# Succinct Rank and Select

# Working With Bits and Words

- When building data structures, we often treat the data we're storing as "black-box" units.
  - e.g. BSTs only care that the items stored are comparable, hash tables only care that they're hashable, etc.
- However, they're made up of individual bits or individual machine words.
- Our next few lectures explore the theme of looking at how values are represented inside the machine from a data structures perspective.
- We'll also see some amazing techniques that involve harnessing the intrinsic parallelism made possible through operations on machine words.

# Where We're Going

- ***Succinct Data Structures (Today)***
  - Minimizing the number of bits necessary to represent a data structure.
- ***Word-Level Parallelism (Next Week)***
  - Harnessing the parallelism inherent in individual integer operations.

# Outline for Today

- ***The Binary Rank Problem***
  - Prefix sums on bitvectors.
- ***Jacobson's Succinct Rank Structure***
  - Solving binary rank using a small number of bits.
- ***The Binary Select Problem***
  - The inverse problem to ranking.
- ***Clark's Succinct Select Structure***
  - Solving selection in a small number of bits.

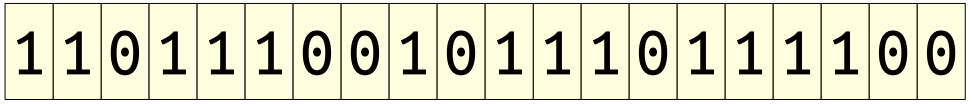
# Binary Ranking

# Binary Ranking

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

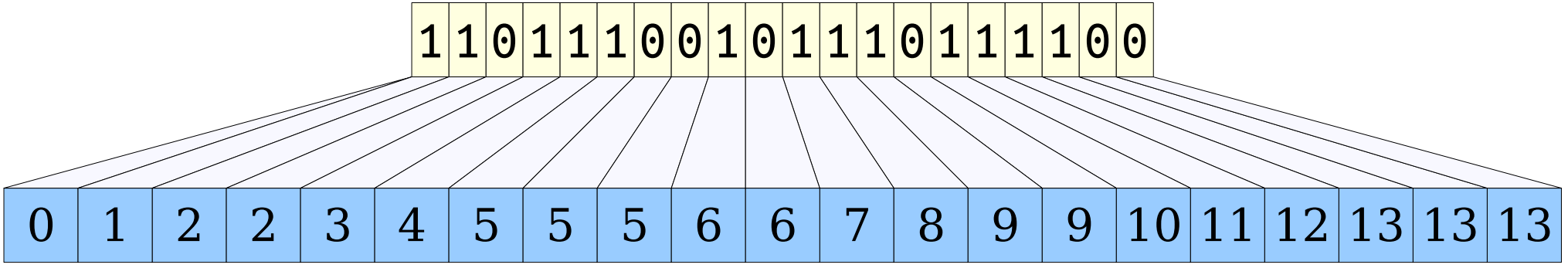
  ***Given a list of n bits and an index i, return the sum of all the bits up to position i in the list.***

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



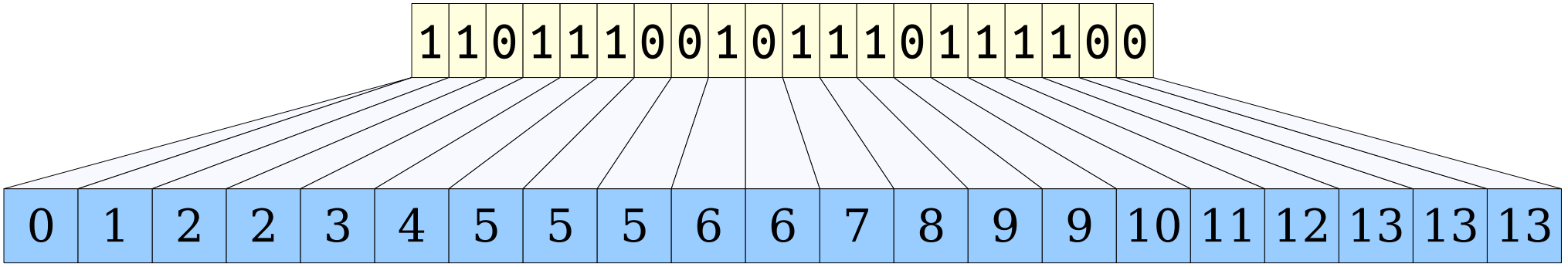
# Binary Ranking

- 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?



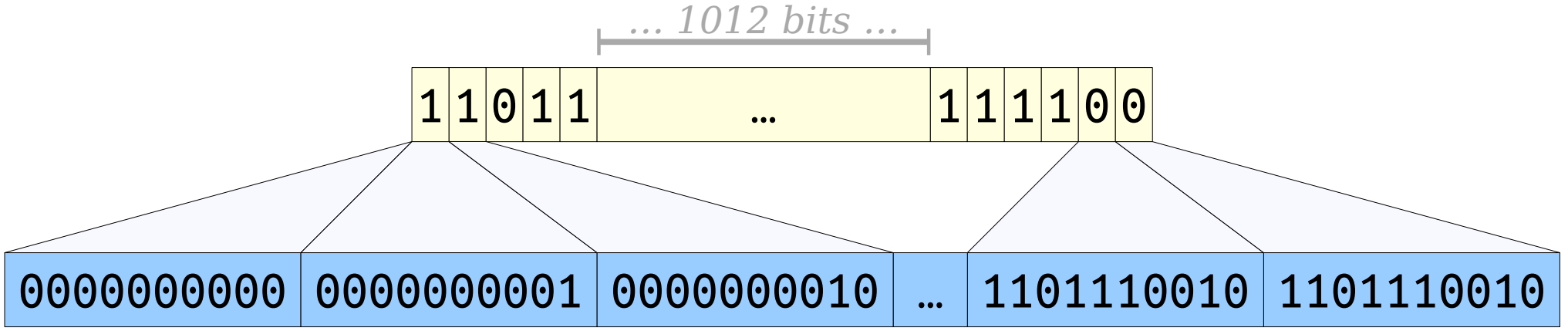
# Binary Ranking

- 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: **$\Theta(nw)$** bits. This leaves a lot to be desired.
  - On a 64-bit machine, this is a 64x blowup in memory!
- Can we do better?



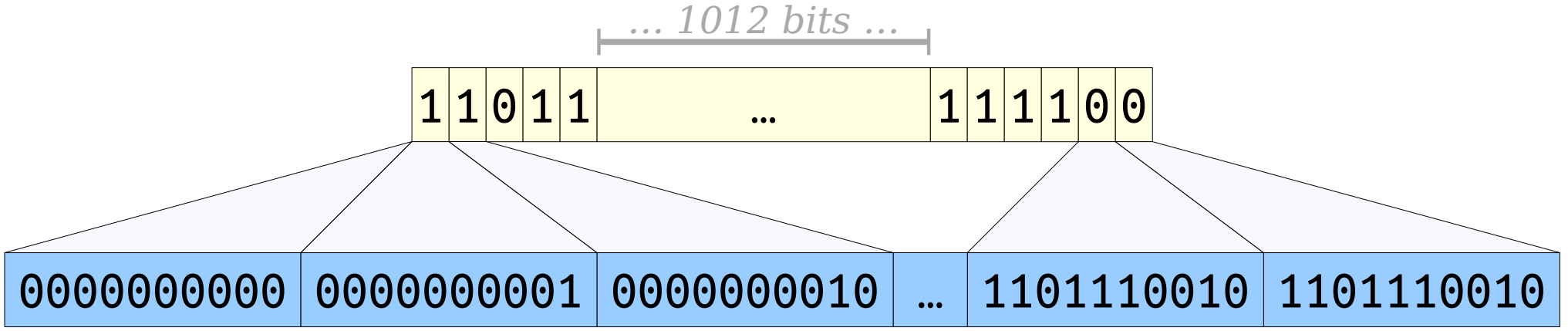
# Counting Bits

- 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 could 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 might be able to squeeze out a few more bits by using shorter integers for earlier values, but nothing that improves asymptotic 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) |

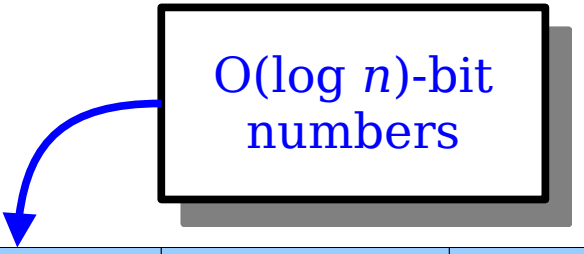
# Counting Bits

- 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

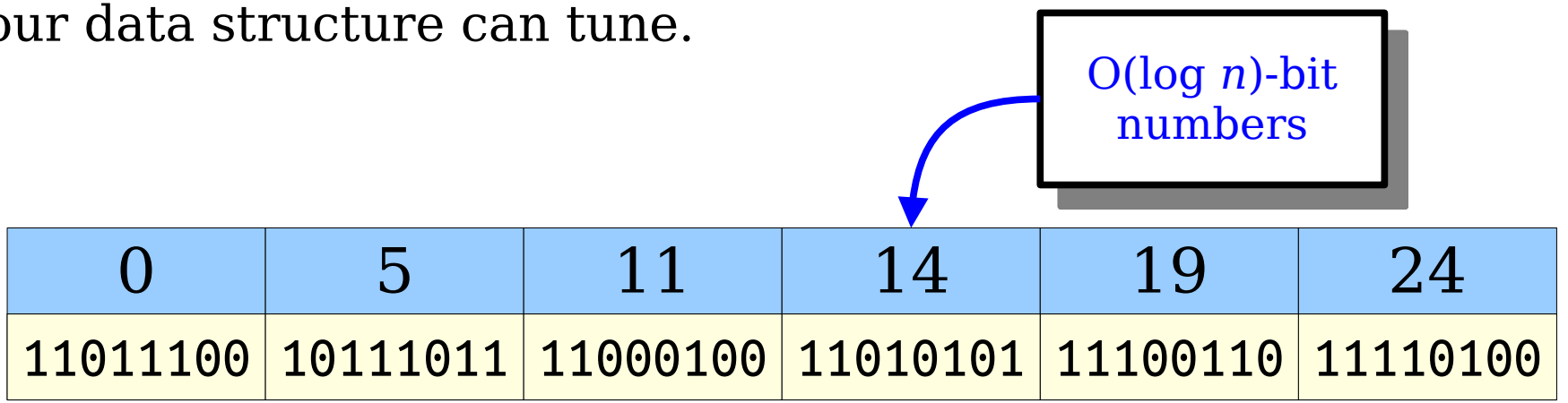
- Split the input array of bits into blocks of $b$ bits each. Then, only store prefix sums at the start of each block.
- To compute the prefix sum at index $k$:
  - Read the prefix sum at the start of block $\lfloor k/b \rfloor$.
  - Run a linear scan to compute the sum of the first $k \bmod b$ bits of the block.




