# Succinct Data Structures

## Part Two

# Outline for Today

- ***Recap from Last Time***
  - Where are we, again?
- ***The Binary Selection Problem***
  - Inverting a prefix sum.
- ***Clark's Succinct Select Structure***
  - A clever and nuanced solution to select.

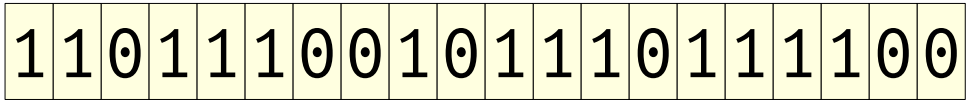
# Recap from Last Time

# 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.



# Jacobson's Rank Structure

- Our succinct solution worked as follows:
  - Split the input into blocks of size $\Theta(\log^2 n)$.
  - Split the blocks into "miniblocks" of size ½ lg $n$.
  - Write down prefix sums at the start of each block and miniblock.
  - Compute a Four Russians table of all possible queries within miniblocks.
- ***Intuition:***
  - Use two levels of recursive subdivision to keep space usage for summaries small.
  - Switch to Four Russians when feasible to avoid extra bits from more summaries.

| | Bits Needed | Query Time |
|---|---|---|
| Two-Level Four Russians (Jacobson's Structure) | $n + o(n)$ | O(1) |

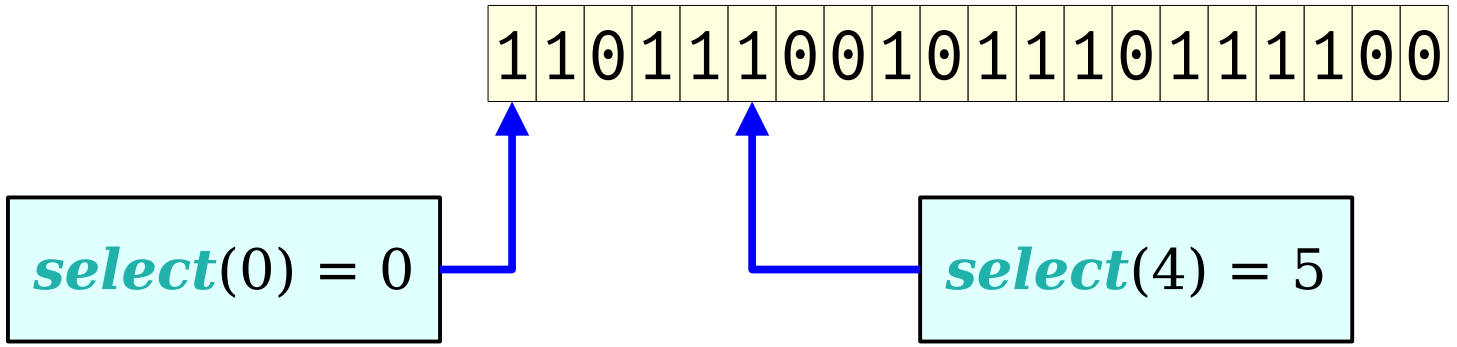
New Stuff!

# Selection

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

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

- For example, ***select***(0) is the index of the first 1 bit in the array, ***select***(1) is the second, etc.



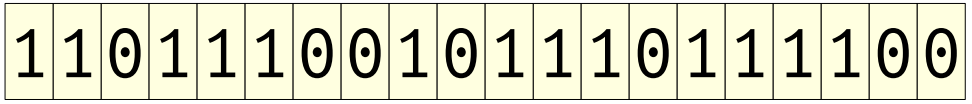
# Selection

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

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

- For example, ***select***(0) is the index of the first 1 bit in the array, ***select***(1) is the second, etc.
- This is a right-inverse of rank:

$$\textit{rank}(\textit{select}(k)) = k$$




***rank***(***select***(5))

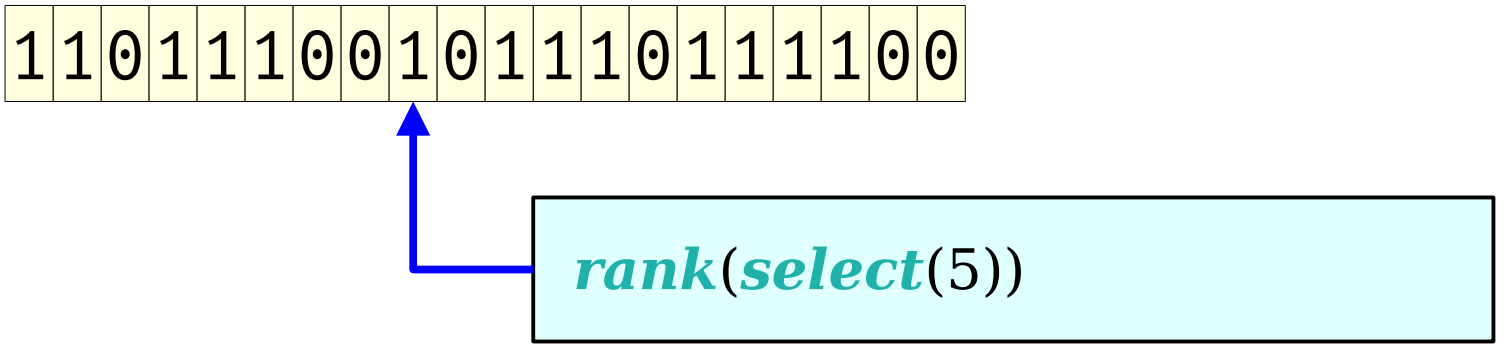
# Selection

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

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

- For example, ***select***(0) is the index of the first 1 bit in the array, ***select***(1) is the second, etc.
- This is a right-inverse of rank:

$$\textit{rank}(\textit{select}(k)) = k$$



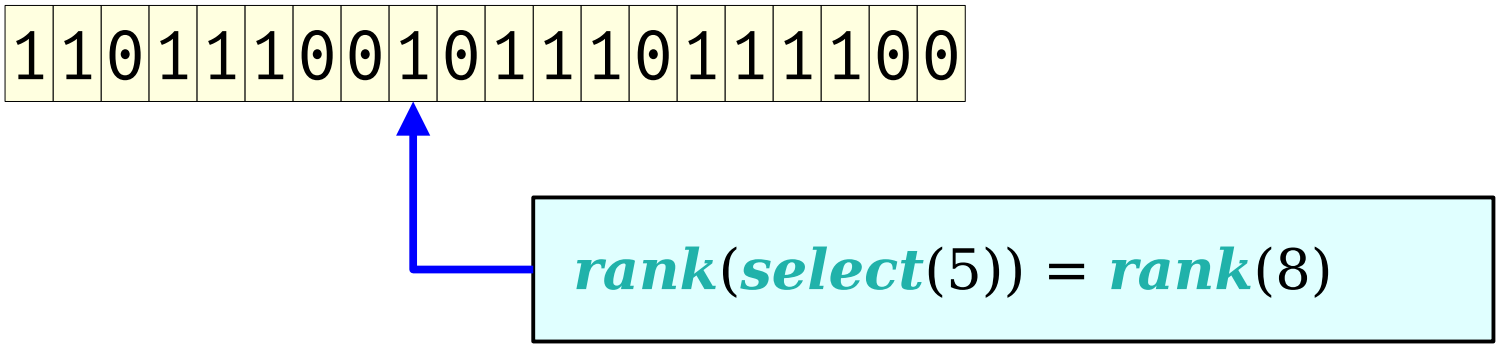
# Selection

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

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

- For example, ***select***(0) is the index of the first 1 bit in the array, ***select***(1) is the second, etc.
- This is a right-inverse of rank:

$$\textit{rank}(\textit{select}(k)) = k$$



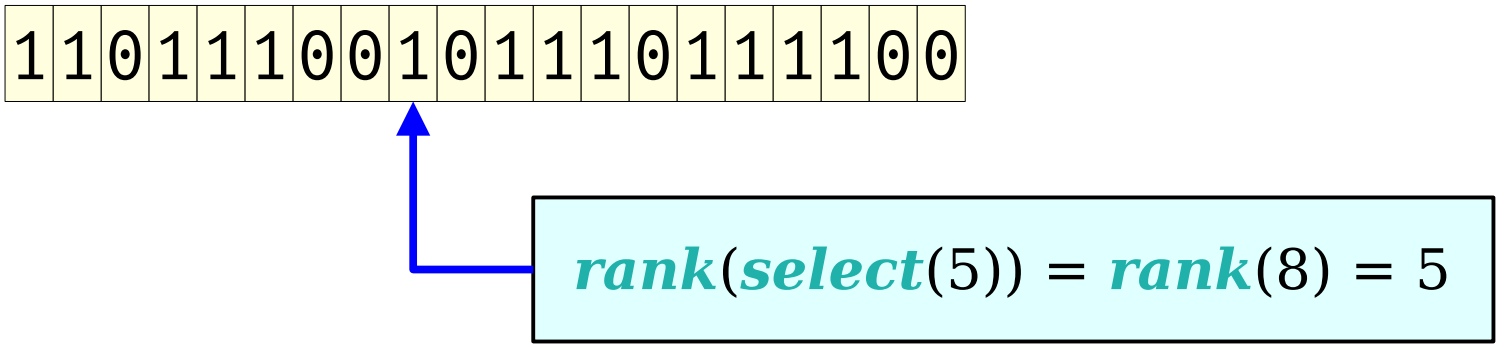
# Selection

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

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

- For example, ***select***(0) is the index of the first 1 bit in the array, ***select***(1) is the second, etc.
- This is a right-inverse of rank:

$$\textit{rank}(\textit{select}(k)) = k$$



# Selection

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

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

- For example, ***select***(0) is the index of the first 1 bit in the array, ***select***(1) is the second, etc.
- This is a right-inverse of rank:

$$\textit{rank}(\textit{select}(k)) = k$$

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

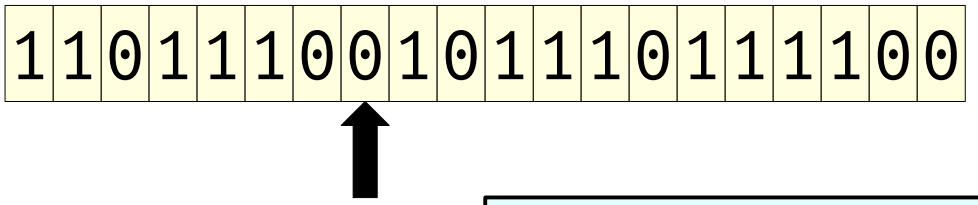
# Selection

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

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

- For example, ***select***(0) is the index of the first 1 bit in the array, ***select***(1) is the second, etc.
- This is a right-inverse of rank:

$$\textit{rank}(\textit{select}(k)) = k$$




***select***(***rank***(7))

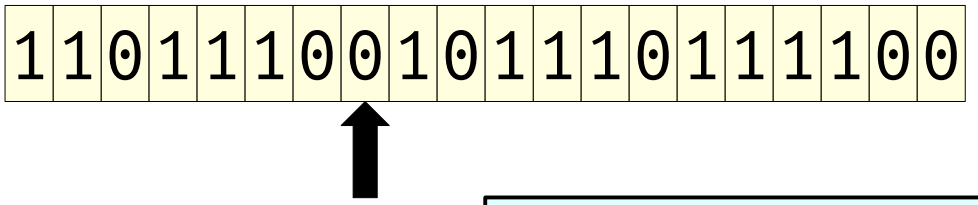
# Selection

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

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

- For example, ***select***(0) is the index of the first 1 bit in the array, ***select***(1) is the second, etc.
- This is a right-inverse of rank:

$$\textit{rank}(\textit{select}(k)) = k$$




***select***(***rank***(7)) = ***select***(5)

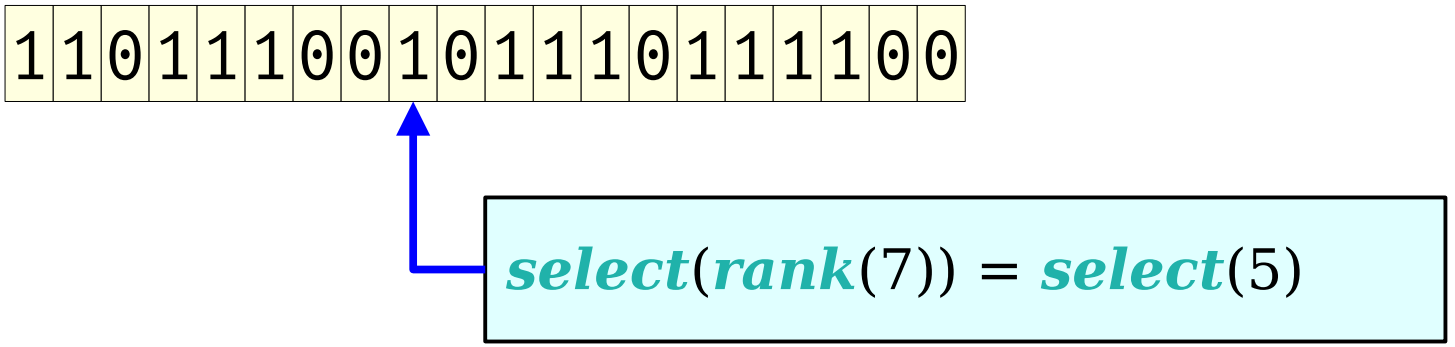
# Selection

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

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

- For example, ***select***(0) is the index of the first 1 bit in the array, ***select***(1) is the second, etc.
- This is a right-inverse of rank:

$$\textit{rank}(\textit{select}(k)) = k$$



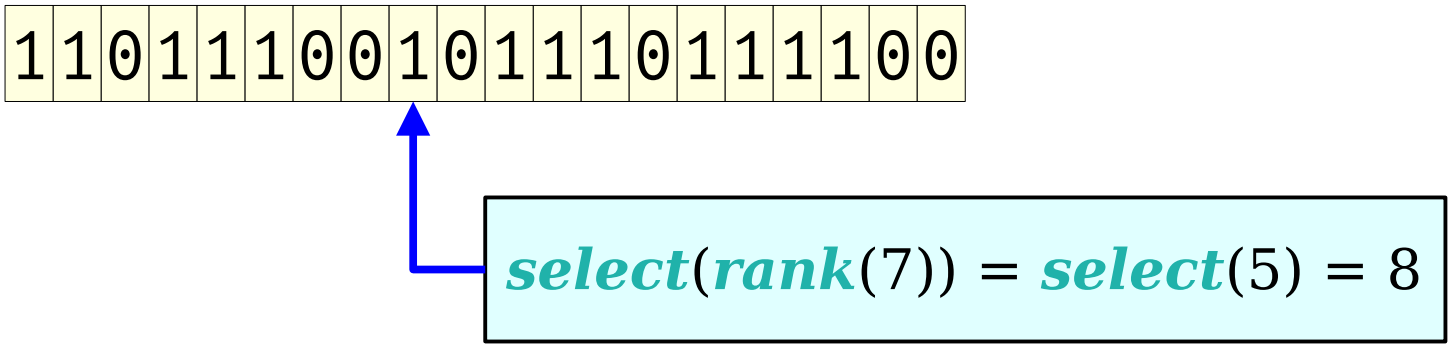
# Selection

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

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

- For example, ***select***(0) is the index of the first 1 bit in the array, ***select***(1) is the second, etc.
- This is a right-inverse of rank:

