if it uses $B + o(B)$ bits, where B is the information-theoretic minimum number of bits needed to solve the problem.
- In the case of binary rank, we must use at least n bits of space.
  - We can recover the original bit array using rank queries, and an arbitrary n-element bit array can't be stored in fewer than n bits.
  - (Why can't we use fewer than n bits?)
- Our space usage for our rank structure is $n + o(n)$ and is thus succinct.

Further Work

- These ideas – plus some further refinements – work well in practice.
  - Check out the libraries `rank9`, `poppy`, etc. to see how these look in practice.
- Further work in Theoryland has produced $\langle O(n / \log^k n), O(k) \rangle$ structures for any constant k.
  - Many of the techniques employed here come from data compression – very cool!
- There's also work done into compressing bitvectors while allowing for fast access to individual elements, allowing for even greater space reductions.
  - Assuming the bitvector has some "nice" structure to it, we can sometimes encode it in space $o(n)$ as well!

Summary for Today

- When you drop to the level of counting individual bits, data structure design gets a lot more complex (and interesting)!
- Recursively subdividing larger structures into smaller pieces is a great way to reduce space usage.
- The Method of Four Russians is a fantastic way to handle arrays once they get sufficiently small.
- Using a fixed number of recursive reductions, then switching to a Four Russians speedup, is a common strategy for building sublinear-space data structures.

Next Time

- ***Succinct Select***
  - Computing the inverse of rank queries.
- ***Sparse/Dense Subdivisions***
  - Handling disparate cases nonuniformly.