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



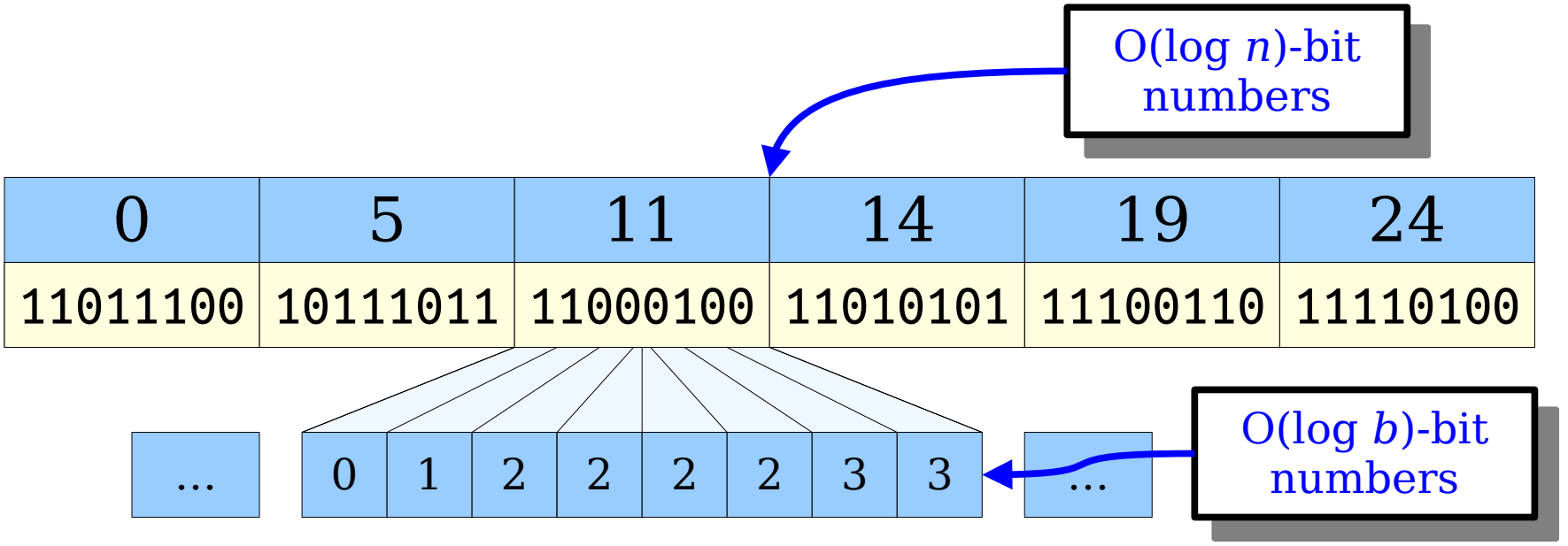
# The Story So Far

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

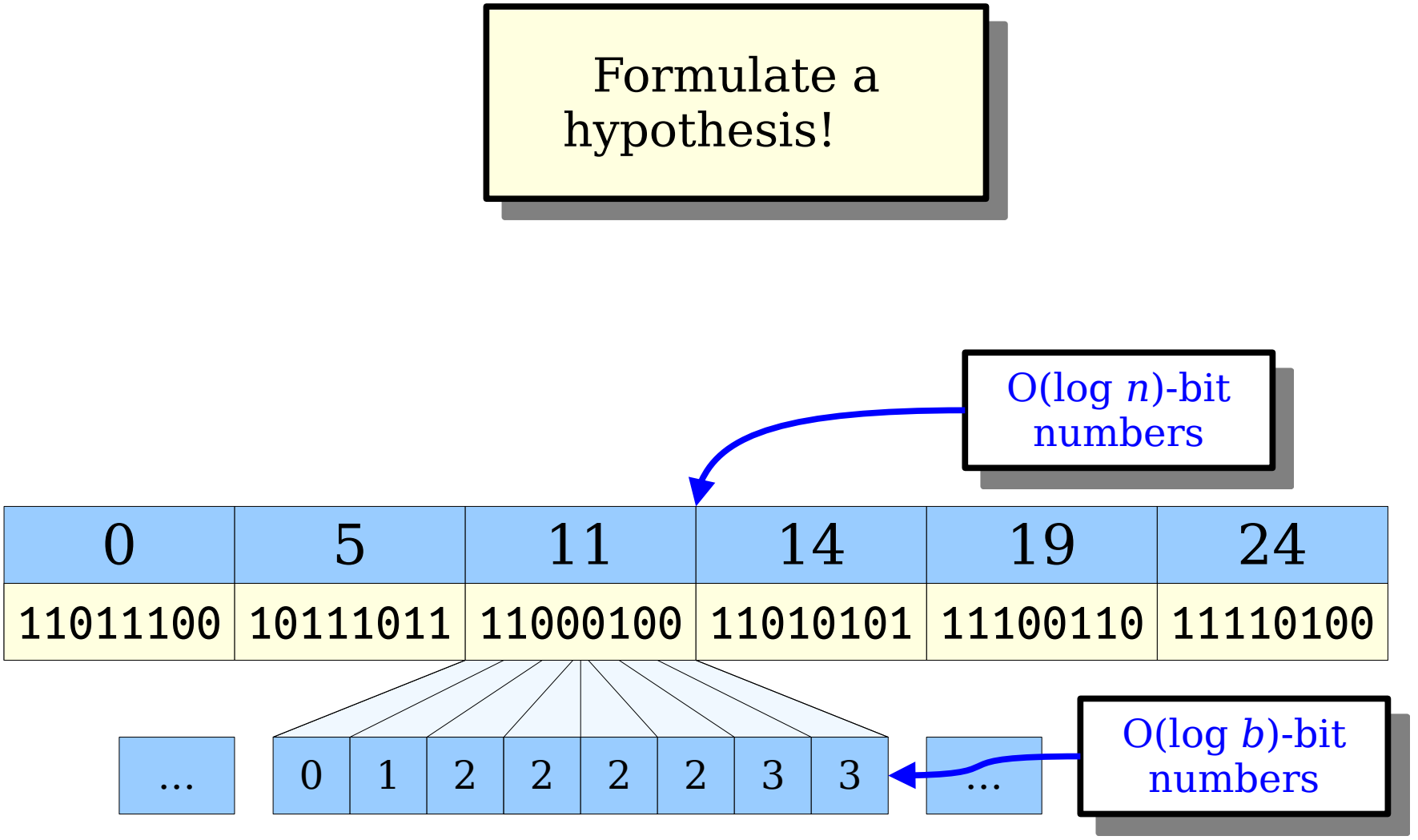
# Combining Things Together

- 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 $\langle \Theta(n \log n), O(1) \rangle$ solution at a per-block level.



# Combining Things Together

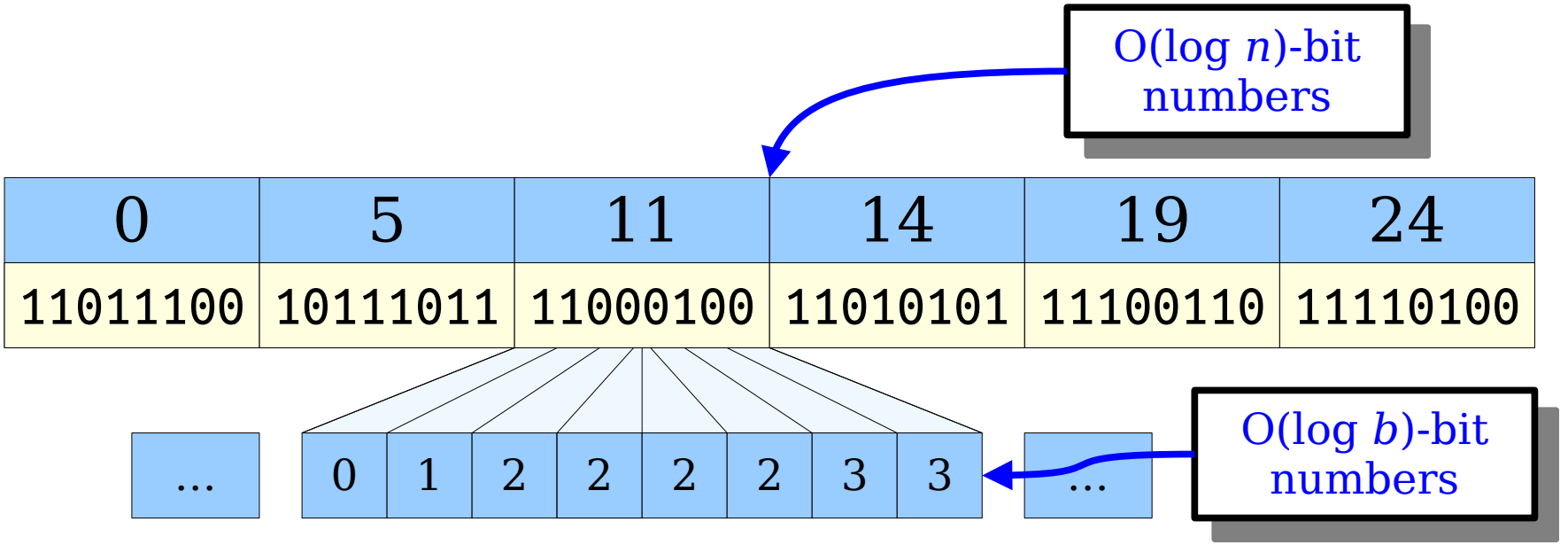
- How much memory does this use?



# Combining Things Together

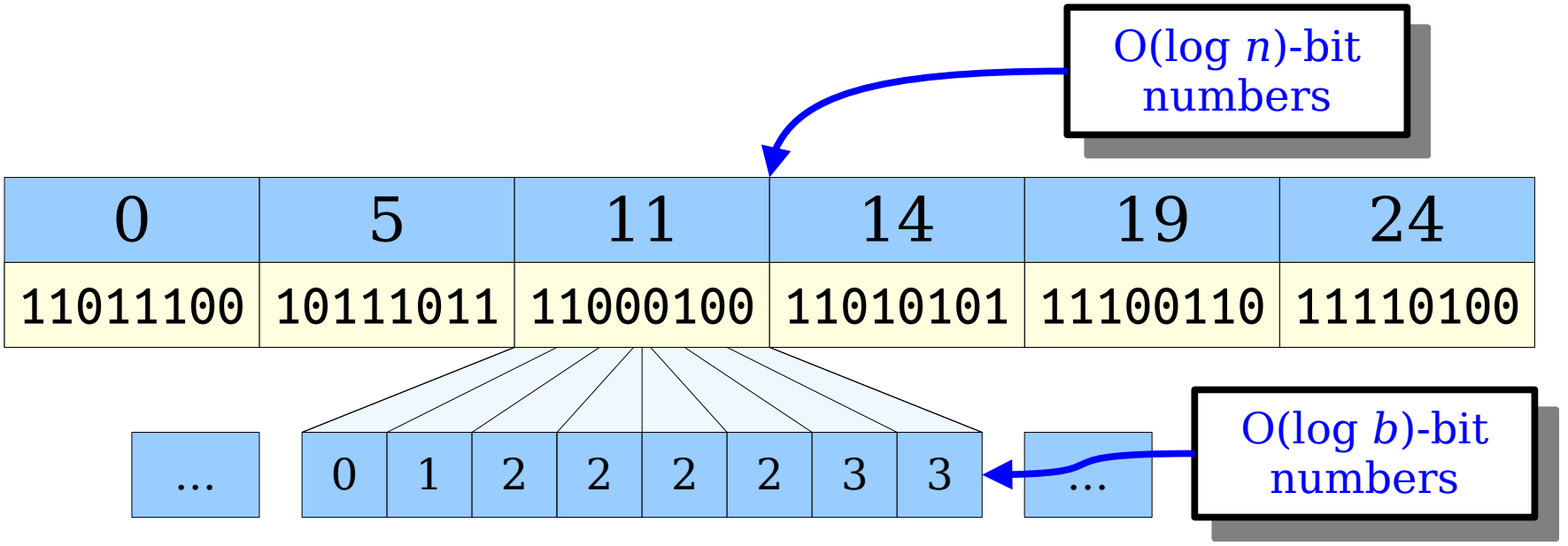
- How much memory does this use?

Discuss with your neighbors!



# Combining Things Together

- At the top level, we're storing $\Theta(n / b)$ integers, one at the start of each block, and each uses $O(\log n)$ bits.
  - Space for those integers: $O((n \log n) / b)$.
- There are $\Theta(n / b)$ blocks, and each block requires $O(b \log b)$ storage.
  - There are $b$ bit positions within each block, and each position has a prefix sum that goes between 0 and $b$. That means we need $O(b \log b)$ total space.
  - Space for those block-level structures: $O(n \log b)$.
- Total space usage: $\mathbf{O\left(\frac{n \log n}{b} + n \log b\right)}$.



# 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:

$$\frac{-n \log n}{b^2} + \frac{n}{b} = 0$$

$$-\log n + b = 0$$

$$b = \log n$$

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

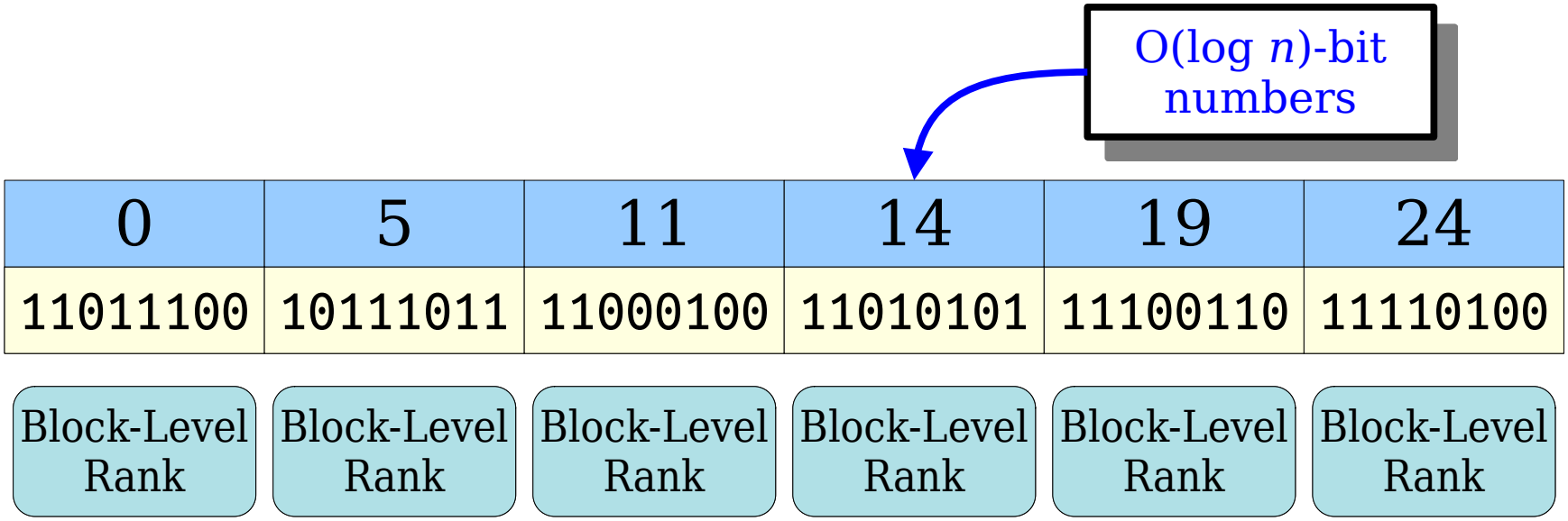
# The Story So Far

- Our new approach is more space-efficient than our original approach, and works nicely in practice.
- However, we're still using more bits for our rank data structure than the array of bits itself needs.
- ***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 | $O(n \log \log n)$ | $O(1)$ |

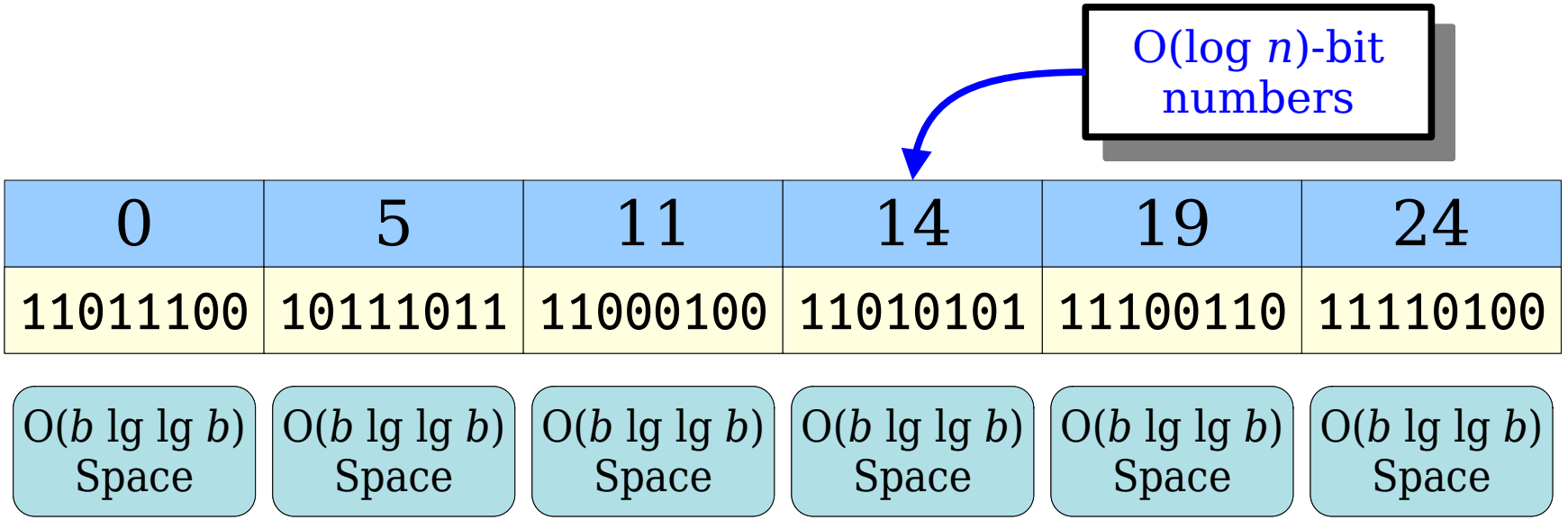
# Feedback Loops

- 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.