$$\textit{rank}(\textit{select}(k)) = k$$

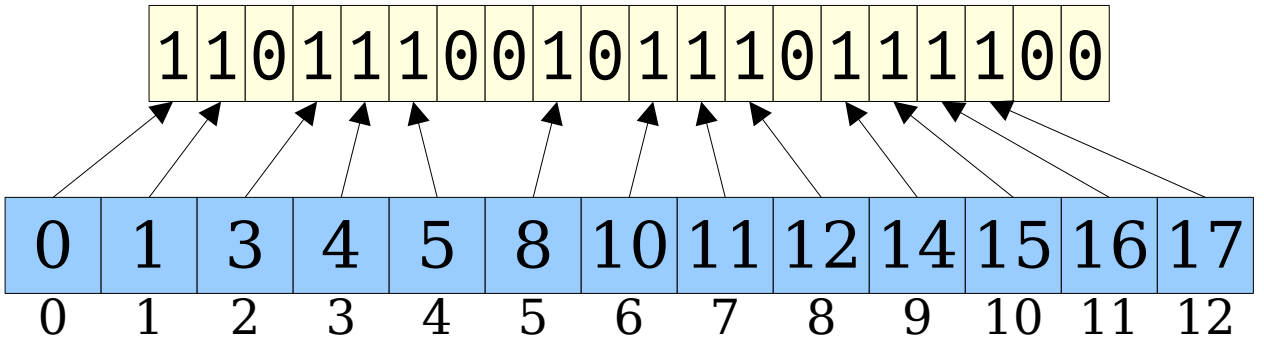


# You Gotta Start Somewhere

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

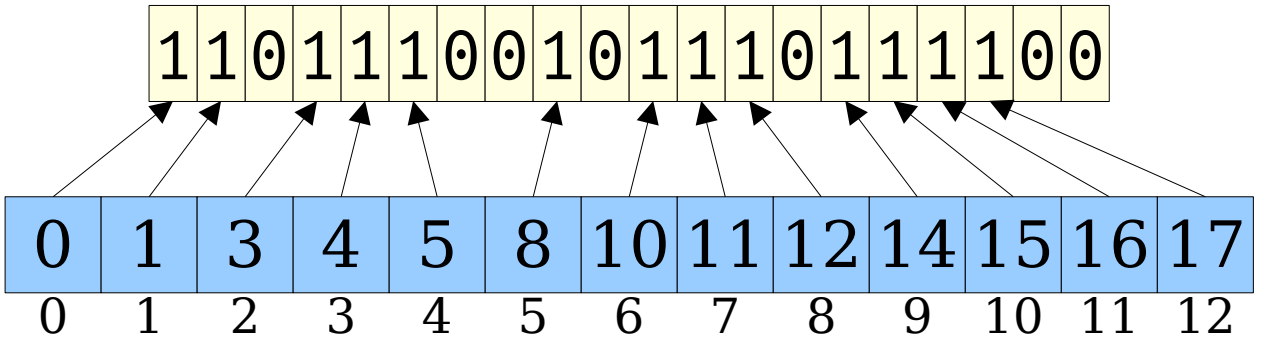
Answer at

***https://cs166.stanford.edu/pollev***



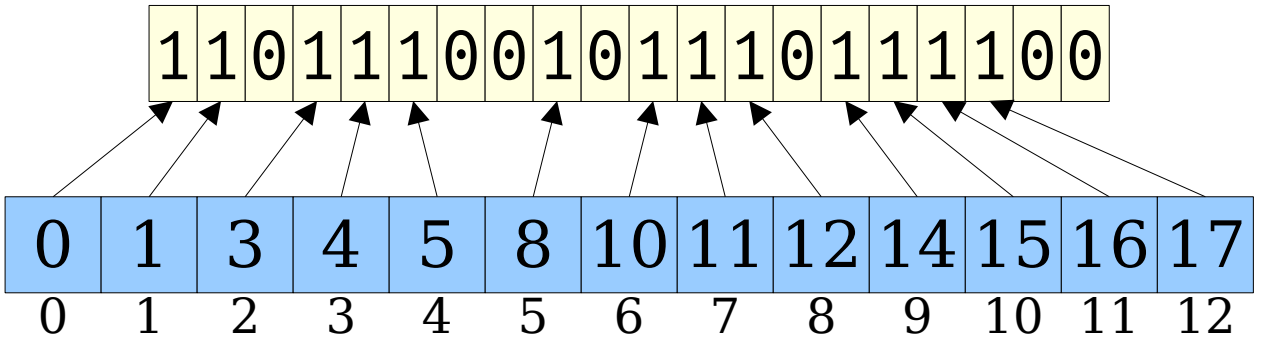
# You Gotta Start Somewhere

- ***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. $m = ½n$), in which case this is a $\Theta(n \log n)$-space solution.
- Can we do better?

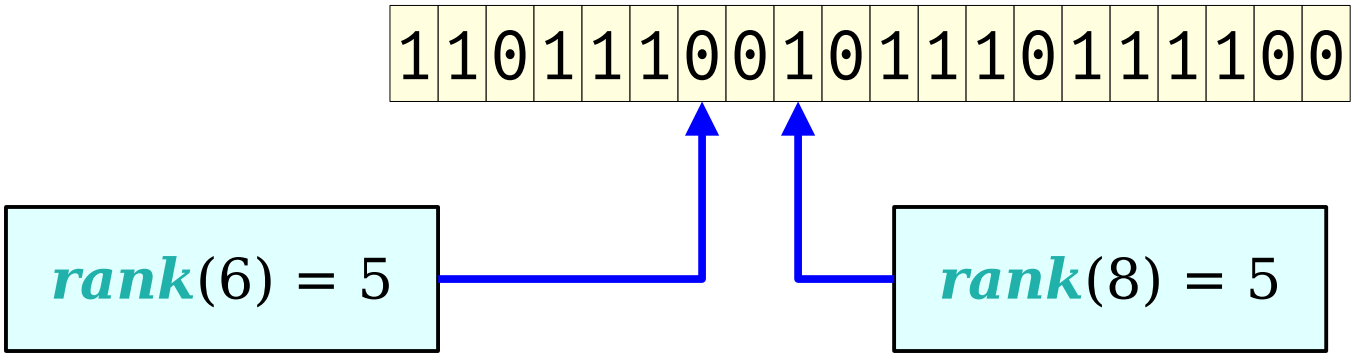


# Another Idea

- ***Recall:*** ***select*** is a right inverse of ***rank***:

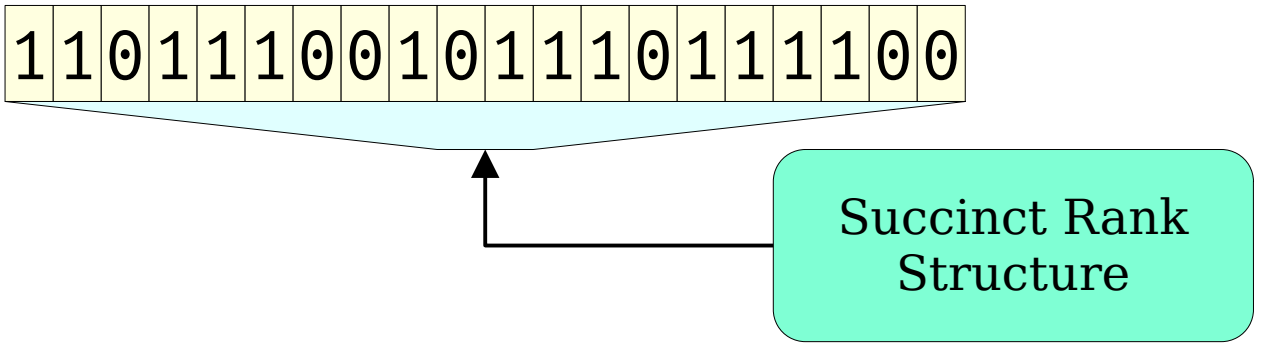
$$\textbf{\textit{rank}}(\textbf{\textit{select}}(k)) = k.$$

- We can compute ***select***$(k)$ by finding the largest index $i$ where ***rank***$(i) = k$.
  - Why do we want the *largest* index?
- How quickly can find that index?



# Another Idea

- ***Idea:*** Get $n + o(n)$ space by using our sublinear-space Jacobson rank structure.
- Build a Jacobson rank structure for the bits.
  - Total space usage: $n + o(n)$.
- To ***select***$(k)$, binary search over indices $i$ to find the largest one where ***rank***$(i) = k$.
  - Query time: O(log $n$).



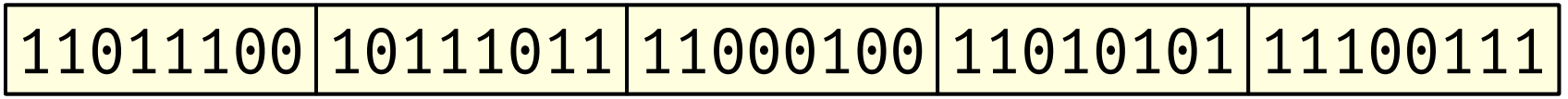
# The Story So Far

- We have a bag of tricks from last time (splitting into blocks, recursively feeding structures into themselves, Four Russians speedups, etc.).
- How might we apply these to binary selection?

| | Bits Needed | Query Time |
|---|---|---|
| Precompute-All | O($m \log n$) | O(1) |
| Binary Search w/Jacobson Rank | $n + o(n)$ | O($\log n$) |

# Subdividing the Problem

- In both RMQ and succinct rank, our solution involved splitting the input into O($n$ / $b$) blocks of size $b$.
- Can we use that idea here as well?



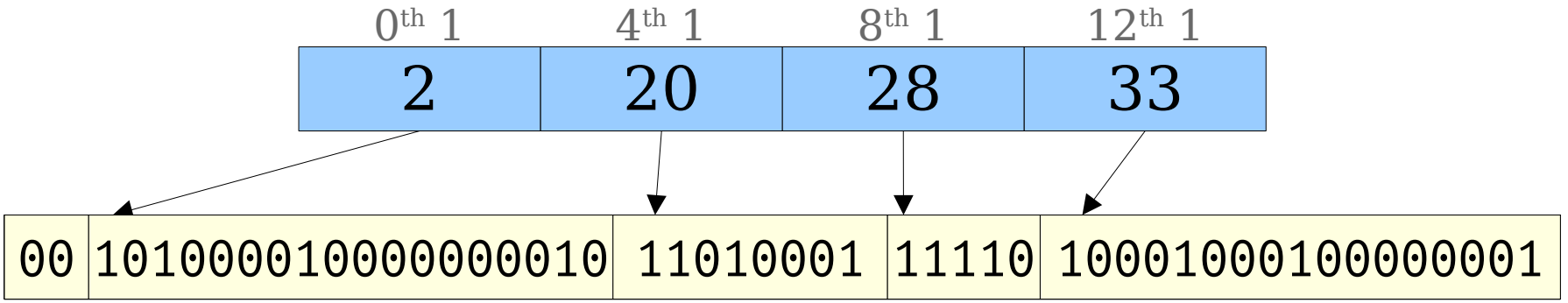
# Subdividing the Problem

- ***Key Challenge:*** If we split the input into blocks of size $b$, we can't easily tell which block to look in to find the $k$th 1 bit.
  - Different blocks may have different numbers of 1s in them.
- ***Question:*** Is there some other way to subdivide our problem into smaller pieces that avoids this issue?

| 11011100 | 10111011 | 11000100 | 11010101 | 11100111 |
|---|---|---|---|---|

# Subdividing the Problem

- With ***rank*** queries, our inputs are indices into the array (e.g. $0^{th}$ index, $1^{st}$ index, $2^{nd}$ index, etc.).
- With ***select*** queries, our inputs are ordinal numbers of 1 bits (e.g. $0^{th}$ 1 bit, $1^{st}$ 1 bit, $2^{nd}$ 1 bit, etc.).
- ***Idea:*** Pick an integer $c$, then write down the position of every $c$th 1 bit in the array.
- This subdivides our array into a series of ***chunks***, each of which has $c$ 1 bits in it.
  - Each chunk starts just before one of the precomputed 1 bit locations.
  - Edge cases: we don't consider a prefix of 0s at the front of the array to be a chunk, and the last chunk may have fewer than $c$ 1s.



# Subdividing the Problem

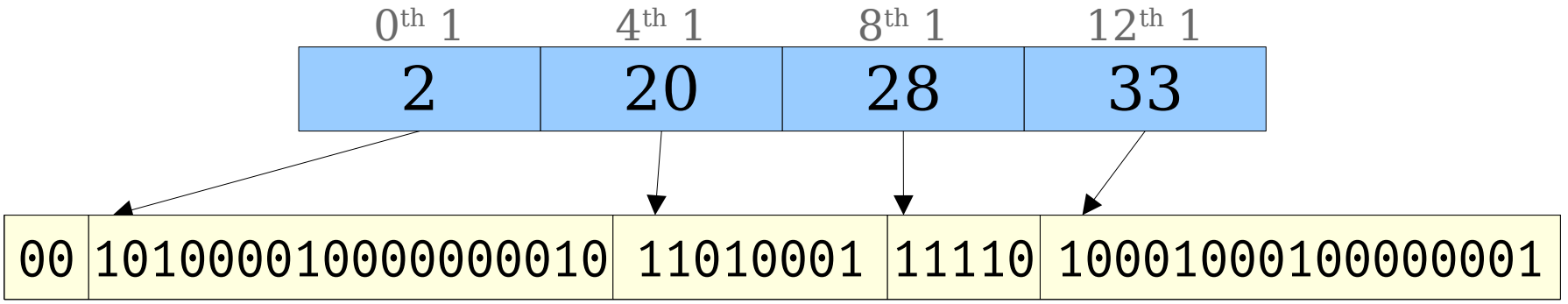
- Crucial details, which will drive most of the complexity of our solution:

***Every chunk has exactly c 1 bits.***

☞ ***Not all chunks have length c.*** ☞

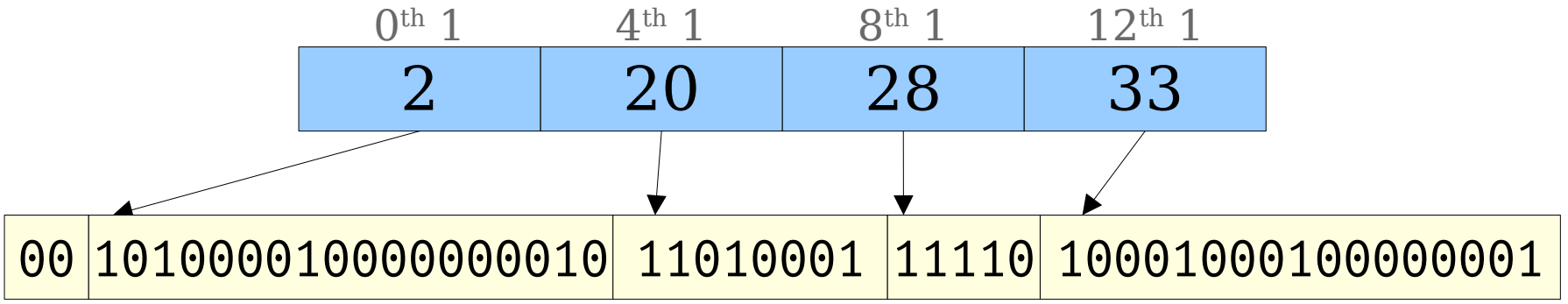
***Not all chunks have the same size.***

- This runs contrary to the intuitions we developed when working with fixed-sized blocks.
- Keep this in mind going forward – it's *super easy* to get this wrong and get lost.