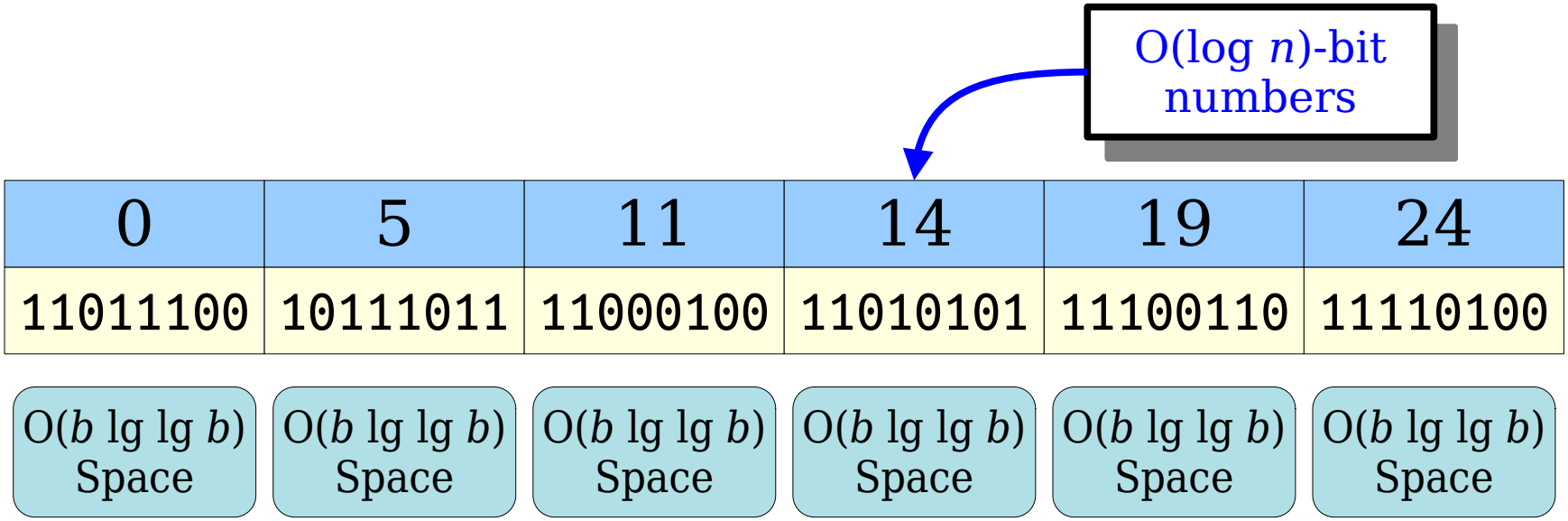
# Feedback Loops

- 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?



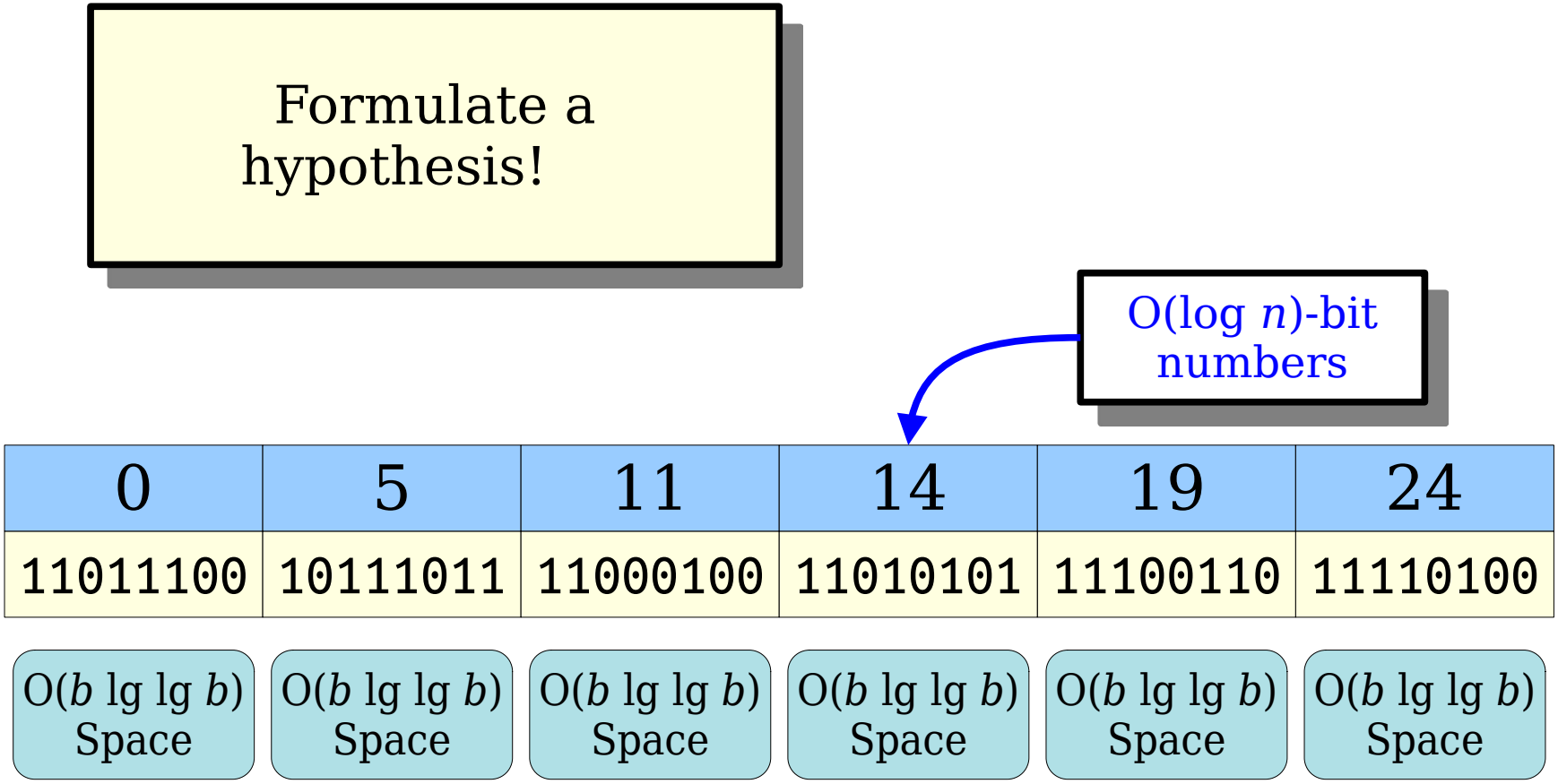
# Feedback Loops

- 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.



# Feedback Loops

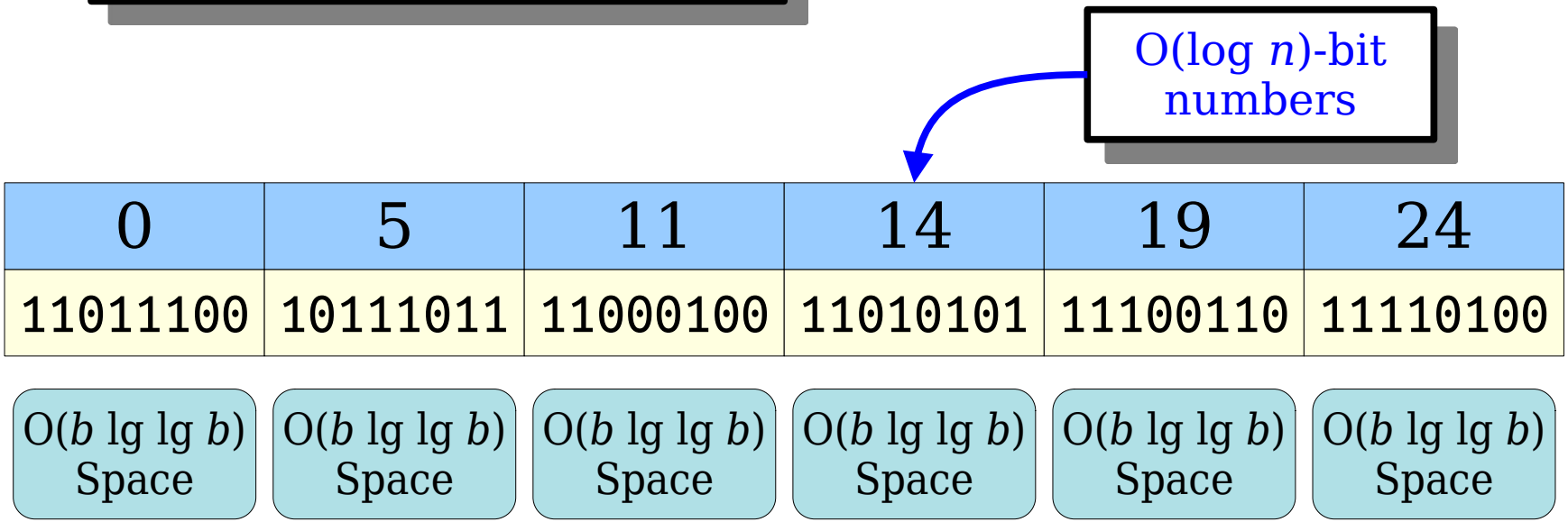
- How much memory does this structure use, and what's the query cost?



# Feedback Loops

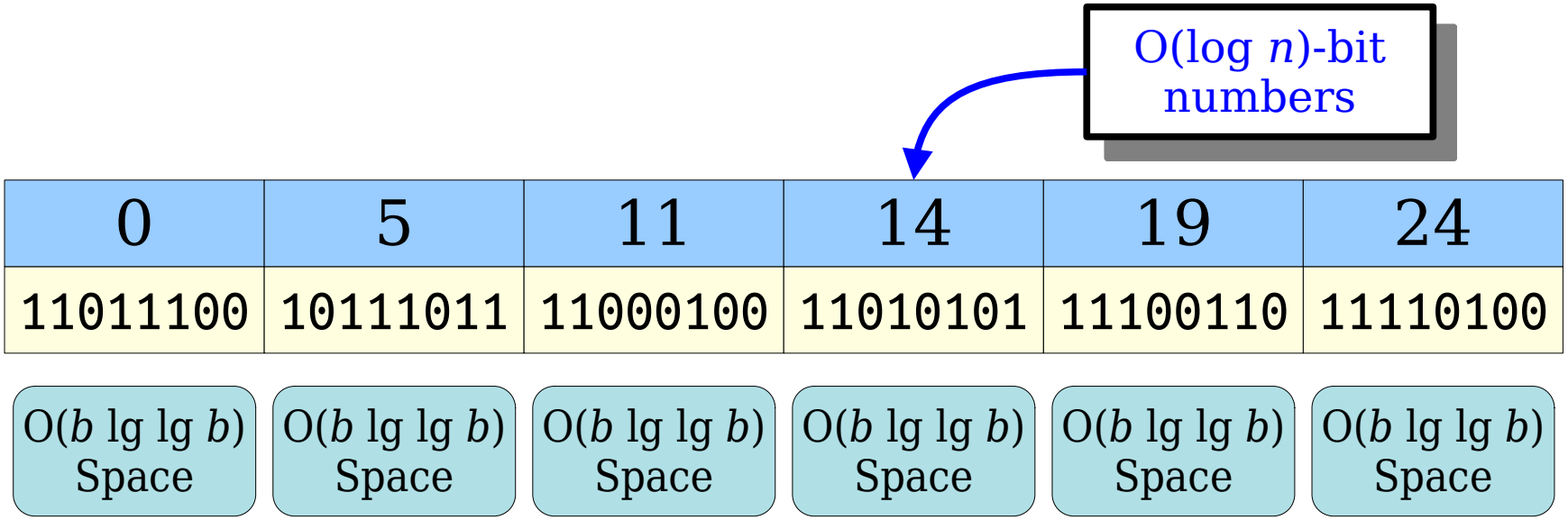
- How much memory does this structure use, and what's the query cost?

Discuss with your neighbors!



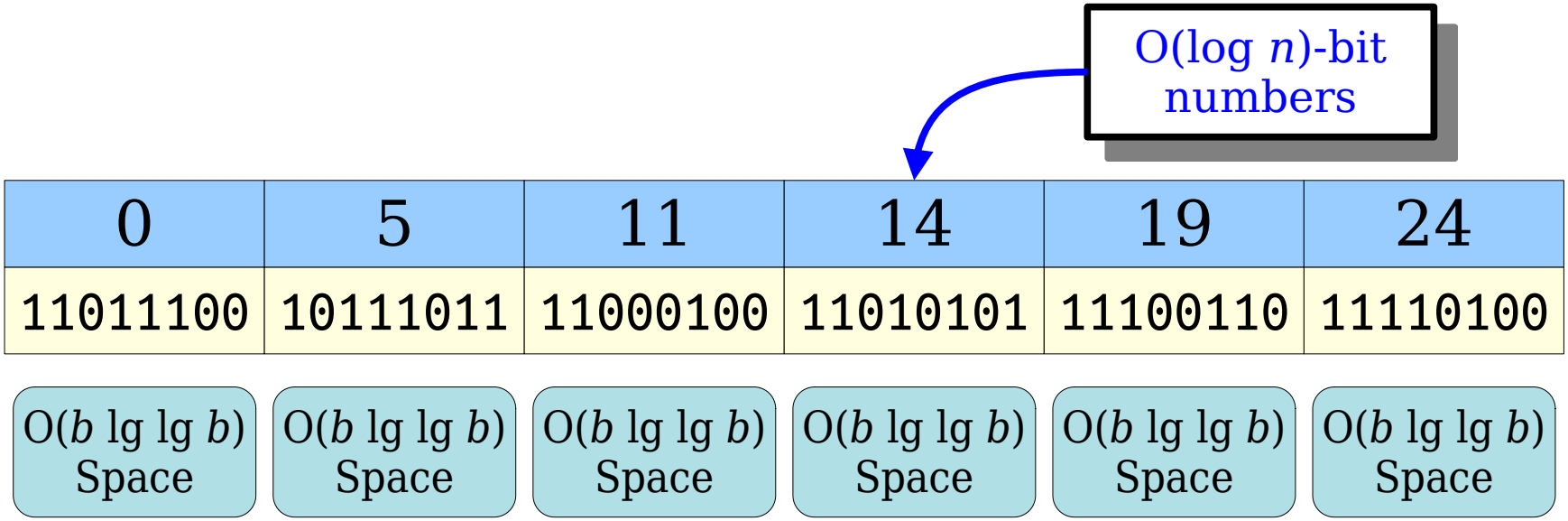
# Feedback Loops

- How much memory does this structure use, and what's the query cost?
- Memory: O((*n* log *n*) / *b* + *n* log log *b*)
- Query Time: O(1)



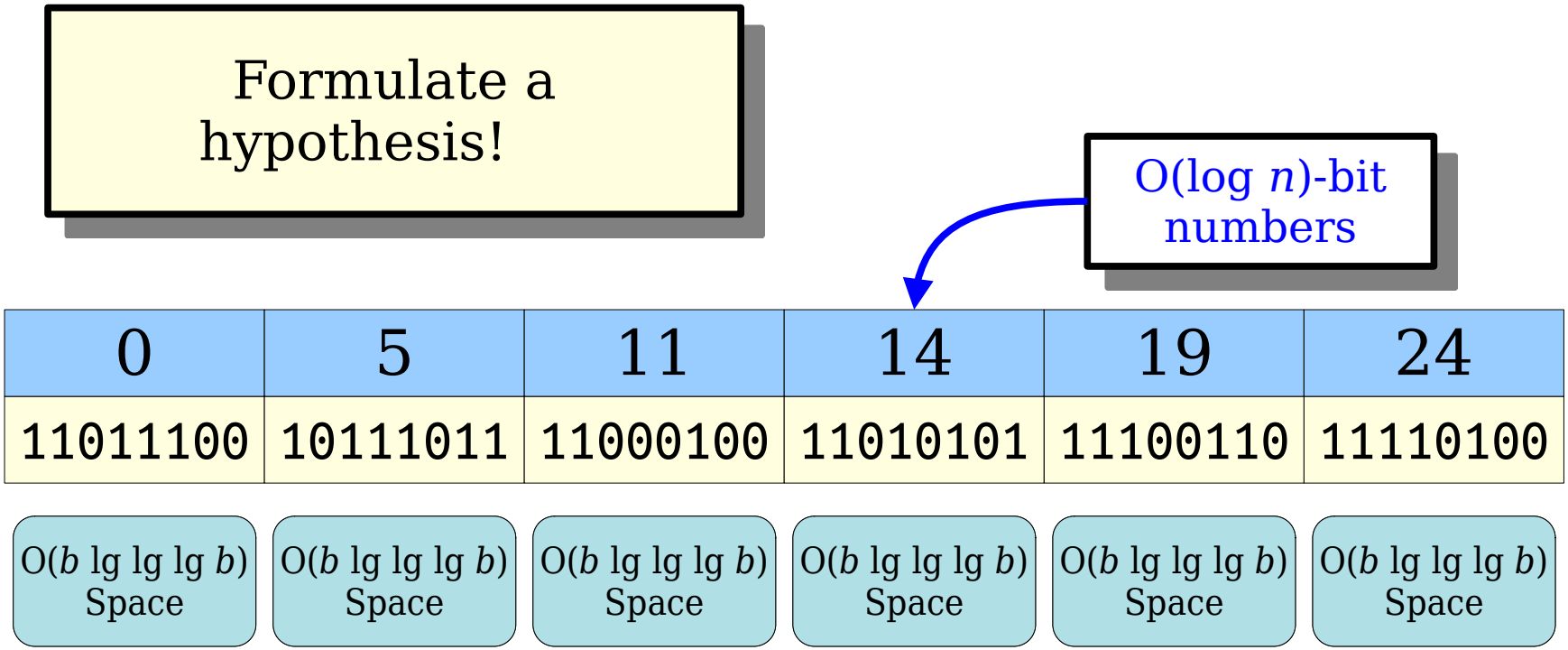
# Feedback Loops

- ***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

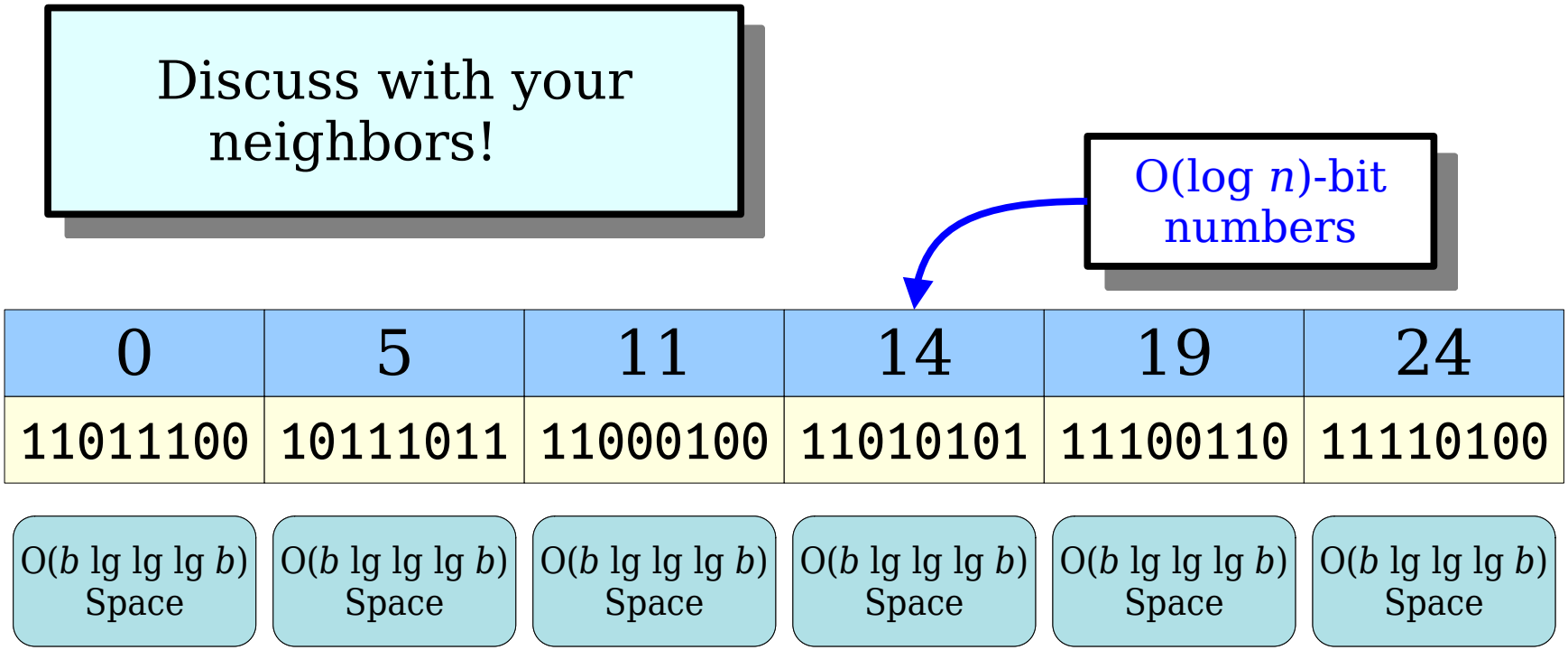
- As you might expect, we can feed this solution back into itself to come up with a $\langle \Theta(n \log \log \log \log n), O(1) \rangle$ solution to ranking.
- More generally, let $\log^{(k)} n$ denote the logarithm function iterated $k$ times.
- ***Question:*** Does this solution allow us to get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



# 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 $k$?

Discuss with your neighbors!



# 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 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1101110010111011110001001101010111001101111011111100 | | | | | | | | | | | | | | |

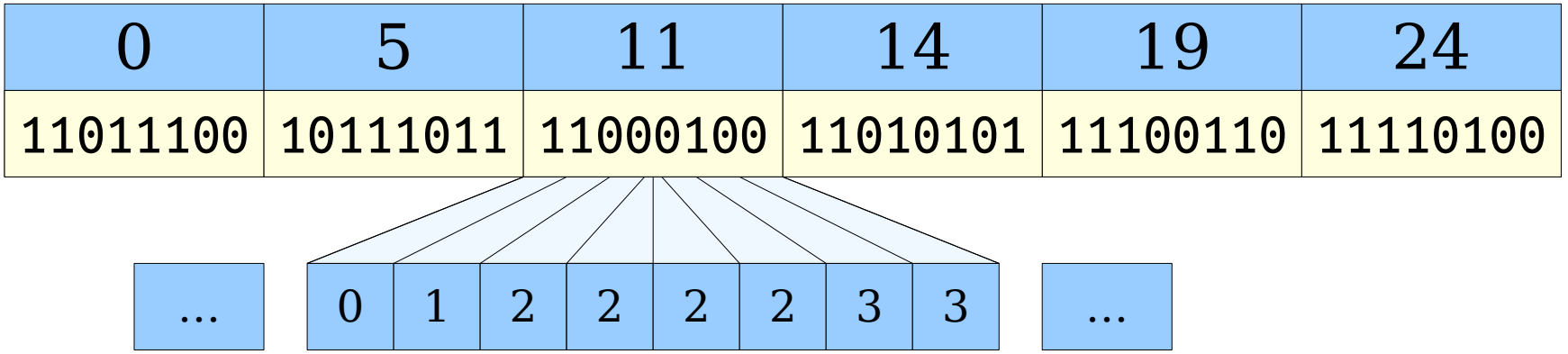
# Counting Layers

- 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.

| 0 | 1 | 2 | 2 | 3 | 4 | 5 | … | 25 | 26 | 27 | 28 | 29 | 29 | 29 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 |

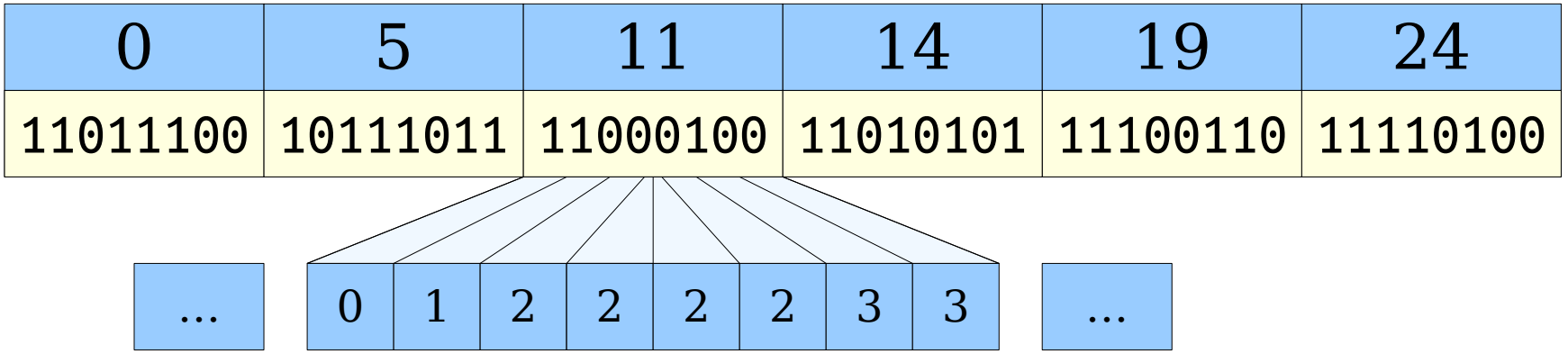
# Counting Layers

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



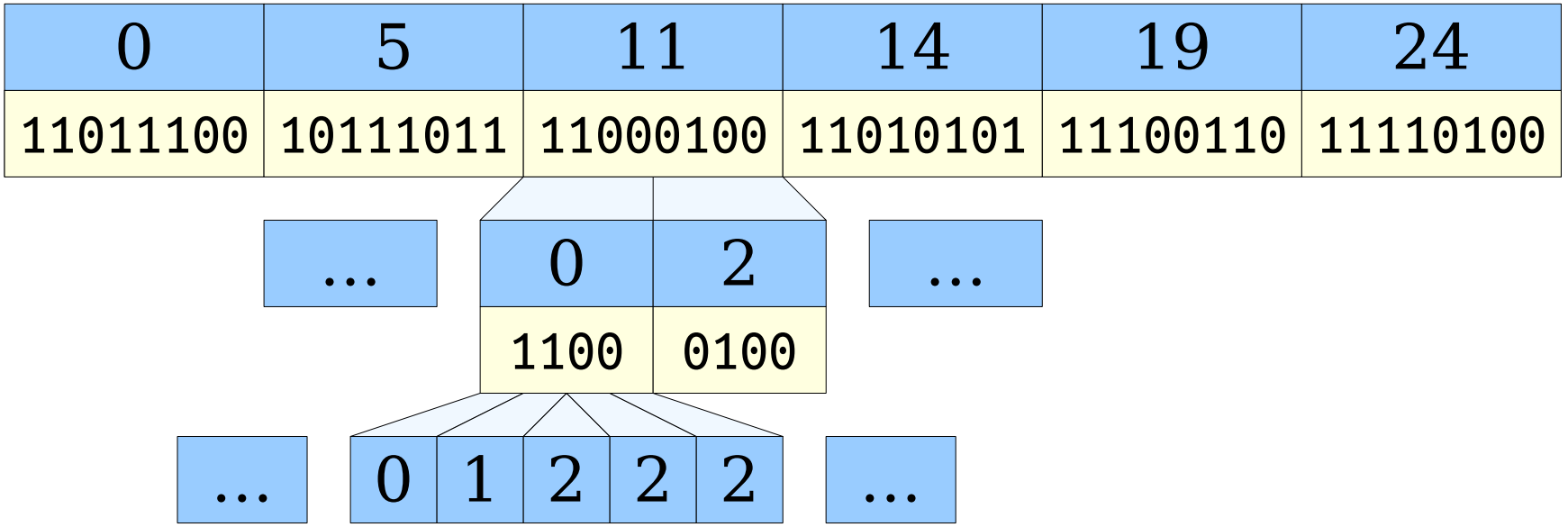
# Counting Layers

- 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.”