# Subdividing the Problem

- How much space do we need for the summary array?
  - There are **O(*n* / *c*)** chunks. (*Why not Θ(n / c)?*)
  - Each index needs **O(log *n*)** bits.
  - Total space: O((***n* log *n***) / ***c***).
- For sufficiently large *c*, this would be sublinear space. This is promising!

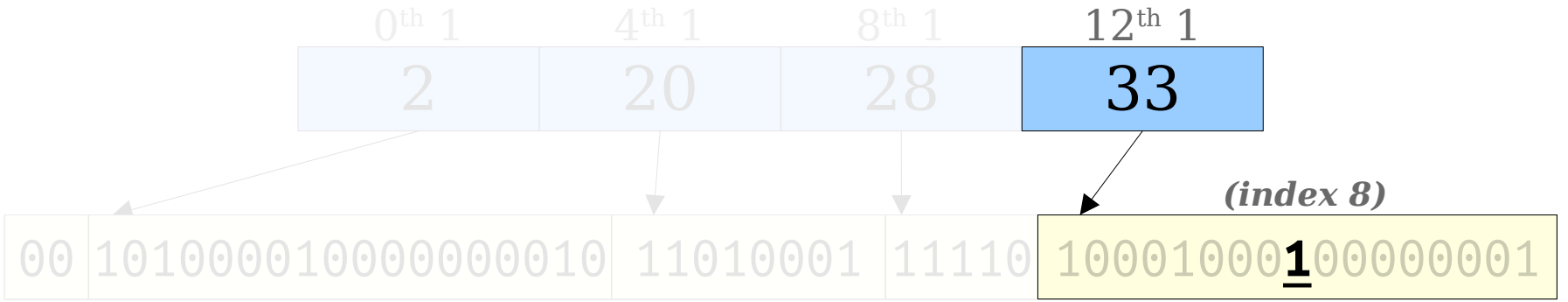


# Subdividing the Problem

- To answer a ***select***($k$) query, we do the following:
  - Compute $\lfloor k/c \rfloor$, the index of the chunk with the $k$th 1 bit.
  - Look up the start position of that chunk in the table.
  - Somehow **(?)** find the index of the ($k$ mod $c$)th 1 bit within that chunk.
  - Add that index to the start position of the chunk.
- The first two steps here take time O(1).
- How do we do the last step?

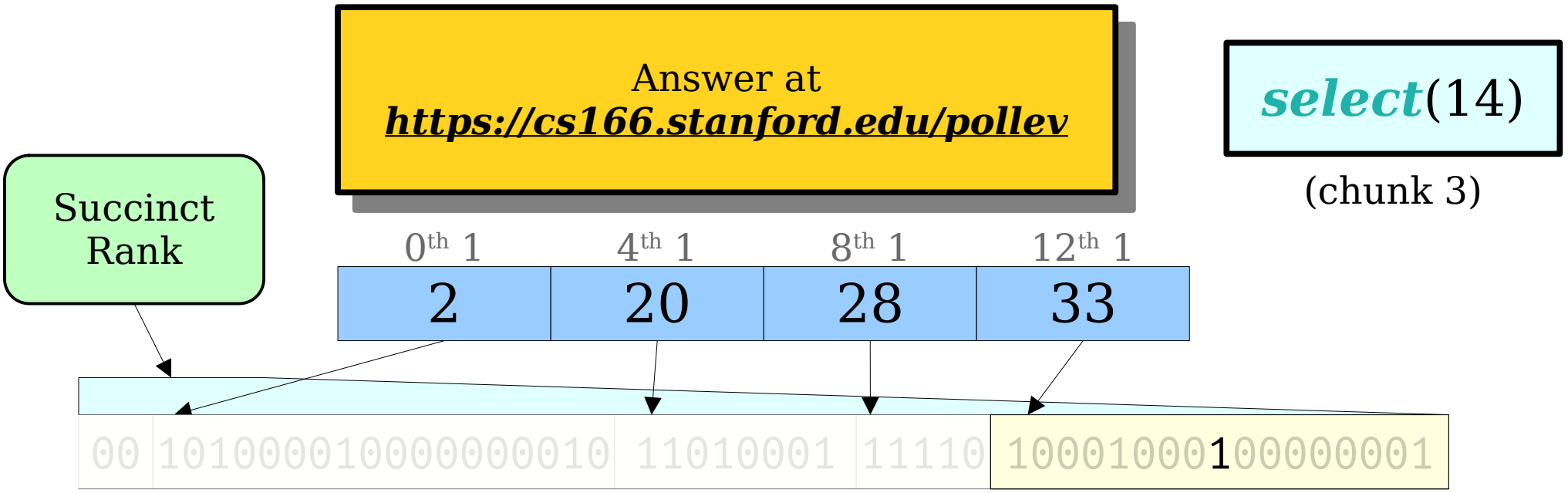
***select***(14) = 41

(chunk 3)



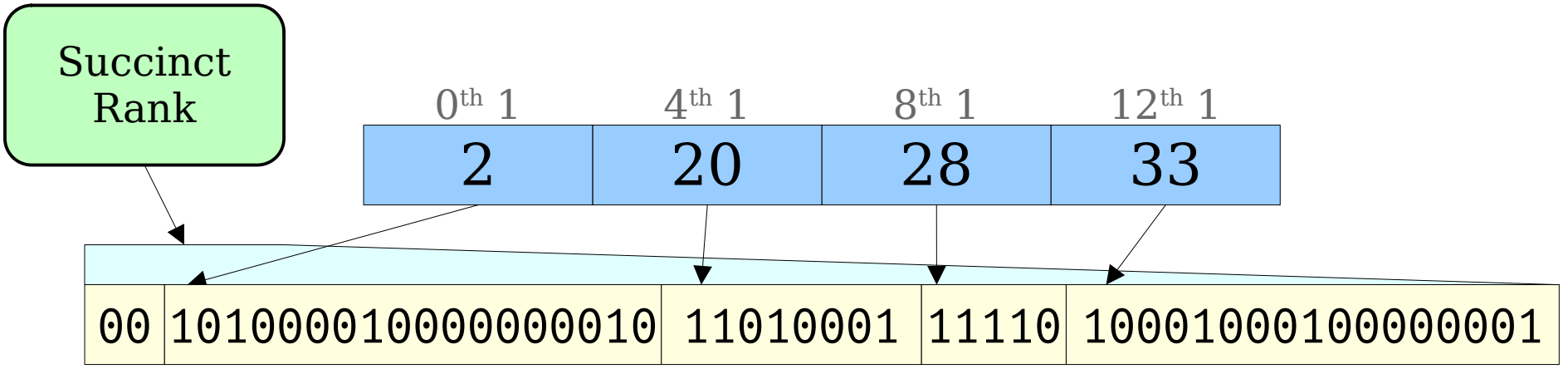
# Searching Within Chunks

- ***Approach 1:*** Use binary search within each chunk.
  - Construct a (global) succinct rank structure over the entire array of bits.
  - To find the *k*th 1 bit in a given chunk, use a binary search limited just to the range of bits within that chunk.
- ***Question:*** What's the runtime of this binary search?



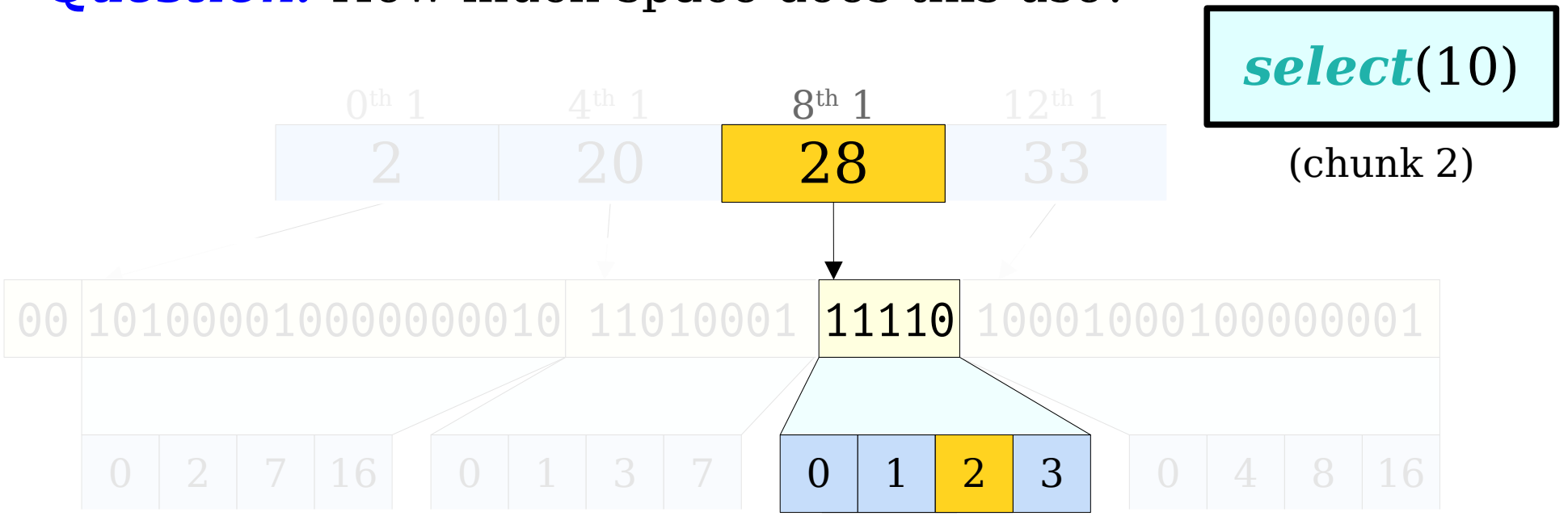
# Searching Within Chunks

- ***Answer:*** It depends on how big the chunks are.
  - A chunk of length $L$ requires $O(\log L)$ work.
  - If $\log L = o(\log n)$, this will be faster than binary searching over the whole array.
  - If $\log L = \Theta(\log n)$, this is not much better than binary searching over the whole array.
- We need a way to handle large chunks efficiently.



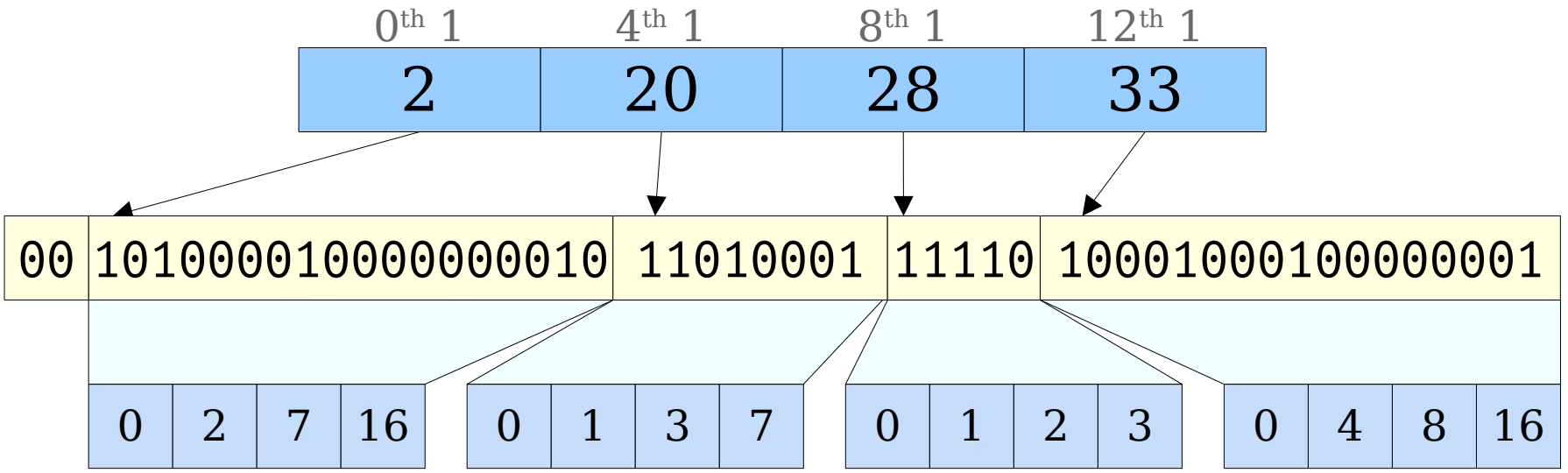
# Searching Within Chunks

- ***Idea 2:*** Precompute and store all the answers.
  - Write down the relative positions of each 1 bit within each chunk.
  - Answer queries by adding the relative offset within the chunk to the chunk start position.
- All queries can now be answered in time O(1).
- ***Question:*** How much space does this use?



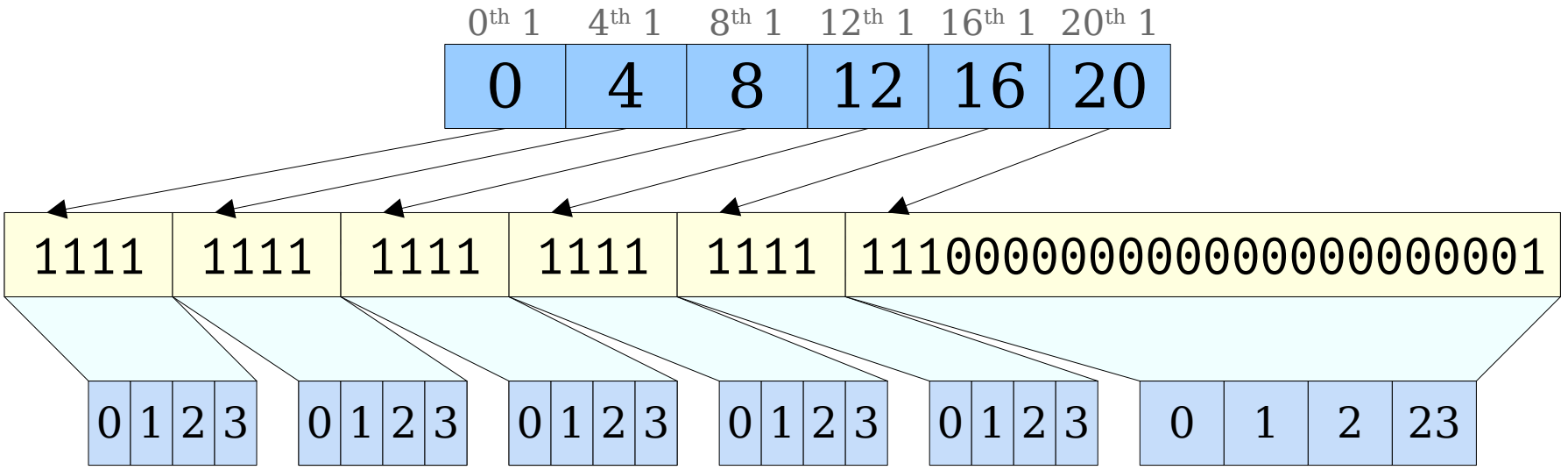
# Searching Within Chunks

- ***Answer:*** It depends on the sizes of the chunks.
  - If the chunk has size $L$, each index requires O(log $L$) bits. We thus use O($c$ log $L$) bits per chunk.
- Large chunks use more bits than small chunks.
- ***Plot Twist:*** While the above is true, it's actually not the large chunks we need to worry about!