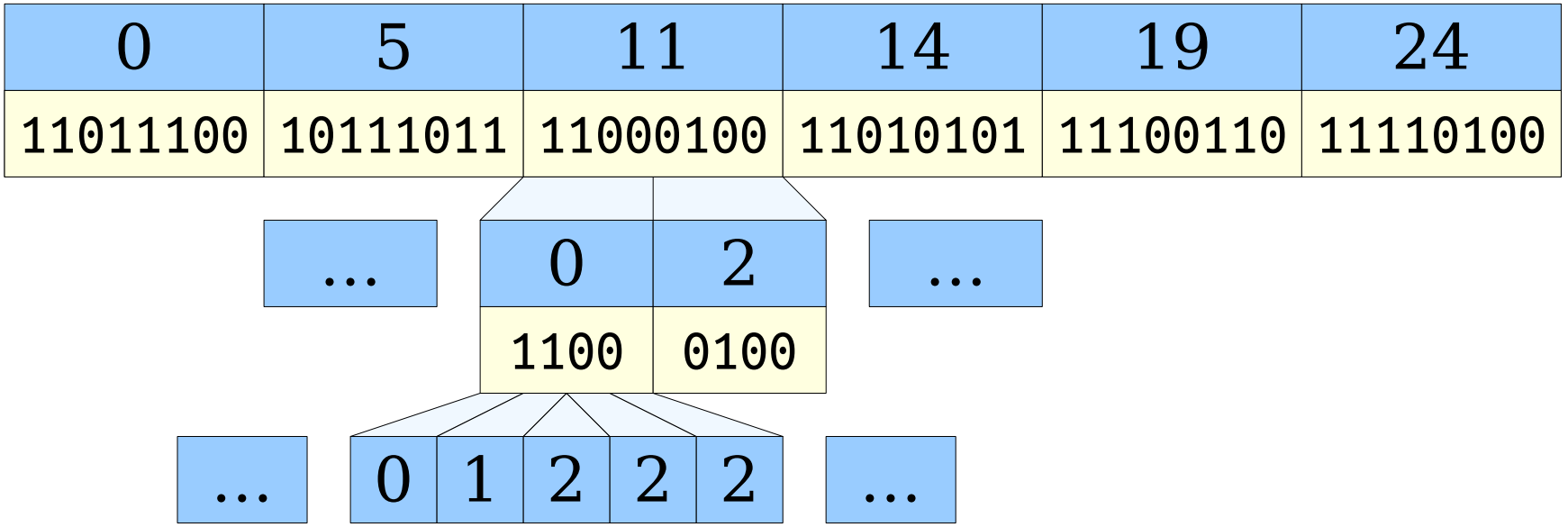
# Counting Layers

- 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.”



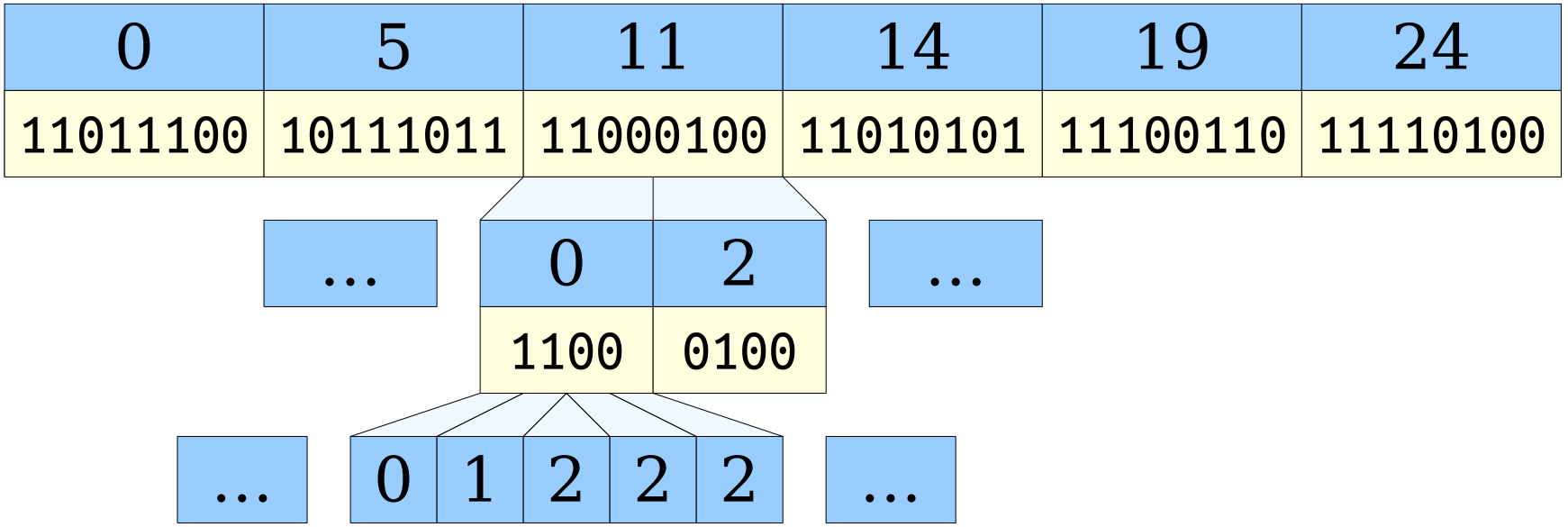
# Counting Layers

- 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.
  - The last layer uses $O(n \log^{(k)} n)$ bits.
- Our query time is $O(k)$, since we have $k$ layers to navigate.



# Counting Layers

- We now have a $\langle O(nk + n \log^{(k)} n), O(k) \rangle$ solution for ranking.
- If $k$ is a fixed constant, this is a $\langle O(n \log^{(k)} n), O(1) \rangle$ 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 ≈5$n$ 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* $n$) |

# 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 |
| *Index* `3` | 0 | 1 | 1 | 2 | 1 | 2 | 2 | 3 |

# 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 |
| *Index* 3 | 0 | 1 | 1 | 2 | 1 | 2 | 2 | 3 |

# The Four Russians Strategy

- Total space: $O(2^b \cdot b \cdot \log b)$.
- Plugging in $\boldsymbol{b = \frac{1}{2} \lg n}$ gives a space usage of
  - $= O(2^{\frac{1}{2} \lg n} \cdot \log n \cdot \log \log n)$
  - $= O(n^{\frac{1}{2}} \log n \log \log n)$
  - $= o(n^{\frac{2}{3}})$.
- 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 |
| *Index* 3 | 0 | 1 | 1 | 2 | 1 | 2 | 2 | 3 |

# 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 |
| *Index* 3 | 0 | 1 | 1 | 2 | 1 | 2 | 2 | 3 |

# The Four Russians Strategy

- To perform a query for the rank sum up to index $k$:
  - Compute $\lfloor k/b \rfloor$ to determine which block $k$ falls in.
  - Use the bits of that block as an index into the secondary table, then look up row $k$ mod $b$.
- Query time: **O(1)**.

| 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 |
| *Index* 3 | 0 | 1 | 1 | 2 | 1 | 2 | 2 | 3 |

# 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 our overall space usage to something that is $o(n)$, something whose space as a fraction of the number of bits decreases as $n$ gets larger.

# 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?

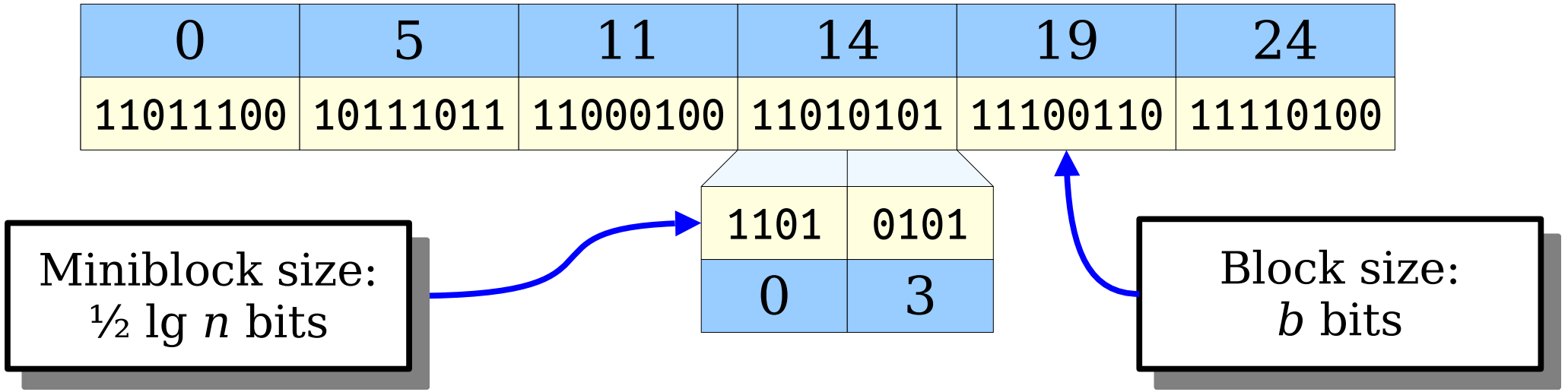
| | Bits Needed | Query Time |
|---|---|---|
| Multilevel Prefix Sums | $O(n \log^* n)$ | $O(\log^* n)$ |
| Four Russians | $O(n)$ | $O(1)$ |

# 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 $n$.

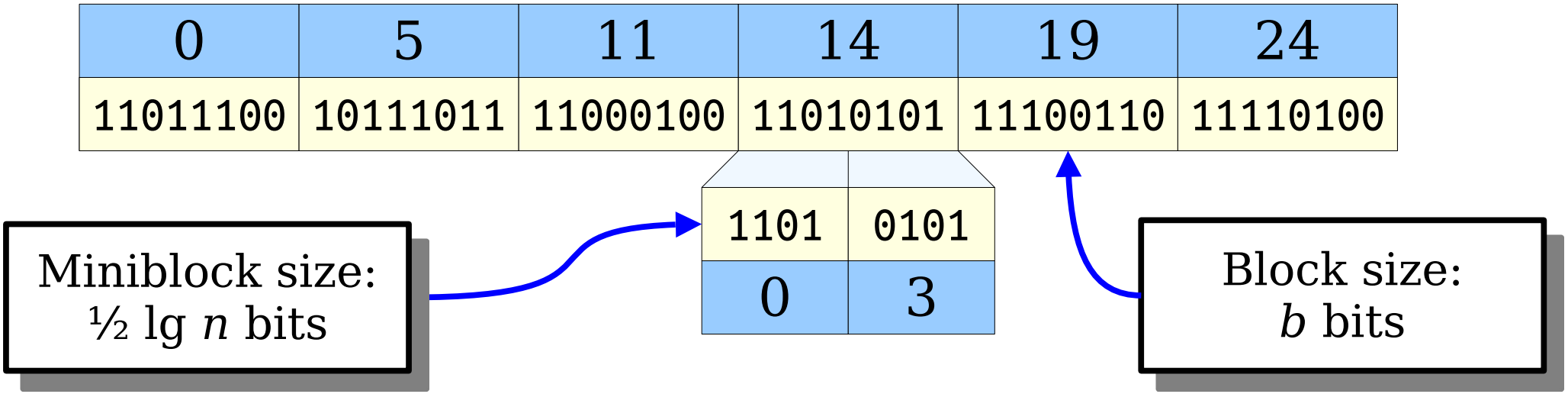
# The Combined Approach

- Subdivide the array into $\Theta(n / b)$ blocks of size $b$.
- Write prefix sums of $O(\log n)$ bits at the start of each block.
- Subdivide each block into $\Theta(b / \lg n)$ miniblocks of size $\frac{1}{2} \lg n$.
- 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.



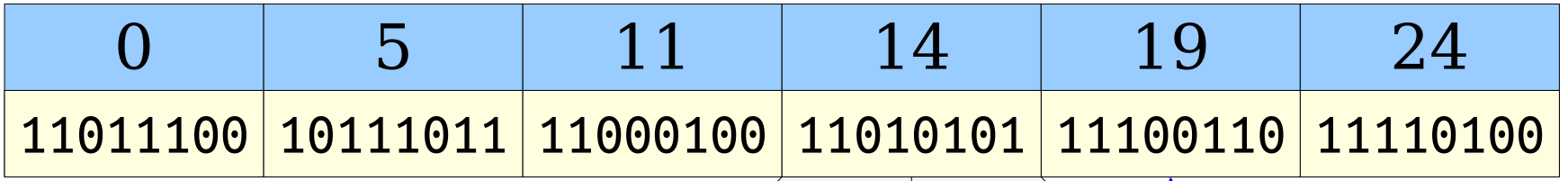
# The Combined Approach

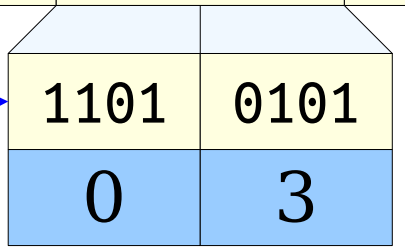
- To perform a query for the prefix sum at index $k$:
  - Divide $k$ by $b$ to get the index of the block containing $k$. Write down the prefix sum at the start of that block.
  - Divide $k$ mod $b$ by ½ lg $n$ to get the index of the miniblock containing $k$. Write down the prefix sum at the start of the miniblock.
  - Look up ($k$ mod $b$) mod 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).