# Searching Within Chunks

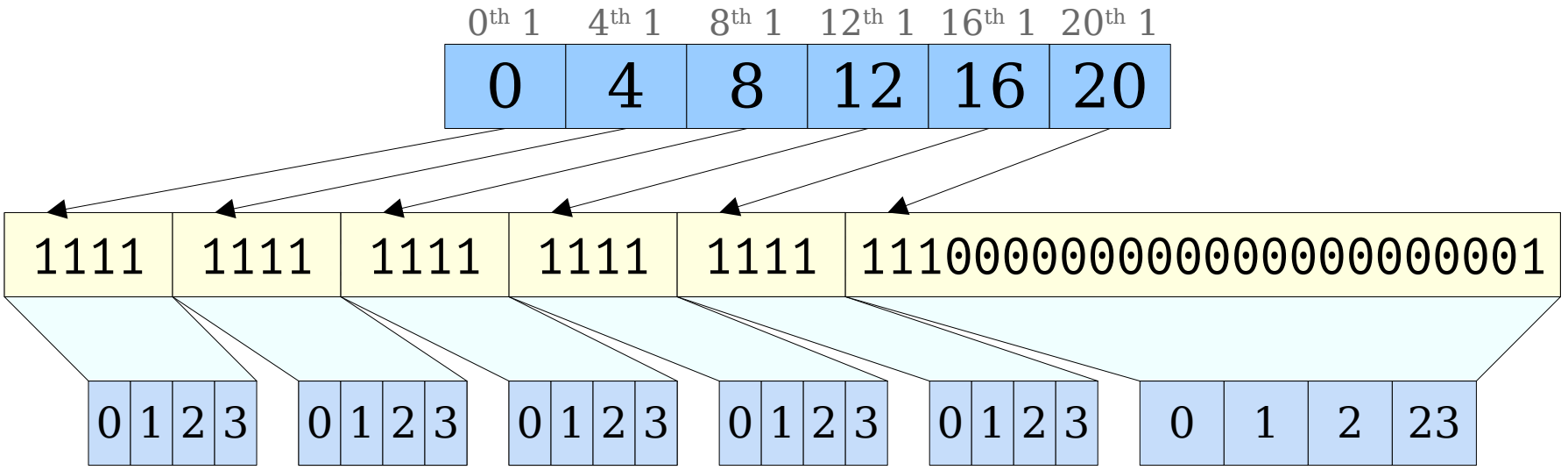
- Here’s an array of bits where all but one chunk is small and the rest are large.
- Although that last chunk needs more bits per number, most of the bits are accounted for by those earlier chunks.
- Let’s quantify how bad this is.



# Searching Within Chunks

- Suppose we have $\Theta(n)$ total 1 bits all at the front of the array.
- Then there will be **$\Theta(n / c)$** "small" chunks and 1 "large" chunk.
- Each small chunk needs **$O(c \log c)$** bits to write down indices.
- The one large chunk needs **$O(c \log n)$** bits for its indices.
- Total space:

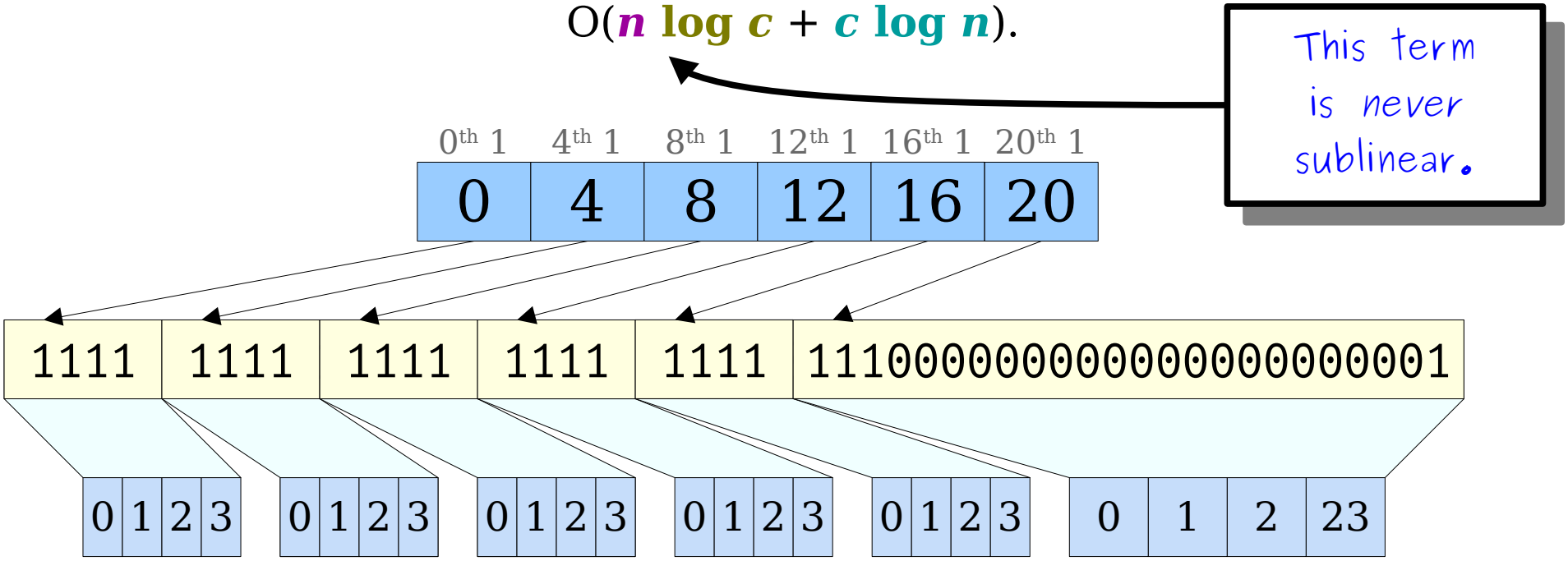
$$O((n / c)\, c \log c + c \log n).$$



# Searching Within Chunks

- Suppose we have $\Theta(n)$ total 1 bits all at the front of the array.
- Then there will be **$\Theta(n / c)$** "small" chunks and 1 "large" chunk.
- Each small chunk needs **$O(c \log c)$** bits to write down indices.
- The one large chunk needs **$O(c \log n)$** bits for its indices.
- Total space:

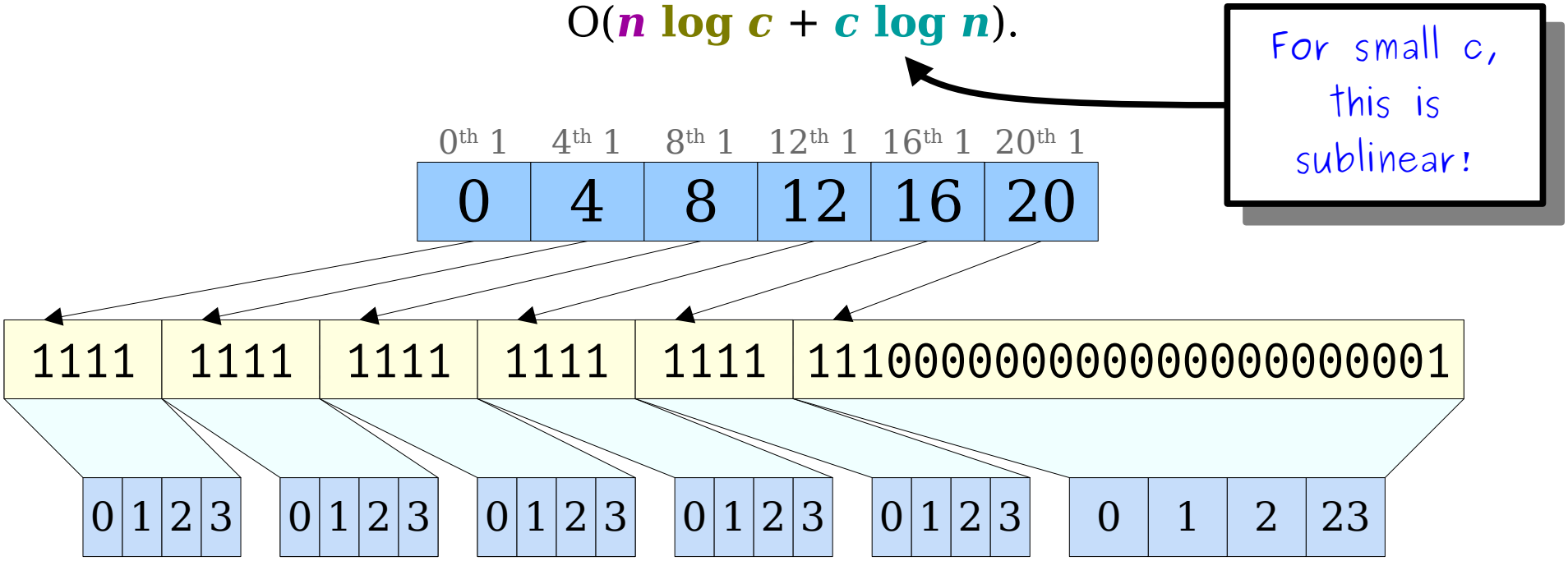
$O(n \log c + c \log n)$.



# Searching Within Chunks

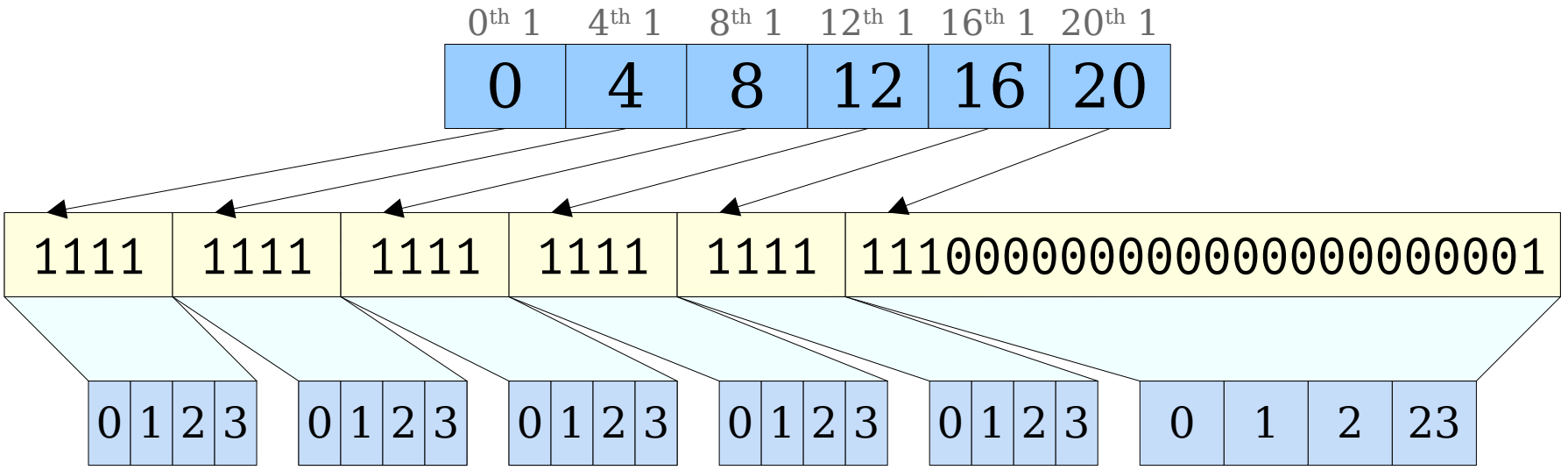
- Suppose we have $\Theta(n)$ total 1 bits all at the front of the array.
- Then there will be **$\Theta(n / c)$** "small" chunks and 1 "large" chunk.
- Each small chunk needs **$O(c \log c)$** bits to write down indices.
- The one large chunk needs **$O(c \log n)$** bits for its indices.
- Total space:

$O(n \log c + c \log n)$.



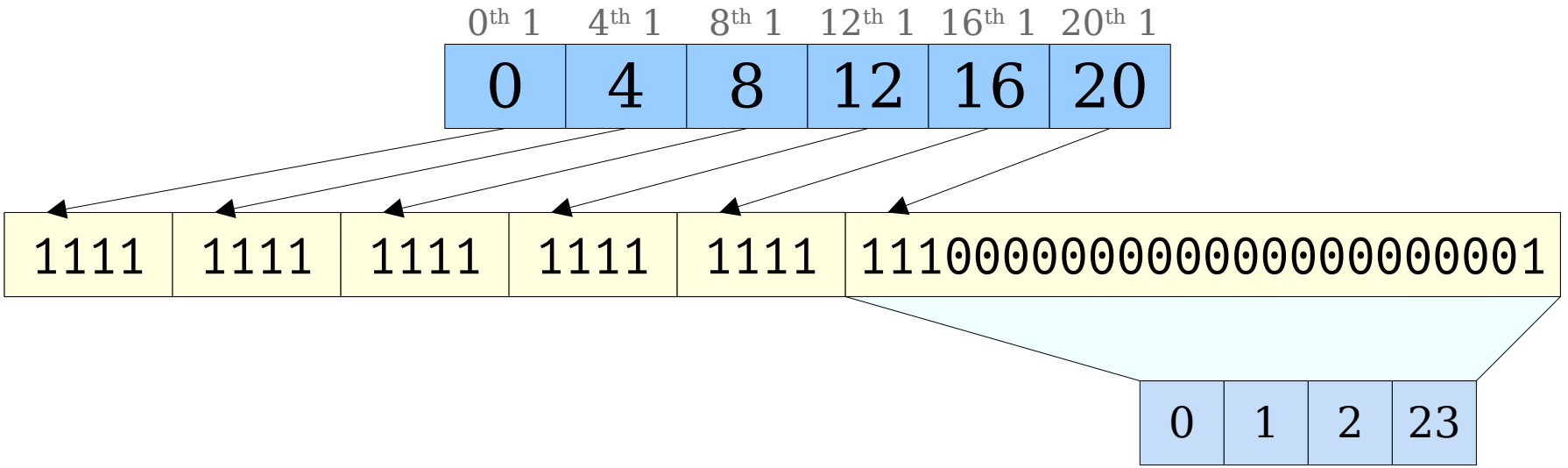
# Searching Within Chunks

- This single large chunk actually doesn't hurt us all that much.
  - "Large" means "sparse;" there's a lot more 0s than 1s.
  - We already saw that writing down the indices of all 1s works great in cases where the number of 1 bits is small.
- ***Insight:*** If we only write down 1 bit positions for chunks above a certain size threshold (that are "sufficiently sparse,") we won't use all that much space.