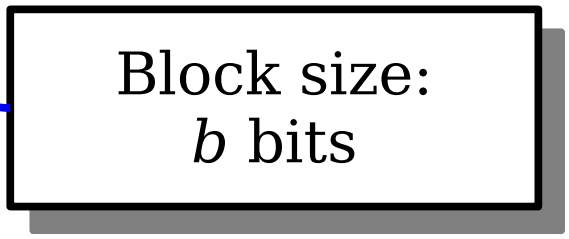
# The Combined Approach

- Space for top-level array: O((*n* log *n*) / *b*).
- Space for the blocks: 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*)**.
- 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:

$$\frac{-n \log n}{b^2} + \frac{n}{b \log n} = 0$$

$$-\log^2 n + b = 0$$

$$b = \log^2 n$$

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

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

# 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) | $O\left(\frac{n \log \log n}{\log n}\right)$ | $O(1)$ |

# 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.
  - So the bitvector itself might use $o(n)$ space!

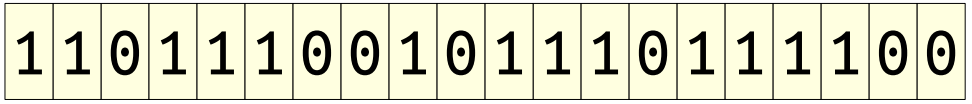
# Succinct Select

# Selection

- The select operation works as follows:

  ***Given an array of bits and a number k, return the index of the kth 1 bit in the array.***

- This is essentially the inverse of the rank operation.
- ***Goal:*** Build a data structure for selection that uses $o(n)$ bits.



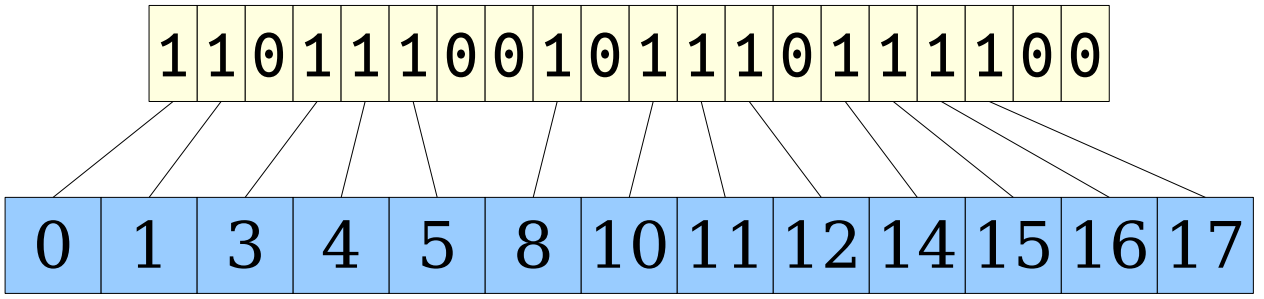
# Adapting Our Techniques

- We have a bunch of techniques at our disposal when going into this problem:
  - Break the input apart into blocks to reduce the number of bits needed to write down indices within blocks.
  - Feed data structures back into themselves to significantly decrease the size of the problem.
  - Once the input is down to a small size, apply the Method of Four Russians: precompute all possible problems and stash them in a table.
- However, our solution is going to be a lot more subtle than the previous one due to some nuances of the nature of select.

# You Gotta Start Somewhere

- ***Initial Idea:*** Form an array containing answers to all possible queries.
- ***Question:*** How much memory does this take?

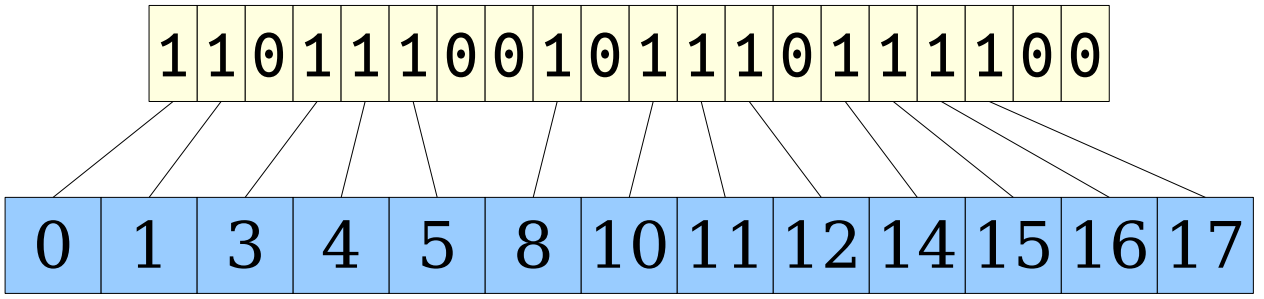
Formulate a hypothesis!



# You Gotta Start Somewhere

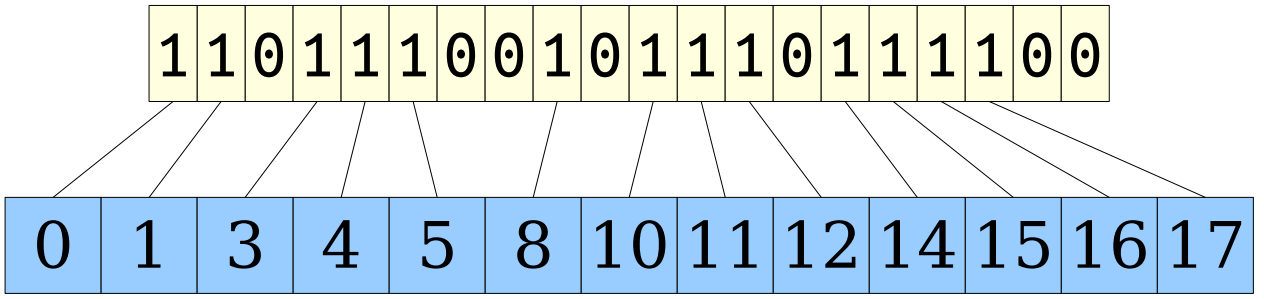
- ***Initial Idea:*** Form an array containing answers to all possible queries.
- ***Question:*** How much memory does this take?

Discuss with your neighbors!



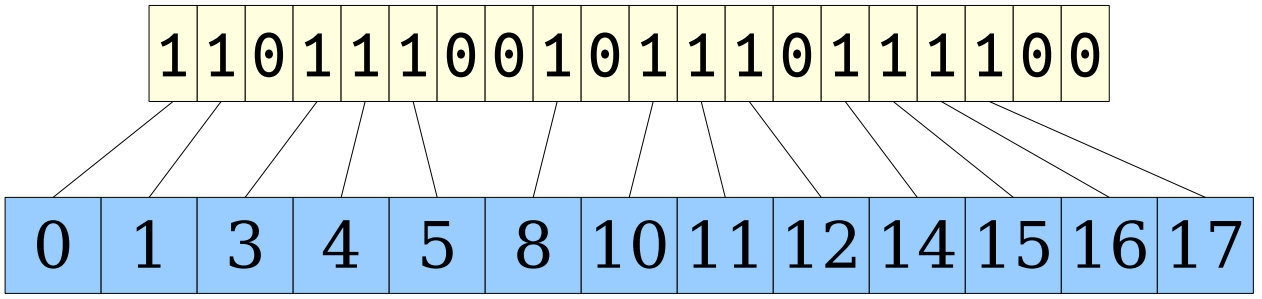
# You Gotta Start Somewhere

- ***Initial Idea:*** Form an array containing answers to all possible queries.
- ***Question:*** How much memory does this take?
- ***Answer:*** It depends on how many 1 bits are in the array.



# You Gotta Start Somewhere

- Let $n$ denote the length of the input array and $m$ denote the number of 1 bits.
- We need $O(m \log n)$ bits for this approach.
  - Each index requires $O(\log n)$ bits; $m$ indices needed.
- If $m = o(n / \log n)$, this is already an $o(n)$-space solution!
- Many practical problems have $m = \Theta(n)$ (e.g. $m = \frac{1}{2}n$), in which case this is a $\Theta(n \log n)$-space solution.
- Can we do better?



# Blocked on Blocking

- In the case of rank, our first step was to break the input apart into blocks.
- That worked nicely because

  $$\textbf{rank(0, } k\textbf{) = rank(0, } r\textbf{) + rank(}r\textbf{, } k\textbf{)}$$

  holds for any $0 \leq r \leq k$.
- This lets us break up the input at regular boundaries to get nicely-shaped subproblems.
- The formula given above, however, doesn't work for select. Is there an analog that does?

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

# Blocked on Blocking

- Let select($r$, $k$) denote the index of the $k$th 1 bit that appears at or after index $r$.
- For any $0 \leq s \leq k$, we have the following:

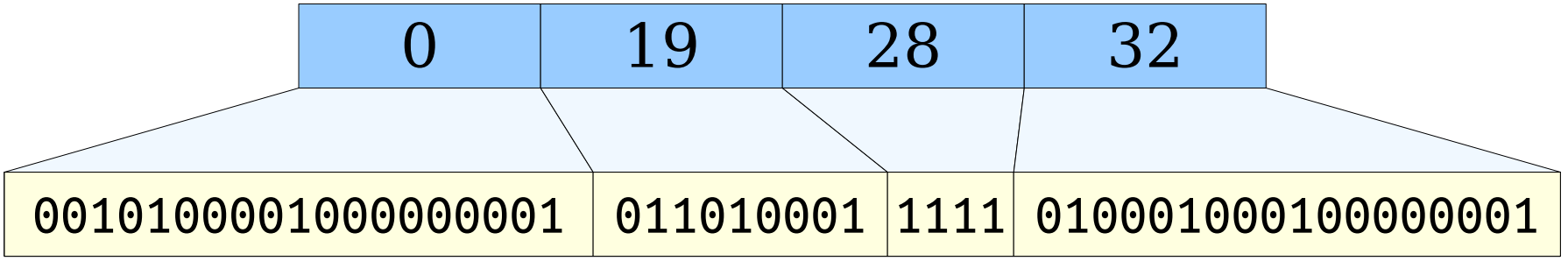
  **select(0, $k$) = select(select(0, $s$), $k$ – $s$).**

- This allows us to break the original problem apart into (uneven-size) smaller subproblems by splitting at positions of individual 1 bits.

```
00101000010000000010110100011111010001000100000000100
```

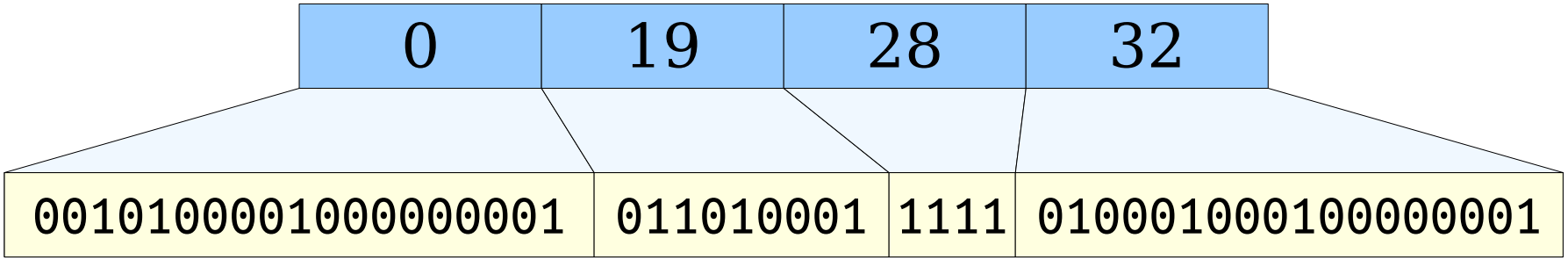
# The Chunking Strategy

- ***Idea:*** Pick a "chunk size" $c$, then break the input into $O(n / c)$ chunks by splitting at every $c$th 1.
  - These chunks may have uneven numbers of bits.
  - Note that there might not be $\Theta(n / c)$ chunks. *(Why?)*
- Store the starting index of each chunk in a summary array. This uses $O((n \log n) / c)$ bits as each index needs $O(\log n)$ bits.
- To compute select($k$), do the following:
  - Compute $k / c$ to determine which chunk to look in.
  - Look within that chunk for the ($k$ mod $c$)th 1 bit.
- How do we do that second step?



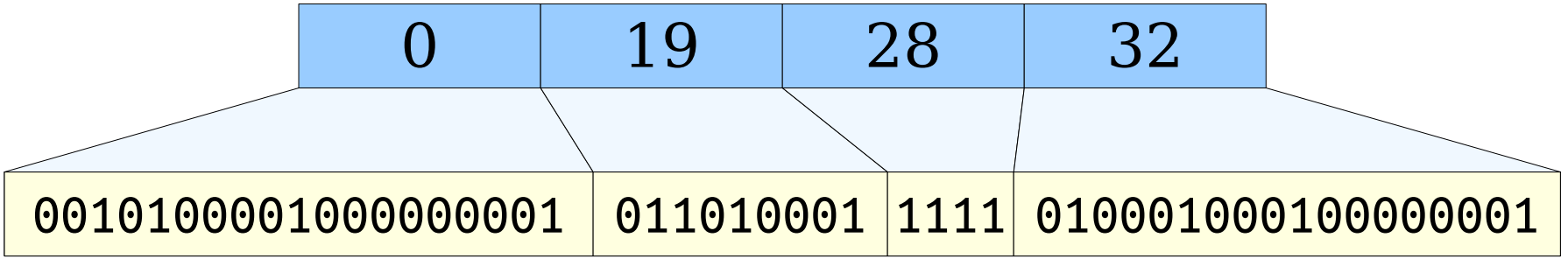
# The Chunking Strategy

- Because our chunks don’t have uniform size, doing a linear scan within the chunk will not necessarily take time O($c$).
  - $c$ counts how many 1 bits are in the chunk, not how many *total* bits are in the chunk.
- Because our chunks don’t have uniform size, the index of the 1 bits within each block doesn’t necessarily use space O(log $c$).
  - The chunk size might be as big as $\Theta(n)$.
- Therefore, our earlier tricks from rank aren’t going to work here. We need to find a different strategy.