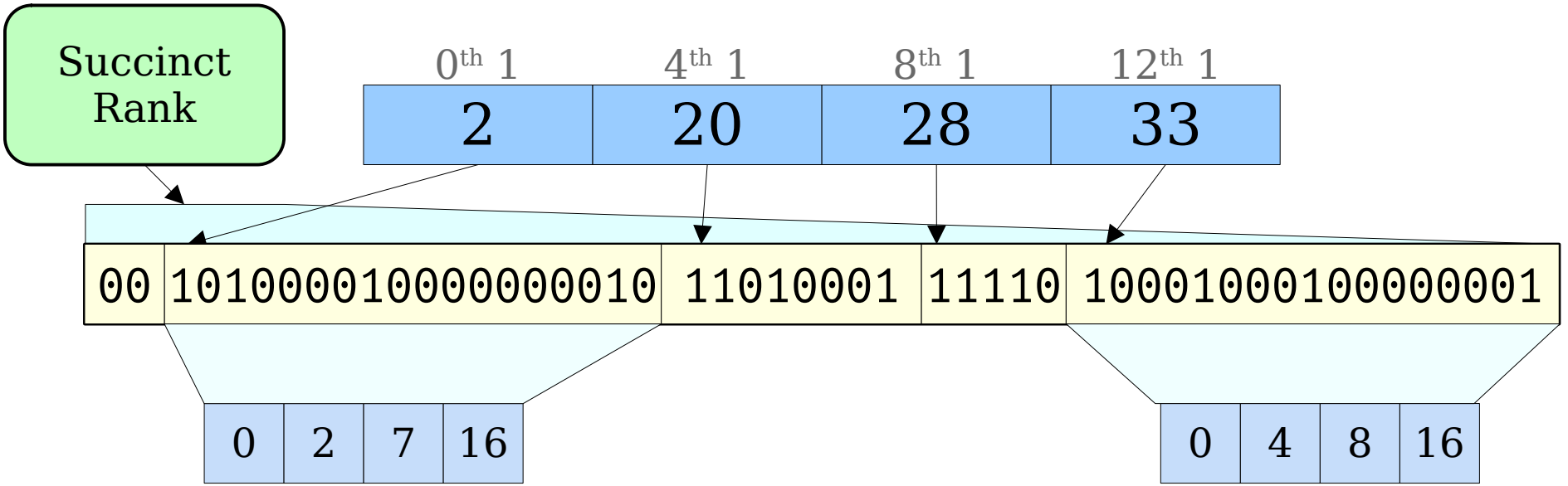
# The Best of Both Worlds

- Our binary search solution is
  - always space-efficient, but
  - only time-efficient on small chunks.
- Our precomputed table approach is
  - always time-efficient, but
  - only space-efficient on large chunks.
- ***Idea:*** Combine these approaches together!



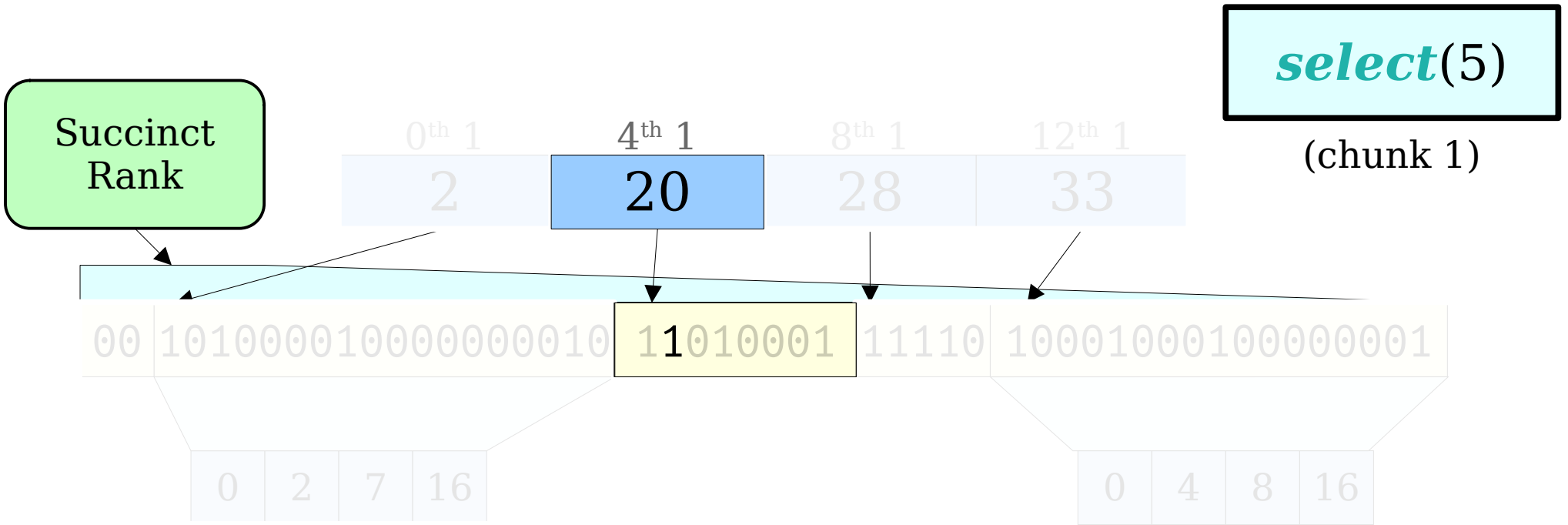
# The Best of Both Worlds

- Pick a chunk size $c$. Split the array into chunks containing exactly $c$ 1 bits each. Write down the positions of each $c$th 1 bit.
- Pick a ***threshold size*** $t$. For each chunk of length $t$ or more, write down the relative position of each 1 bit within the chunk.
- Form a succinct rank structure over the original bit array.



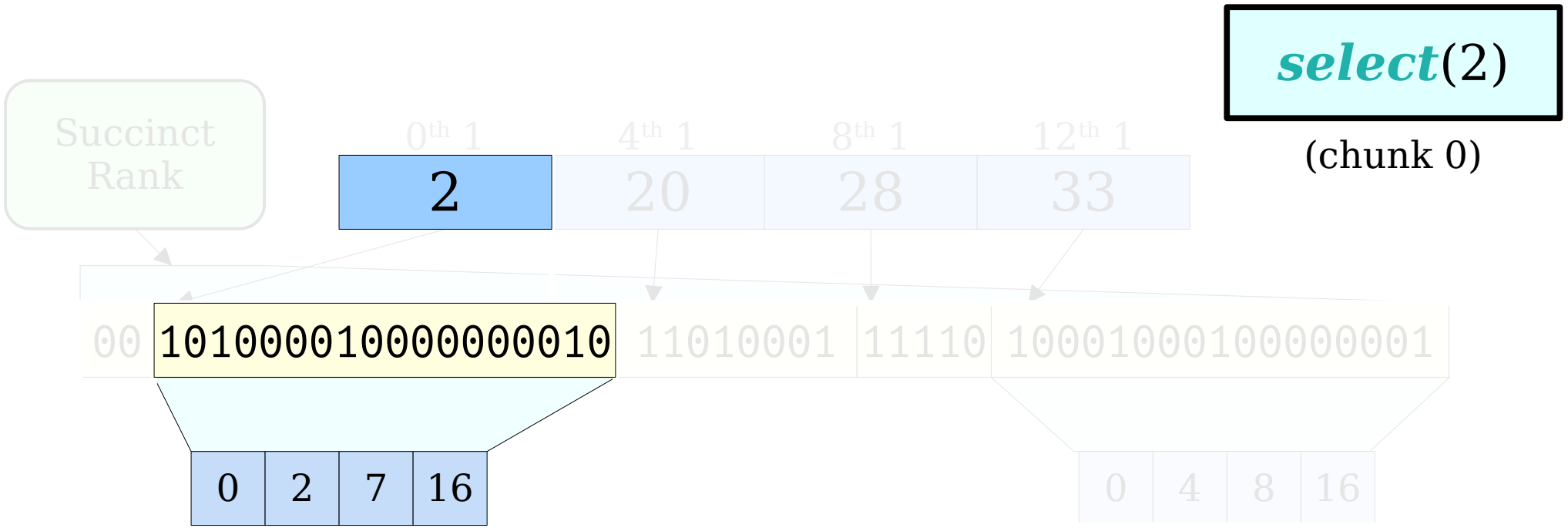
# The Best of Both Worlds

- To answer ***select***($k$):
  - Compute $\lfloor k/c \rfloor$, the index of the chunk containing bit $k$.
  - If that chunk is small (size < $t$), binary search over it to find bit $k$.



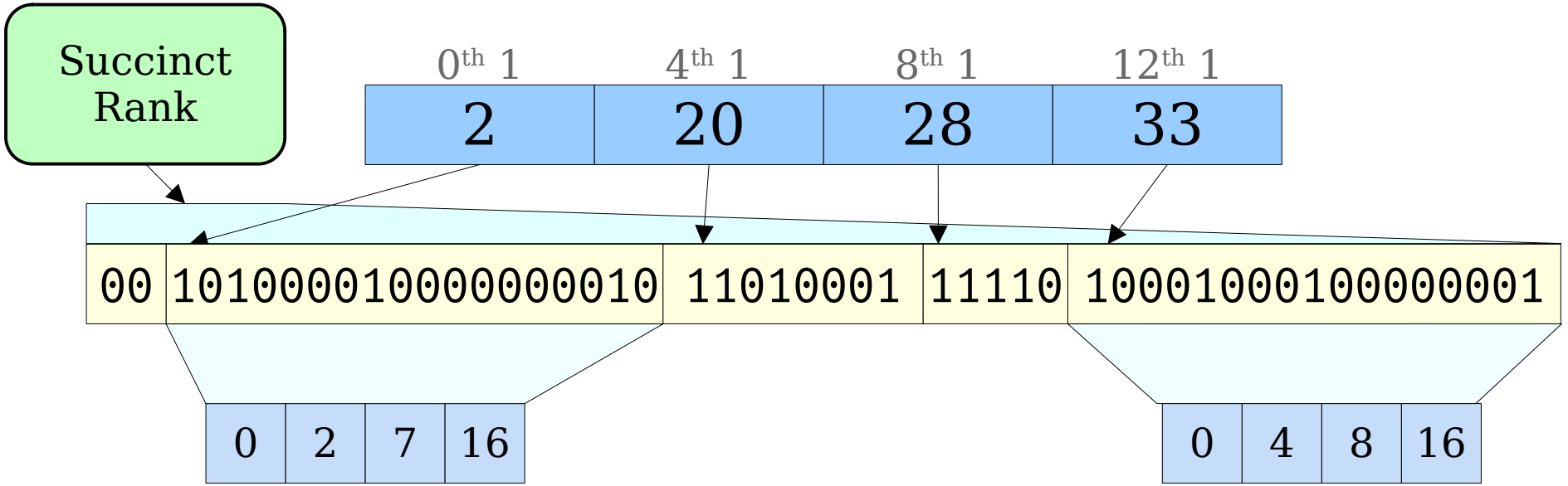
# The Best of Both Worlds

- To answer ***select***($k$):
  - Compute $\lfloor k/c \rfloor$, the index of the chunk containing bit $k$.
  - If that chunk is small (size < $t$), binary search over it to find bit $k$.
  - If that chunk is large (size ≥ $t$), look up the index of the ($k$ mod $c$)th 1 bit in that block. Add it to the index of the $\lfloor k/c \rfloor$th 1 bit from the top array.



# The Best of Both Worlds

- To answer ***select***($k$):
  - Compute $\lfloor k/c \rfloor$, the index of the chunk containing bit $k$.
  - If that chunk is small (size < $t$), binary search over it to find bit $k$.
  - If that chunk is large (size ≥ $t$), look up the index of the ($k$ mod $c$)th 1 bit in that block. Add it to the index of the $\lfloor k/c \rfloor$th 1 bit from the top array.
- Time required: **O(log *t*)**.

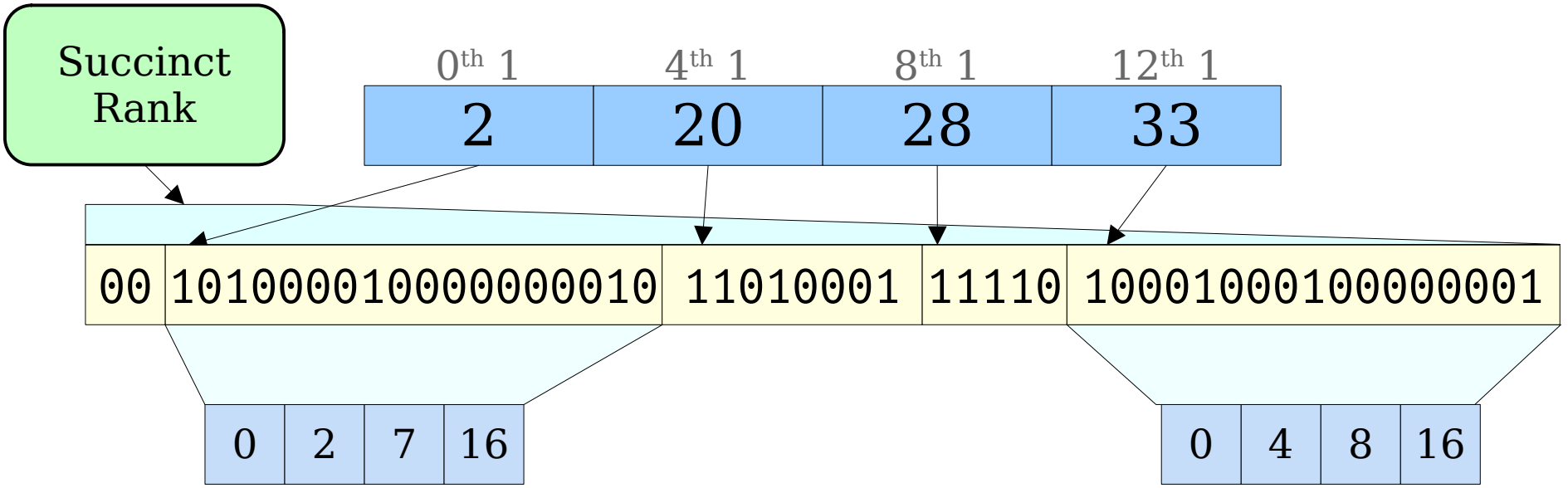


# The Best of Both Worlds

- The top-level array uses O((*n* log *n*) / *c*) bits. How much space do we need for the precomputed indices across all the large chunks?

Answer at

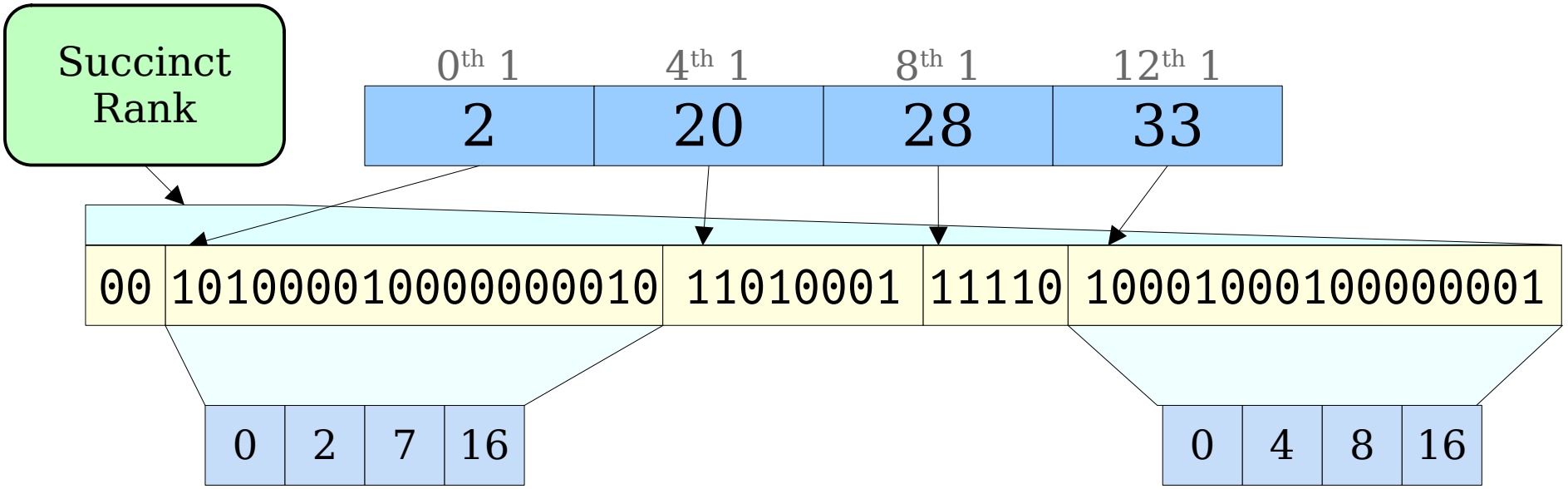
***https://cs166.stanford.edu/pollev***



# The Best of Both Worlds

- The top-level array uses O((*n* log *n*) / *c*) bits. How much space do we need for the precomputed indices across all the large chunks?
  - There can be at most O(***n* / *t***) "large" chunks (size ≥ *t*). *(Why?)*
  - Each large chunk writes ***c*** indices, each of which needs O(**log *n***) bits.
  - Space: O((***cn* log *n* / *t***).
- Space needed for the top-level array and the in-chunk arrays:

$$O\left(\frac{n \log n}{c} + \frac{cn \log n}{t}\right)$$



# Tuning the Data Structure

- Query Time: $O(\log t)$.
- Space Usage:

Binary search over small chunks.

$$O\left(n \log n \cdot \left[\frac{1}{c} + \frac{c}{t}\right]\right) + n + o(n)$$

Top-level table; large chunk tables.

Original bit array plus Jacobson rank.

# Tuning the Data Structure

- Query Time: O(log $t$).
- Space Usage:

$$O\left(n \log n \cdot \left[\frac{1}{c} + \frac{c}{t}\right]\right) + n + o(n)$$

$t$ needs to be small relative to $n$.

$c$ needs to be large relative to log $n$.

$c$ needs to be small relative to $t$.

- ***One good choice:*** Choose $c = \lg^2 n$, $t = \lg^4 n$.

# Tuning the Data Structure

- Query Time: O(log $t$).
- Space Usage:

$$\mathrm{O}\left(n \log n \cdot \left[\frac{1}{c} + \frac{c}{t}\right]\right) + n + o(n)$$

- ***One good choice:*** Choose $c = \lg^2 n$, $t = \lg^4 n$.

# Tuning the Data Structure

- Query Time: O(log **log² *n***).
- Space Usage:

$$O\left(n \log n \cdot \left[\frac{1}{c} + \frac{c}{t}\right]\right) + n + o(n)$$

- ***One good choice:*** Choose $c = \lg^2 n$, $t = \lg^4 n$.

# Tuning the Data Structure

- Query Time: **O(log log *n*)**.
- Space Usage:

$$O\left(n \log n \cdot \left[\frac{1}{c} + \frac{c}{t}\right]\right) + n + o(n)$$

- ***One good choice:*** Choose $c = \lg^2 n$, $t = \lg^4 n$.

# Tuning the Data Structure

- Query Time: **O(log log *n*)**.
- Space Usage:

$$O\left(n \log n \cdot \left[\frac{1}{c} + \frac{c}{t}\right]\right) + n + o(n)$$

- ***One good choice:*** Choose $c = \lg^2 n$, $t = \lg^4 n$.

# Tuning the Data Structure

- Query Time: **O(log log *n*)**.
- Space Usage:

$$O\left(n \log n \cdot \left[\frac{1}{\log^2 n}\right]\right) + n + o(n)$$

- ***One good choice:*** Choose $c = \lg^2 n$, $t = \lg^4 n$.

# Tuning the Data Structure

- Query Time: **O(log log *n*)**.
- Space Usage:

$$O\left(n \log n \cdot \left[ \frac{1}{\log^2 n} \right]\right) + n + o(n)$$

- ***One good choice:*** Choose $c = \lg^2 n$, $t = \lg^4 n$.

# Tuning the Data Structure

- Query Time: **O(log log *n*)**.
- Space Usage:

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

- ***One good choice:*** Choose $c = \lg^2 n$, $t = \lg^4 n$.

# Tuning the Data Structure

- Query Time: **O(log log *n*)**.
- Space Usage:

$$o(n) + n + o(n)$$

- ***One good choice:*** Choose $c = \lg^2 n$, $t = \lg^4 n$.

# Tuning the Data Structure

- Query Time: **O(log log *n*)**.
- Space Usage:

$$o(n) \quad + \quad n \quad + \quad o(n)$$

- ***One good choice:*** Choose $c = \lg^2 n$, $t = \lg^4 n$.

# Tuning the Data Structure

- Query Time: **O(log log *n*)**.
- Space Usage: ***n* + *o*(*n*)**.

- ***One good choice:*** Choose $c = \lg^2 n$, $t = \lg^4 n$.

# The Story So Far

- By splitting the array into chunks and combining our existing techniques, we can improve our query time to O(log log $n$) with sublinear auxiliary space.
- Can we do better?

| | Bits Needed | Query Time |
|---|---|---|
| Binary Search w/Jacobson Rank | $n + o(n)$ | O(log $n$) |
| Precompute-All | O($m$ log $n$) | O(1) |
| Hybrid Precompute + Binary Search | $n + o(n)$ | O(log log $n$) |

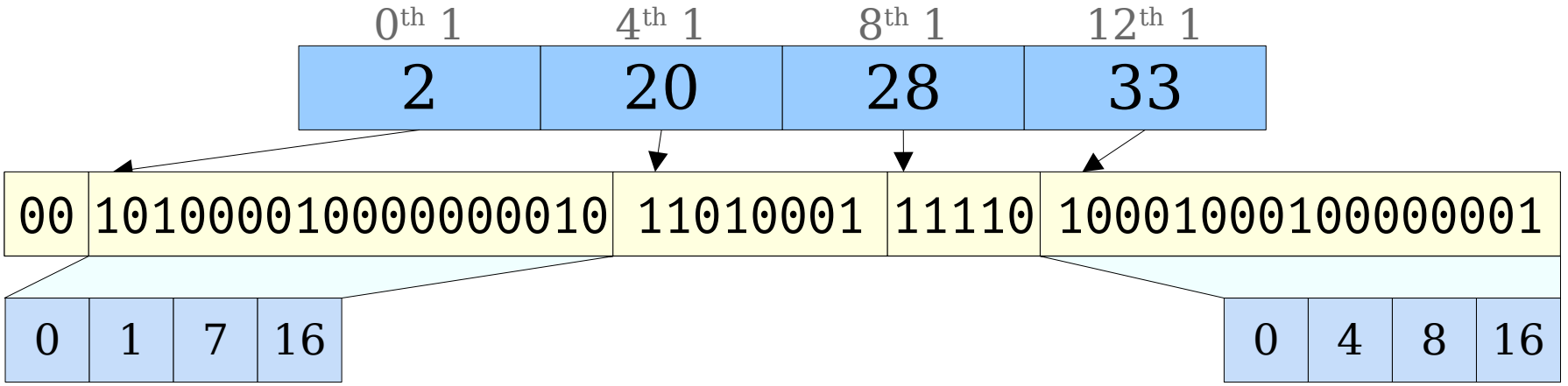
# The Story So Far

- ***Recall:*** Why is our query time O(log log $n$)?
  - Large chunks (size ≥ $\lg^4 n$) have all answers precomputed.
  - Small chunks (size < $\lg^4 n$) use binary search.
- The O(log log $n$) term results from binary searching over those arrays of size $\lg^4 n$.
- ***Idea:*** Repeat this process again to shrink those $\lg^4 n$-sized chunks down even further.

| | Bits Needed | Query Time |
|---|---|---|
| Hybrid Precompute + Binary Search | $n + o(n)$ | O(log log $n$) |

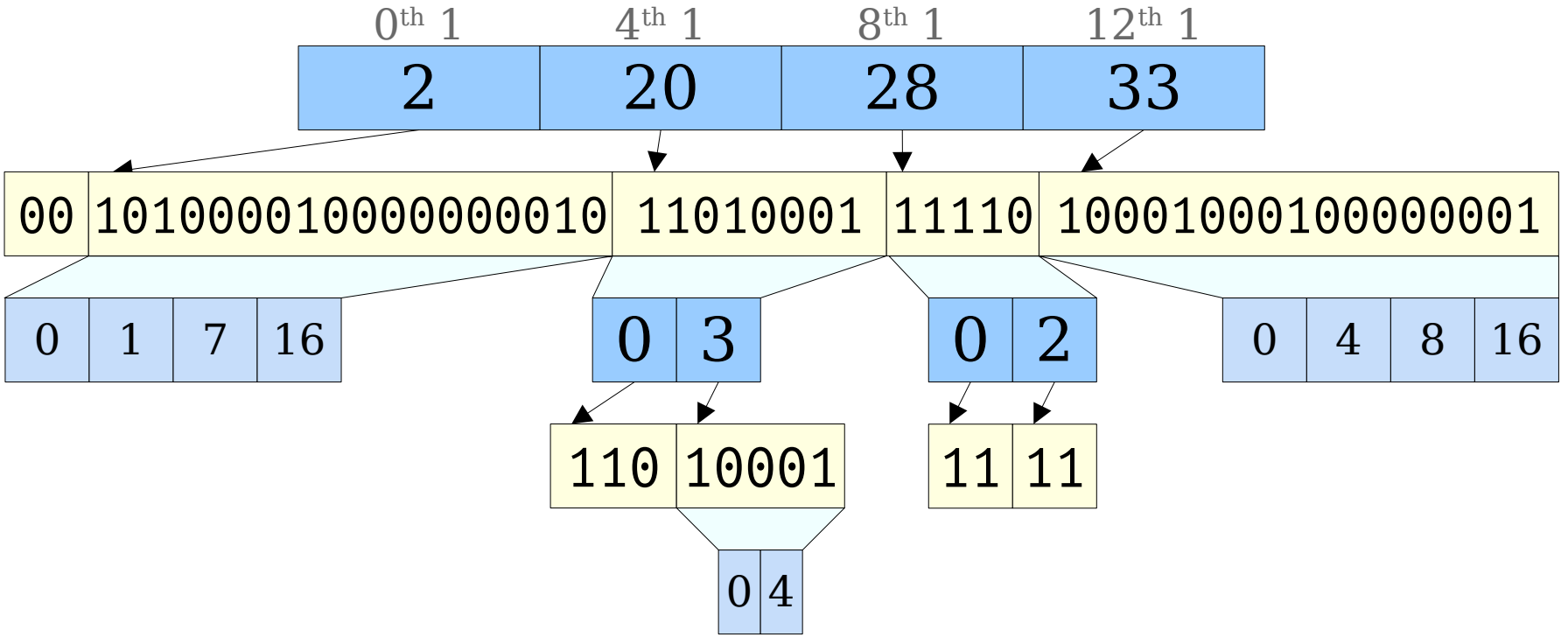
# We Have to Go Deeper

- Begin with the same approach as before.
  - Split into chunks of $c = \lg^2 n$ 1 bits. Write down every $c$th 1 bit's position.
  - Choose a threshold size $t = \lg^4 n$. Write down answers to queries within "large" (size $\geq t$) chunks.
  - Create a succinct rank structure over the original array. (Not shown)
- Answer queries in "large" chunks using precomputed answers.
- ***New:*** Don't use binary search in "small" chunks. Instead…



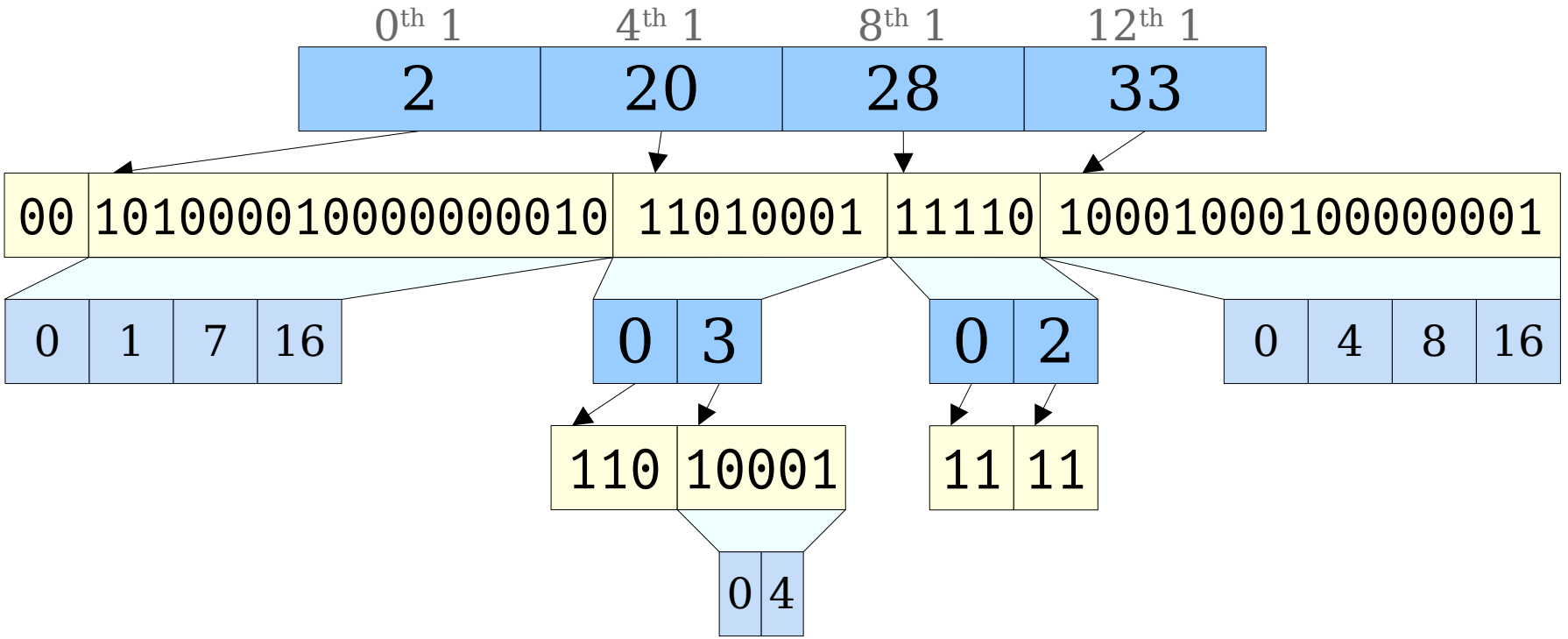
# We Have to Go Deeper

- Use the same approach as at a the top level, just inside the small chunk.
- Choose “minichunk” size ***c’***. Then, in each small chunk:
  - Write down the relative position of every *c*’th 1 bit.
  - Split the small chunk into “minichunks” just before every *c*’th 1 bit.
  - Pick a “mini-threshold” size ***t’***. Write down the relative offsets of each 1 bit within “large minichunks” (size ≥ *t*’).