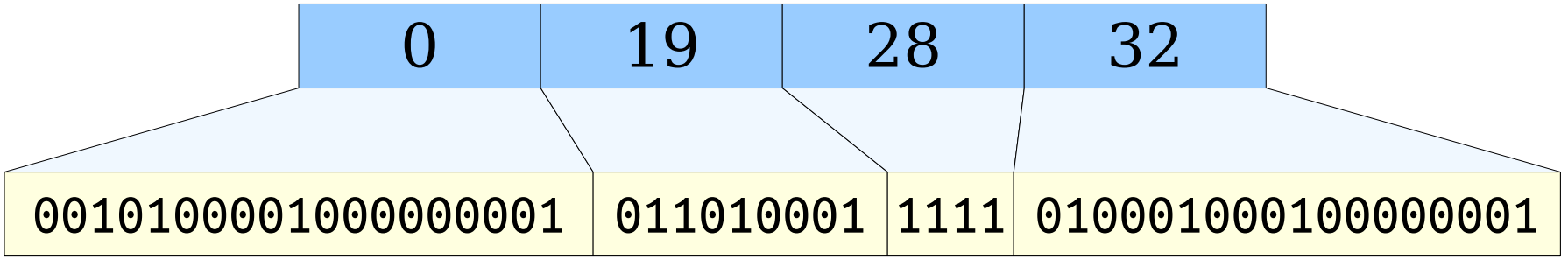
# Small/Large Decomposition

- ***Key Insight:*** Pick a number $L$ and categorize chunks as follows.
  - ***Small chunks*** are ones with fewer than $L$ bits.
  - ***Large chunks*** are ones with at least $L$ bits.
- Intuitively, we'll handle the chunks as follows.
  - There can't be "too many" large chunks in the array. That will bound the cost of dealing with them.
  - All small chunks have a bounded size. From there, we can use our earlier techniques (linear scans, recursion, Four Russians, etc.) to handle them.



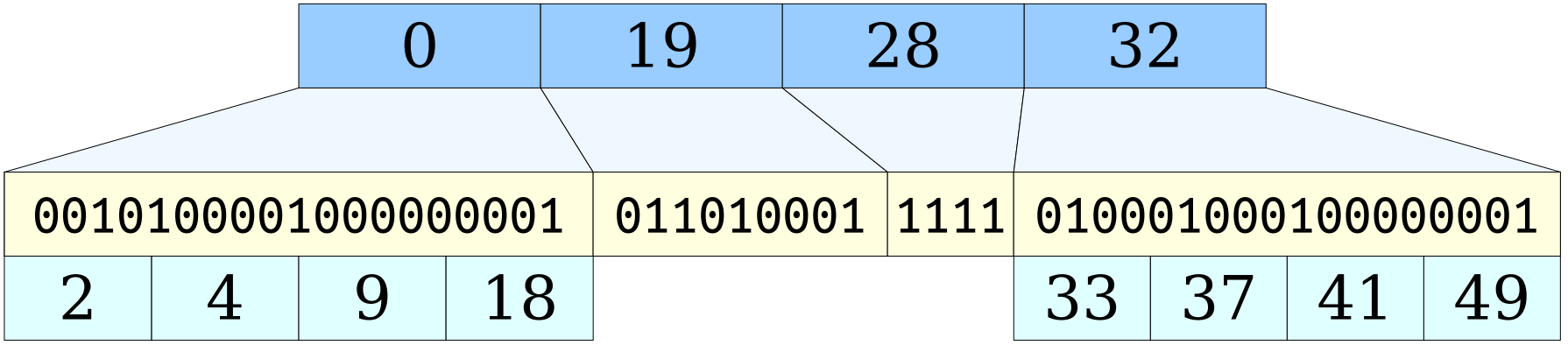
# Small/Large Decomposition

- Here's a framework for performing select($k$):
  - Compute $i = \lfloor k/c \rfloor$, the index of the chunk our bit belongs to.
  - Compare the start positions of chunks $i$ and $i+1$ to determine how many bits are in chunk $i$. Denote this as $r$.
    - If $r \geq L$, use `[insert large strategy here]` to determine the position of the $(k \bmod c)^{\text{th}}$ 1 bit within the (large) chunk.
    - If $r < L$, use `[insert small strategy here]` to determine the position of the $(k \bmod c)^{\text{th}}$ 1 bit within the (small) chunk.
  - Add that bit position to the position stored at the top of chunk $i$.
- We need to determine how to pick $L$ and $c$, as well as what the small and large strategies are.



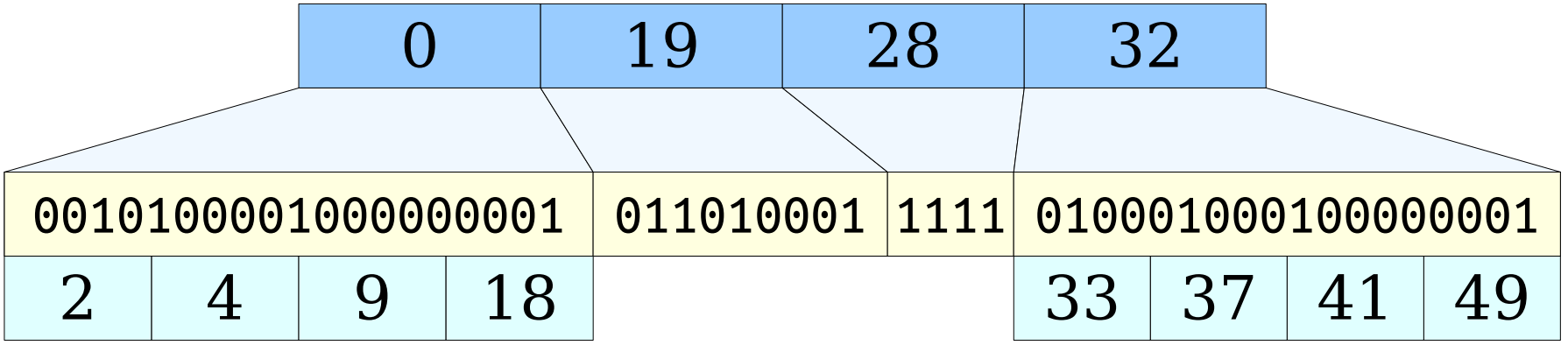
# Handling Large Chunks

- Large chunks have size at least $L$, and there's no upper bound on their size.
  - The index of a 1 bit in a large chunk might require $\Theta(\log n)$ bits.
- However:
  - There can't be that many large blocks. Specifically, there's at most $n / L$ of them. *(Why?)*
  - There aren't that many 1 bits inside a large block. Specifically, there's at most $c$ such bits.
- ***Idea:*** For large chunks, just write down the positions of the 1 bits in the chunk. Then, tune $L$ relative to $c$ to reduce space usage.



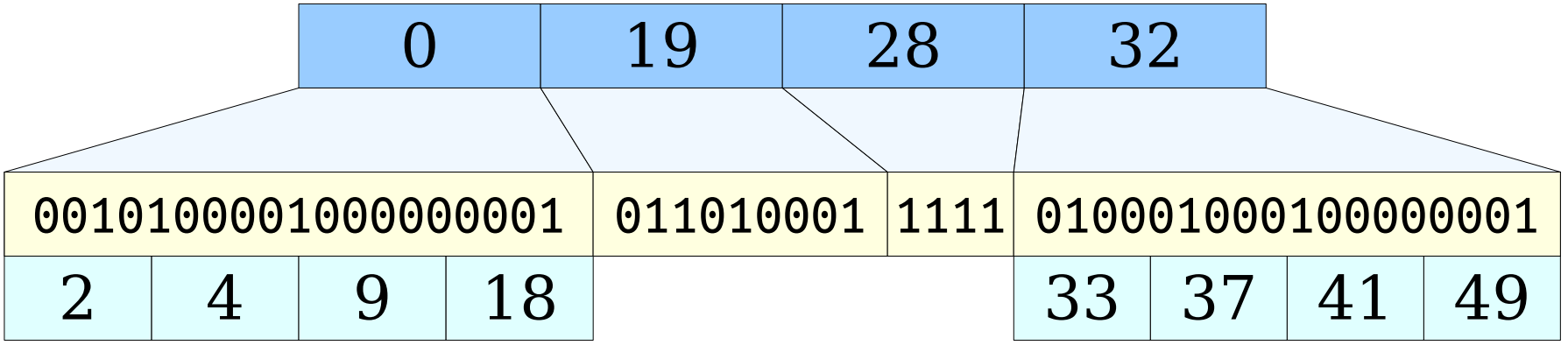
# Handling Large Chunks

- Suppose we write down the positions of the 1 bits within each large block. How many bits of memory does this take?
- ***Answer:*** O((*cn* log *n*) / *L*) bits.
  - There are at most *n* / *L* large blocks.
  - Each large block has *c* 1 bits whose indices must be recorded.
  - Each index into a large block uses O(log *n*) bits.
- Combined with the space for the top-level array, this uses O((*n* log *n*) / *c* + (*cn* log *n*) / *L*) bits.



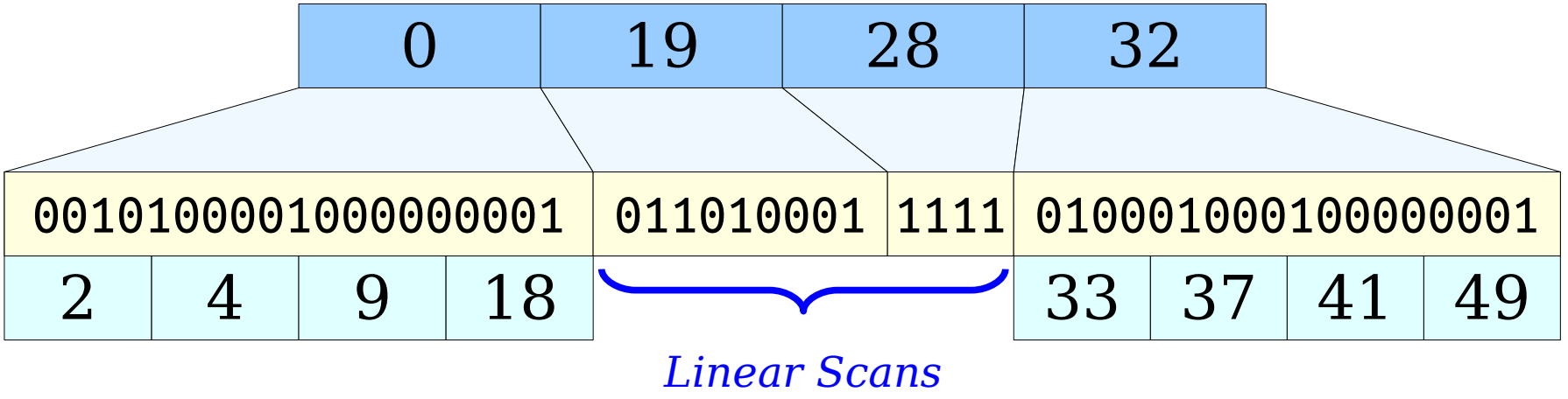
# Optimizing $O\left(\frac{n \log n}{c} + \frac{cn \log n}{L}\right)$

- This quantity is (asymptotically) minimized when the two fractions are (asymptotically) equal.
- This happens when $\boldsymbol{L = \Theta(c^2)}$, in which case the space usage is $\boldsymbol{O((n \log n) / c)}$.



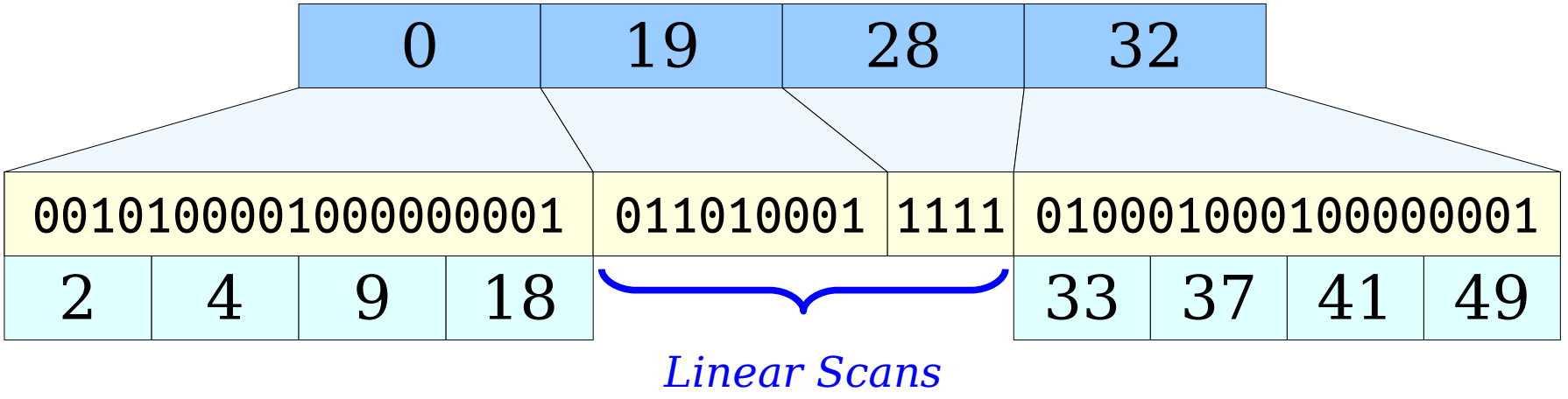
# Handling Small Chunks

- We still need to handle small chunks, which now all have size at most $c^2$.
- ***Initial Idea:*** Use linear scans within those chunks. Each small chunk has size at most $c^2$, giving a query time of $O(c^2)$.



# Putting it All Together

- Split the input into chunks of $c$ bits each.
- For each large chunk containing at least $c^2$ bits, write down the position of each 1 bit in the chunk.
- For each small chunk containing at most $c^2$ bits, use a linear scan within the chunk.
- This gives a $\langle O((n \log n) / c), c^2 \rangle$ solution to selection.