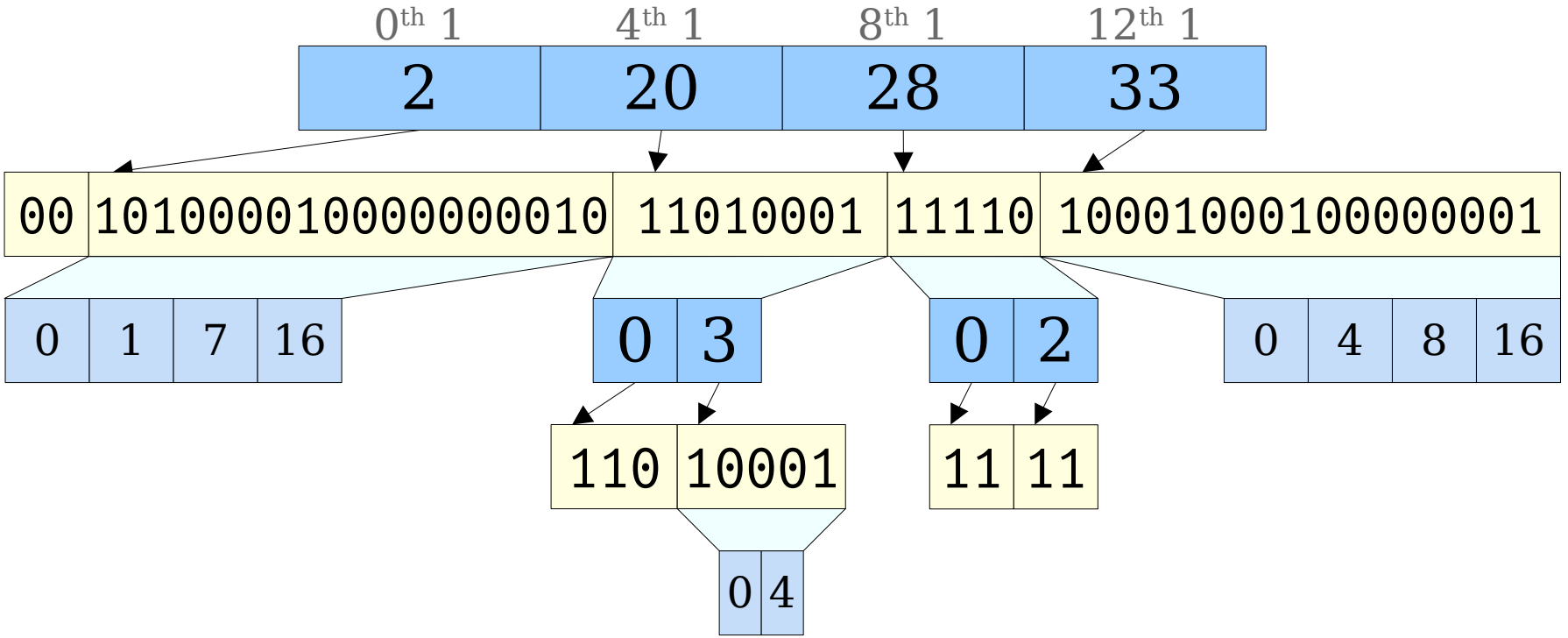
# We Have to Go Deeper

- To determine the position of the $k'$th bit within a small chunk, use the same approach as at the top level:
  - Compute $\lfloor k'/c' \rfloor$, the index of the minichunk containing this bit.
  - If that minichunk is "large" (size ≥ $t'$), look up the offset of the ($k'$ mod $c'$)th offset within the minichunk and add that to the minichunk start position.
  - If that minichunk is "small" (size < $t'$), binary search over the bits in the range using the succinct rank structure.



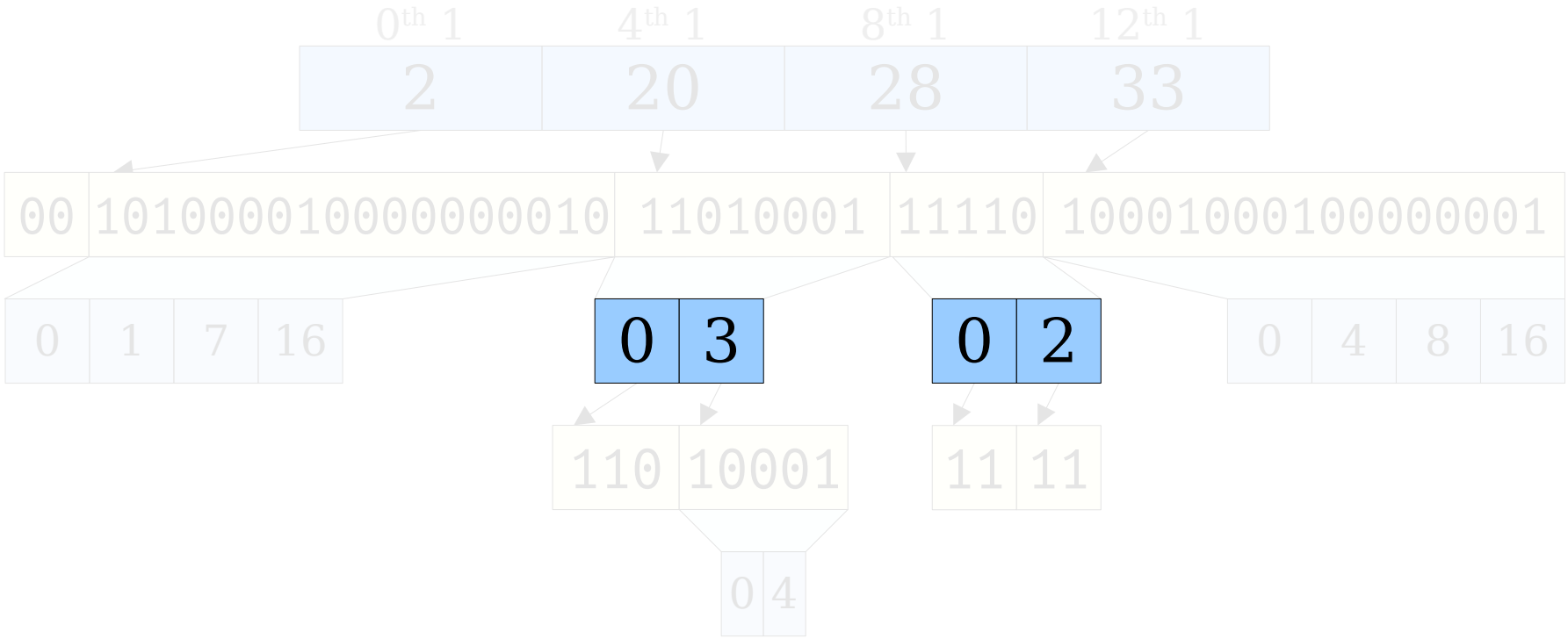
# We Have to Go Deeper

- How fast are queries?
  - Queries in large chunks are answered in time O(1).
  - Queries in “large minichunks” are answered in time O(1).
  - Queries in “small minichunks” are answered in time O(log *t’*)
- Query time: **O(log *t’*)**.



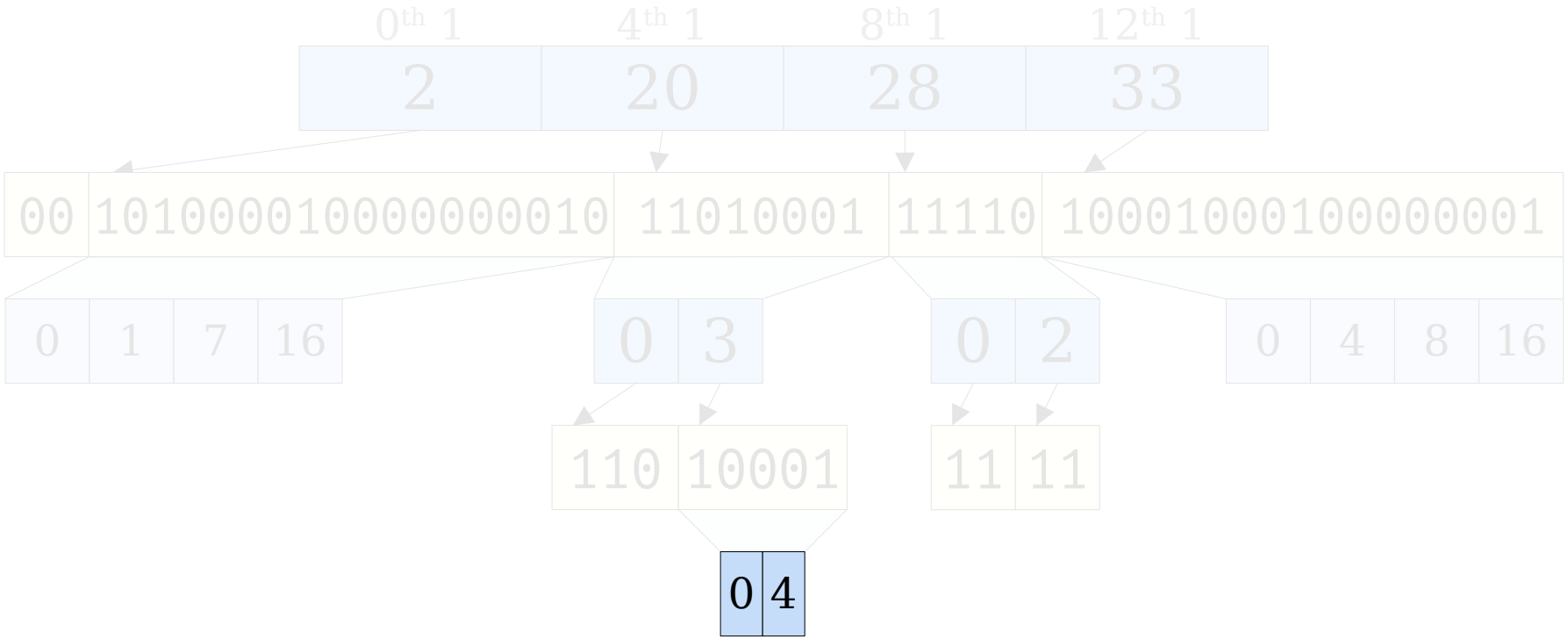
# We Have to Go Deeper

- How much additional space do we need for this approach?
- There are two sources of additional bits:
  - Writing out positions of every *c*'th 1 bit within small chunks.



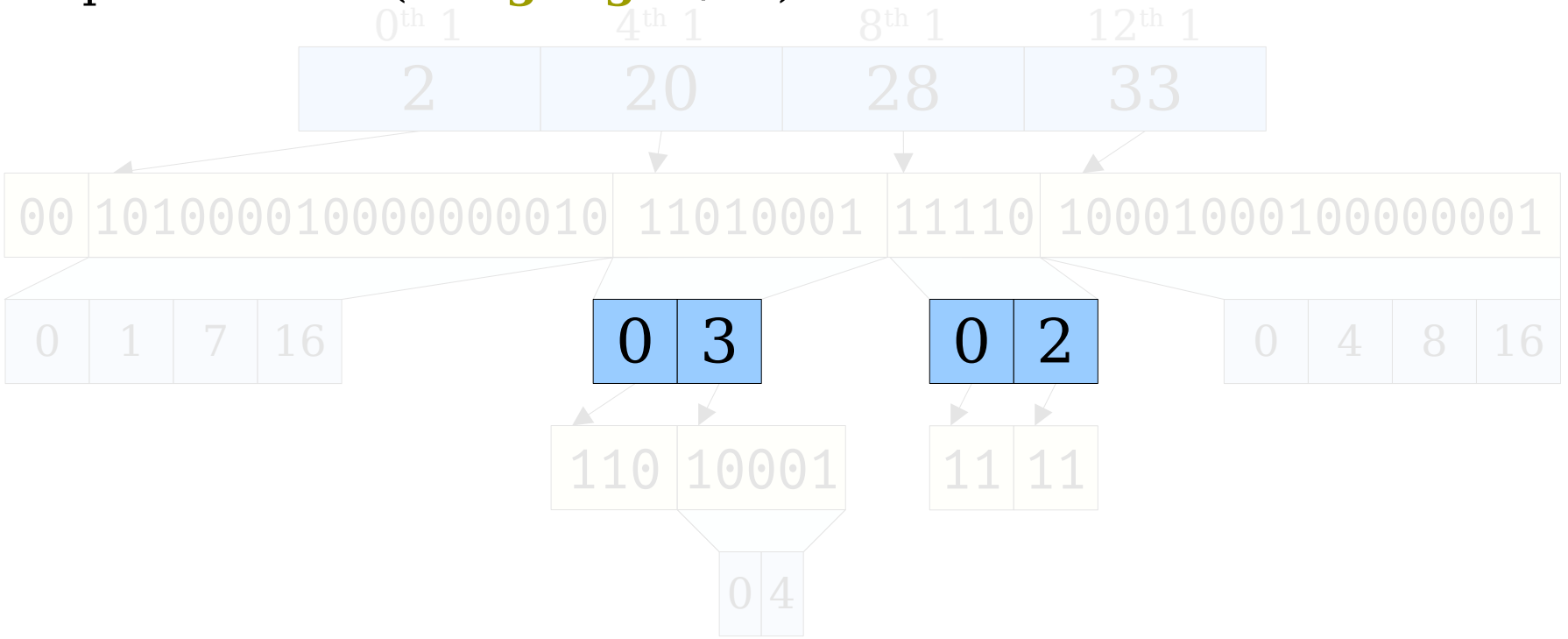
# We Have to Go Deeper

- How much additional space do we need for this approach?
- There are two sources of additional bits:
  - Writing out positions of every *c*'th 1 bit within small chunks.
  - Writing out answers within large minichunks.
- Let's treat each one in isolation.



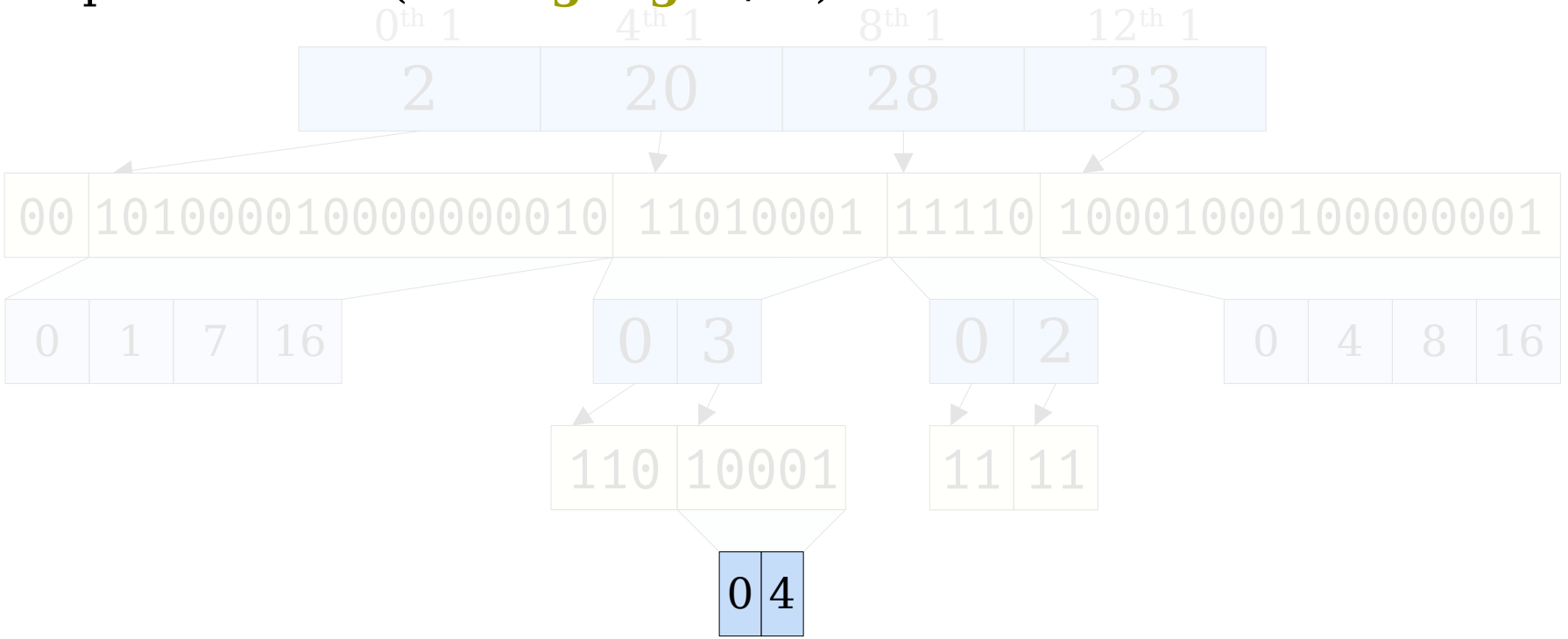
# We Have to Go Deeper

- Space for writing indices of every *c'*th 1 bit in small chunks:
  - Number of small chunks: O(***n* / log² *n***). *(Why?)*
  - Size of each small chunk: O(**log⁴ *n***). *(Why?)*
  - Bits per index: O(log **log⁴ *n***) = O(**log log *n***).
  - Number of indices per small chunk: O(**log² *n* / *c'***). *(Why?)*
- Space used: O(***n* log log *n* / *c'***).