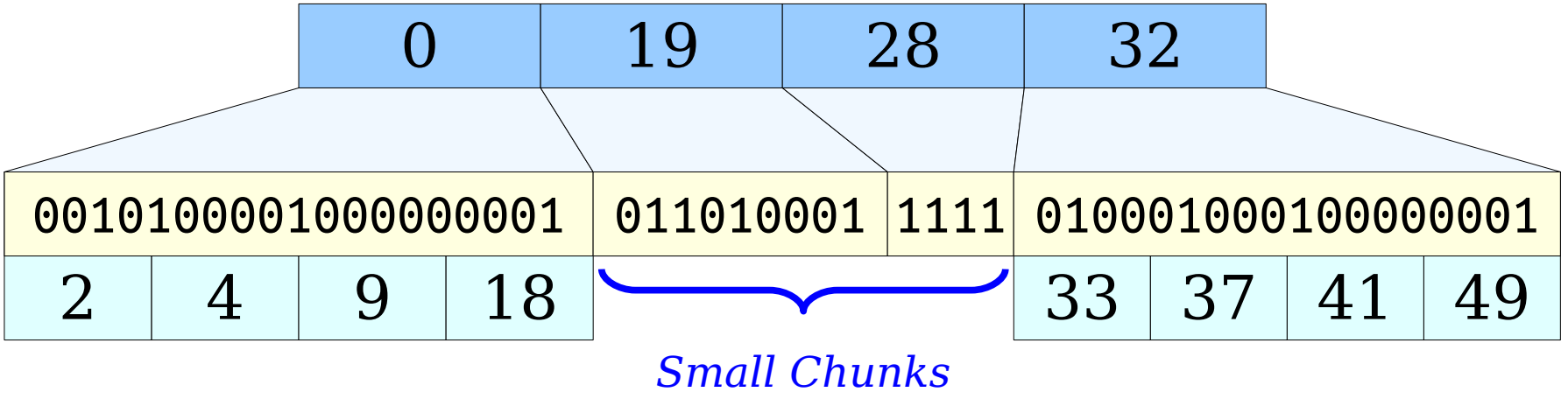
# The Story So Far

- By tuning $c$, we can get sublinear space usage, though at a cost to query time.
- The query time here isn't great because we're using linear scans within small chunks.
- What happens if we pull out more powerful techniques to handle small chunks?

| | Bits Needed | Query Time |
|---|---|---|
| Precomputed Array | $O(n \log n)$ | $O(1)$ |
| Small/Large w/Linear Scans | $O\left(\frac{n \log n}{c}\right)$ | $O(c^2)$ |

# Improving Our Approach

- Our small chunks, as the name suggests, don't have too many bits in them.
  - Specifically, at most $c^2$, where we get to pick $c$.
- ***Idea:*** If the small chunks are sufficiently small, there won't be "too many" possible distinct small chunks, and we can use a Four Russians speedup.
- This would drop our query time to O(1):
  - For large chunks, we explicitly store answers to select queries.
  - All small chunk select queries are already precomputed.



# Improving Our Approach

- For example, suppose our small chunks all have 3 bits or fewer.
- We could precompute a table like the one shown below that encodes all possible select queries.
- With chunk size $c$, small chunks have at most $c^2$ bits, and the table needs $O(2^{c^2} c^2 \log c)$ bits.
- Setting $c = \frac{\sqrt{\lg n}}{2}$ makes the above expression $o(n^{1/2})$, a sublinear number of bits.

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

# The Story So Far

- Directly using Four Russians gives constant query times, but uses superlinear ($\omega(n)$) space.
- Can we do better?

| | Bits Needed | Query Time |
|---|---|---|
| Precomputed Array | $O(n \log n)$ | $O(1)$ |
| Small/Large w/Linear Scans | $O\left(\frac{n \log n}{c}\right)$ | $O(c^2)$ |
| Small/Large w/Four Russians | $O\left(n \sqrt{\log n}\right)$ | $O(1)$ |

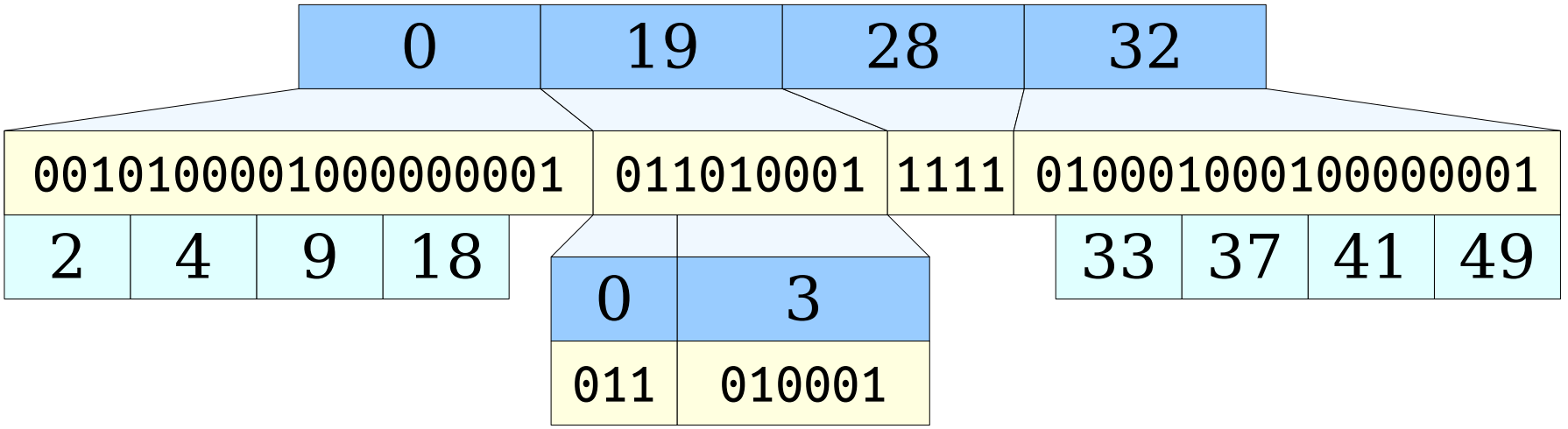
# Improving Small/Large

- Think back to our ***rank*** data structure.
- Our first attempt to use a Four Russians speedup used a two-level structure with a block size of ½ lg $n$.
- We reduced the space usage further by using two layers of blocking.
- Can we do that here?

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

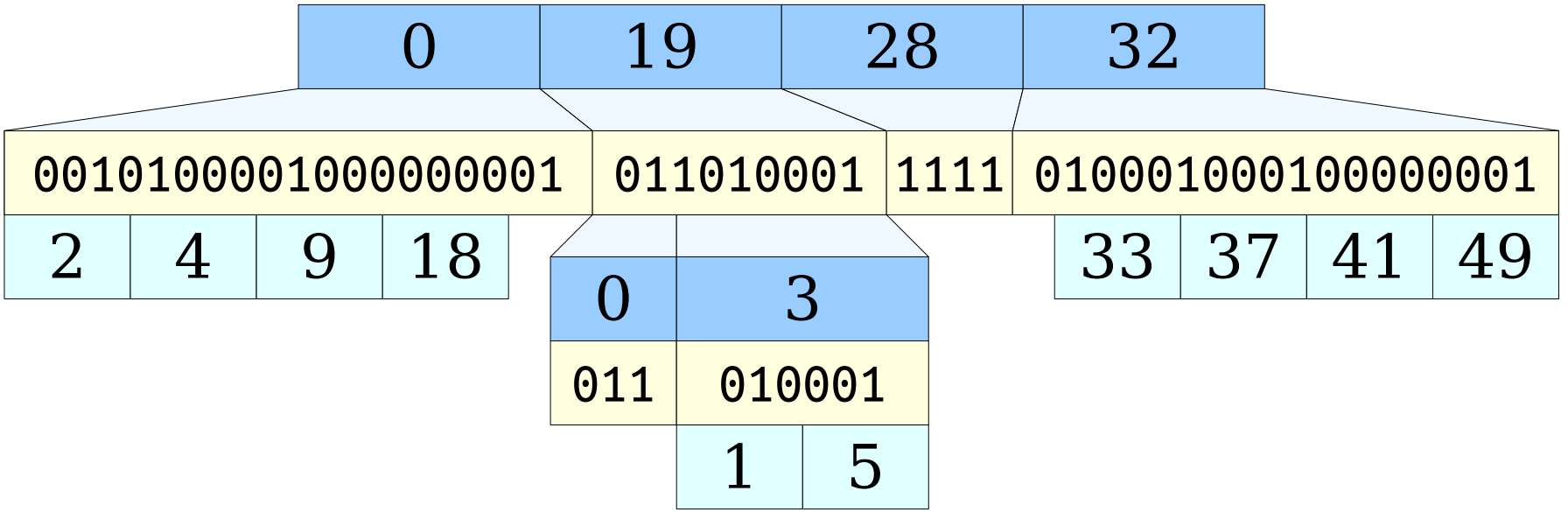
# Improving Small/Large

- ***Idea:*** Recursively apply the chunking construction one more time.
- Pick a "minichunk" size $c_2$. Split each small chunk into minichunks by chopping at each ($c_2$)th 1 bit.
- Write down each minichunk's index relative to its chunk.
- As before, call a minichunk ***large*** if it has at least $(c_2)^2$ bits, and ***small*** if it has fewer than $(c_2)^2$ bits.
- As before, we can choose any strategies we want for handling large and small miniblocks.



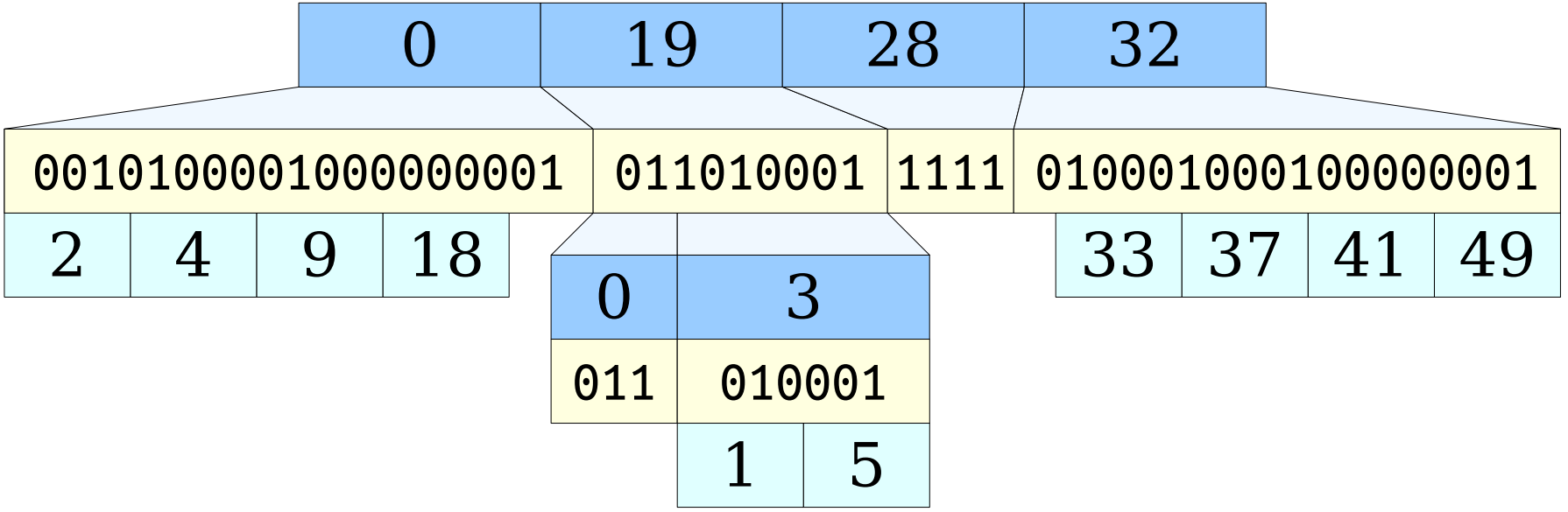
# Handling Large Minichunks

- As with large chunks, we'll handle large minichunks by writing out the indices of all 1 bits in the minichunk.
- This will use a very small amount of space; more on that later.



# Handling Small Minichunks

- A small minichunk has at most $(c_2)^2$ bits.
- If we pick $c_2$ such that $(c_2)^2 \leq \frac{\sqrt{\lg n}}{2}$, we can precompute all possible minichunks and all select queries in them.
- We'll therefore pick $\boldsymbol{c_2 = \frac{\sqrt[4]{\lg n}}{2}}$.



# Improving Small/Large

- If you work out the details on the space usage, you'll find that it comes out to

$$\mathrm{O}\left(\frac{n \log n}{c} + \frac{n \log c}{\sqrt[4]{\log n}}\right).$$

- After a bit of tinkering, you can find that picking $\boldsymbol{c = \Theta(\log^2 n)}$ does a good job balancing these two terms.
- This makes the space usage work out to the (pleasantly confusing)

$$\mathbf{O}\left(\frac{\boldsymbol{n \log \log n}}{\sqrt[4]{\boldsymbol{\log n}}}\right).$$

# The Final Scorecard

- We now have a sublinear-space implementation of select!
- Using more advanced techniques, it's possible to improve this further to $O(n / \log^k n)$ space with query time $O(k)$ for any constant $k$.

| | Bits Needed | Query Time |
|---|---|---|
| Precompute-All | $O(n \log n)$ | $O(1)$ |
| Small/Large w/Four Russians | $O\left(n \sqrt{\log n}\right)$ | $O(1)$ |
| Two-Layer Small/Large w/Four Russians (Clark) | $O\left(\frac{n \log \log n}{\sqrt[4]{\log n}}\right)$ | $O(1)$ |

# In Practice

- The approach we just outlined is great in Theoryland, but leaves a lot to be desired in practice.
  - There are some details we glossed over about how to pack all the bits needed for the relevant tables into a small amount of space while still being navigable. This introduces some overhead.
  - With this implementation, $n$ needs to be colossal before the space overhead drops below $n$ bits.
- In practice, other selection structures are used that have lookup costs like O(log log $n$) but which use significantly less space.
- ***(Possibly?) Open Problem:*** Build a simple, practical, succinct selection structure with fast O(1) query costs.

# 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

- ***Integer Data Structures***
  - Storing integers that fit into machine words.
- ***x-Fast Tries***
  - Tries + Cuckoo Hashing
- ***y-Fast Tries***
  - Tries + Cuckoo Hashing + RMQ + Balanced Trees + Amortization