# We Have to Go Deeper

- Space for indices of 1 bits in “large” minichunks (size ≥ $t'$):
  - Number of large minichunks: O($\boldsymbol{n / t'}$). *(Why?)*
  - Size of each large minichunk: O($\log^4 n$). *(Why?)*
  - Bits per index: O(log $\log^4 n$) = O($\log \log n$).
  - Number of indices per large minichunk: $\boldsymbol{c'}$.
- Space used: O($\boldsymbol{nc' \log \log n / t'}$).

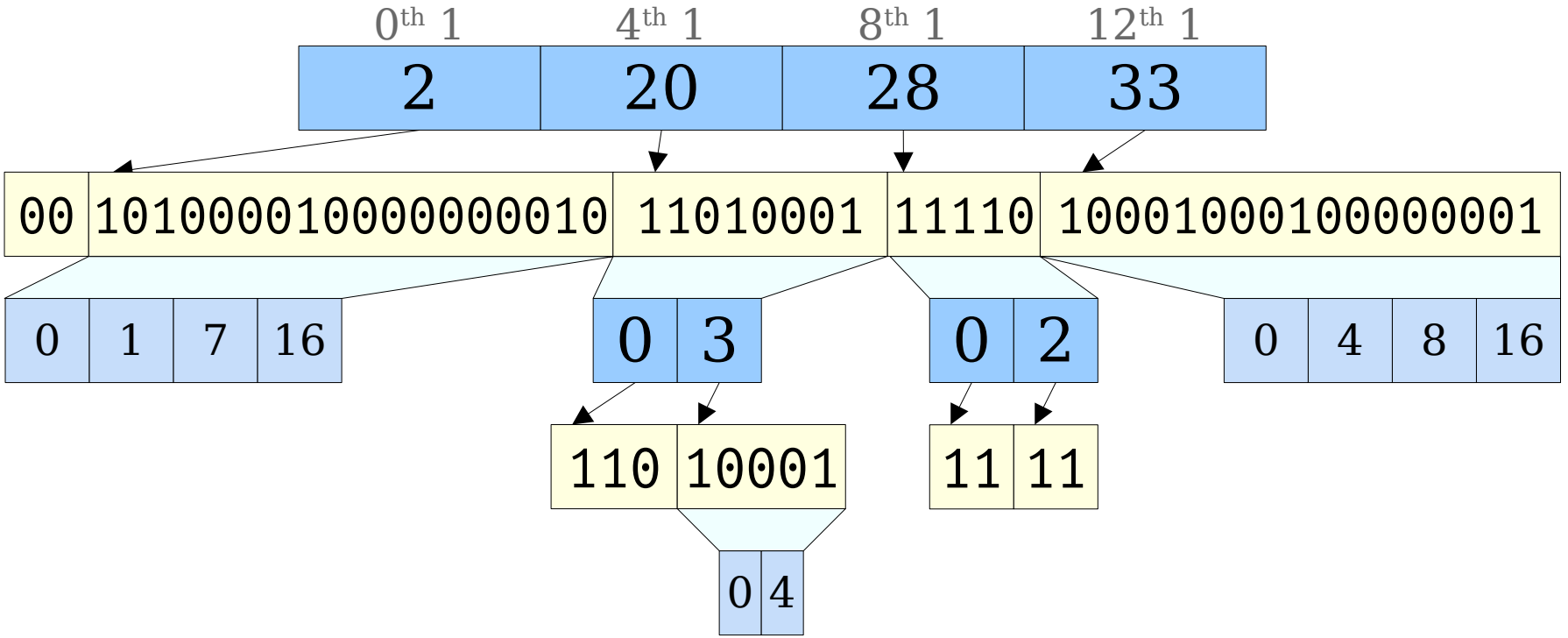


# We Have to Go Deeper

- Total space required for new tables·

$$O\left(\frac{n \log\log n}{c'} + \frac{c' n \log\log n}{t'}\right)$$

- This is what we had the first time around, except with a log log $n$ factor. (Makes sense – we're working with logarithmically-sized arrays.)



# Tuning the Data Structure

- Query Time: O(log $t'$).
- Space Usage:

$$O\left(n \log\log n \cdot \left[\frac{1}{c'} + \frac{c'}{t'}\right]\right) + n + o(n)$$

Binary search over small minichunks.

Chunk offsets within small blocks; answers within large miniblocks.

Original bit array, Jacobson rank, tables from before.

# Tuning the Data Structure

- Query Time: O(log $t'$).
- Space Usage:

$$O\left(n \log\log n \cdot \left\lceil \frac{1}{c'} + \frac{c'}{t'} \right\rceil\right) + n + o(n)$$

$t'$ needs to be small relative to log $n$.

$c'$ needs to be large relative to log log $n$.

$c'$ needs to be small relative to $t'$.

- ***One good choice:*** $c' = \lg^2 \lg^4 n$, $t' = \lg^4 \lg^4 n$.
  - These are the same choices we made at the top level, except that they're applied to $\lg^4 n$ rather than $n$ itself.

# Tuning the Data Structure

- Query Time: O(log $t'$).
- Space Usage:

$$O\left(n \log \log n \cdot \left[\frac{1}{c'} + \frac{c'}{t'}\right]\right) + n + o(n)$$

- ***One good choice:*** $c' = \lg^2 \lg^4 n$, $t' = \lg^4 \lg^4 n$.
  - These are the same choices we made at the top level, except that they're applied to $\lg^4 n$ rather than $n$ itself.

# Tuning the Data Structure

- Query Time: O(log **log² log⁴ *n***).
- Space Usage:

$$O\left(n \log\log n \cdot \left[\frac{1}{c'} + \frac{c'}{t'}\right]\right) + n + o(n)$$

- ***One good choice:*** $c' = \lg^2 \lg^4 n$, $t' = \lg^4 \lg^4 n$.
  - These are the same choices we made at the top level, except that they're applied to $\lg^4 n$ rather than $n$ itself.

# Tuning the Data Structure

- Query Time: **O(log log log *n*)**.
- Space Usage:

$$O\left(n \log\log n \cdot \left[\frac{1}{c'} + \frac{c'}{t'}\right]\right) + n + o(n)$$

- ***One good choice:*** $c' = \lg^2 \lg^4 n$, $t' = \lg^4 \lg^4 n$.
  - These are the same choices we made at the top level, except that they're applied to $\lg^4 n$ rather than $n$ itself.

# Tuning the Data Structure

- Query Time: **O(log log log *n*)**.
- Space Usage:

$$O\left(n \log\log n \cdot \left[\frac{1}{c'} + \frac{c'}{t'}\right]\right) + n + o(n)$$

- ***One good choice:*** $c' = \lg^2 \lg^4 n$, $t' = \lg^4 \lg^4 n$.
  - These are the same choices we made at the top level, except that they're applied to $\lg^4 n$ rather than $n$ itself.

# Tuning the Data Structure

- Query Time: **O(log log log *n*)**.
- Space Usage:

$$O\left(n \log\log n \cdot \left\lceil \frac{1}{\log^2 \log n} \right\rceil\right) + n + o(n)$$

- ***One good choice:*** $c' = \lg^2 \lg^4 n$, $t' = \lg^4 \lg^4 n$.
  - These are the same choices we made at the top level, except that they're applied to $\lg^4 n$ rather than $n$ itself.

# Tuning the Data Structure

- Query Time: **O(log log log *n*)**.
- Space Usage:

$$O\left(n \log\log n \cdot \left[\frac{1}{\log^2 \log n}\right]\right) + n + o(n)$$

- ***One good choice:*** $c' = \lg^2 \lg^4 n$, $t' = \lg^4 \lg^4 n$.
  - These are the same choices we made at the top level, except that they're applied to $\lg^4 n$ rather than $n$ itself.

# Tuning the Data Structure

- Query Time: **O(log log log *n*)**.
- Space Usage:

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

- ***One good choice:*** $c' = \lg^2 \lg^4 n$, $t' = \lg^4 \lg^4 n$.
  - These are the same choices we made at the top level, except that they're applied to $\lg^4 n$ rather than $n$ itself.

# Tuning the Data Structure

- Query Time: **O(log log log *n*)**.
- Space Usage:

$$o(n) + n + o(n)$$

- ***One good choice:*** $c' = \lg^2 \lg^4 n$, $t' = \lg^4 \lg^4 n$.
  - These are the same choices we made at the top level, except that they're applied to $\lg^4 n$ rather than $n$ itself.

# Tuning the Data Structure

- Query Time: **O(log log log *n*)**.
- Space Usage:

$$o(n) \quad + \quad n \quad + \quad o(n)$$

- ***One good choice:*** $c' = \lg^2 \lg^4 n$, $t' = \lg^4 \lg^4 n$.
  - These are the same choices we made at the top level, except that they're applied to $\lg^4 n$ rather than $n$ itself.

# Tuning the Data Structure

- Query Time: **O(log log log *n*)**.
- Space Usage: ***n* + *o*(*n*)**.

- ***One good choice:*** $c' = \lg^2 \lg^4 n$, $t' = \lg^4 \lg^4 n$.
  - These are the same choices we made at the top level, except that they're applied to $\lg^4 n$ rather than $n$ itself.

# The Story So Far

- ***Question:*** Can we get the query time down to O(1)?

| | Bits Needed | Query Time |
|---|---|---|
| Binary Search w/Jacobson Rank | $n + o(n)$ | O(log $n$) |
| Precompute-All | O($m$ log $n$) | O(1) |
| Hybrid Precompute + Binary Search | $n + o(n)$ | O(log log $n$) |
| Two-Level Hybrid + Binary Search | $n + o(n)$ | O(log log log $n$) |

# The Story So Far

- ***Question:*** Why is our query time O(log log log $n$)?
  - We might have to do a binary search in a "small minichunk" of size $\lg^4 \lg^4 n$.
- We could iterate this process again to add another log factor, but that won't get us down to constant time.
- ***Observation:*** $\lg^4 \lg^4 n$ bits is a pretty small number of bits. There can't be that many different small minichunks.
- Where have we seen this before? 🤔

| | Bits Needed | Query Time |
|---|---|---|
| Two-Level Hybrid + Binary Search | $n + o(n)$ | O(log log log $n$) |

# Return of the Four Russians

- Each small minichunk has length at most $t'$.
  - Possible small minichunks: O($\mathbf{2^{t'}}$).
- Each small minichunk has $c'$ 1 bits in it.
  - Possible queries: O($c'$) = O($\mathbf{t'}$).
- Each query answer is an index into an array of $t'$ bits.
  - Bits per query answer: O($\mathbf{\log t'}$).
- Size of the Four-Russians table: O($\mathbf{2^{t'} \cdot t' \cdot \log t'}$).
- As with rank queries, if $t' \leq \frac{1}{2} \lg n$, this is $\mathbf{o(n)}$ bits.

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

# Return of the Four Russians

- Our final* structure is a slight modification from before.
  - Split the array into chunks of $c = \lg^2 n$ 1 bits. Choose threshold $t = \lg^4 n$ and write down answers to all queries in chunks of size $\geq t$.
  - Split small chunks of size $< t$ into minichunks of $c' = \lg^2 \lg^4 n$ 1 bits. Choose minithreshold $t' = \lg^4 \lg^4 n$ and write down answers to all queries in minichunks of size $\geq t'$.
  - ***New:*** Instead of building a succinct rank structure, build a Four Russians table with answers to all possible queries that could be made in small minichunks of size $< t'$.
- Queries are now fully table-driven; time is now **O(1)**. 🎉

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

** almost*

# A Theoryland Hack

- Small minichunks have size $t' = \lg^4 \lg^4 n$ or less, and…
  - ***technically,*** we have $\lg^4 \lg^4 n = o(\log n)$, so…
  - ***technically,*** there is a constant $n_0$ where, for all $n > n_0$, we have $t' \leq \frac{1}{2} \lg n$, so…
  - ***technically,*** the Four Russians table uses $o(n)$ bits because
    - if $n \geq n_0$, then $t' \leq \frac{1}{2} \lg n$ and the table uses $o(n)$ bits, and if
    - if $n < n_0$, the table uses at most a constant number of bits.
- So, um, what is $n_0$, and what is that "constant number of bits" for the table?
- ***Answer:*** $n_0 \approx 2^{327,880,383}$, and we need about $2^{163,940,224}$ bits.

# The Final Scorecard?

| | Bits Needed | Query Time |
|---|---|---|
| Binary Search w/Jacobson Rank | $n + o(n)$ | O(log $n$) |
| Precompute-All | O($m$ log $n$) | O(1) |
| Hybrid Precompute + Binary Search | $n + o(n)$ | O(log log $n$) |
| Two-Level Hybrid + Binary Search | $n + o(n)$ | O(log log log $n$) |
| Two-Level Hybrid + Four Russians. * | $n + o(n)$ | O(1) |

** technically*

# Making this Practical

Why did we pick $t = \lg^4 \lg^4 n$?

# Tuning the Data Structure

- Query Time: $O(\log t')$.
- Space Usage:

$$O\left(n \log\log n \cdot \left\lceil \frac{1}{c'} + \frac{c'}{t'} \right\rceil\right) + n + o(n)$$

$t'$ needs to be small relative to $\log n$.

$c'$ needs to be large relative to $\log\log n$.

$c'$ needs to be small relative to $t'$.

- ***One good choice:*** $c' = \lg^2 \lg^4 n$, $t' = \lg^4 \lg^4 n$.
  - These are the same choices we made at the top level, except that they're applied to $\lg^4 n$ rather than $n$ itself.

# Tuning the Data Structure

- Query Time: O(log $t'$).

$t'$ needs to be small relative to log $n$.

- Space Usage:

$$O\left(n \log\log n \cdot \left[\frac{1}{c'} + \frac{c'}{t'}\right]\right) + n + o(n)$$

$c'$ needs to be large relative to log log $n$.

$c'$ needs to be small relative to $t'$.

- ***One good choice:*** $c' = \lg^2 \lg^4 n$, $t' = \lg^4 \lg^4 n$.
  - These are the same choices we made at the top level, except that they're applied to $\lg^4 n$ rather than $n$ itself.

# Our New Constraints

- Query Time: O(1).
- Space Usage:

$$O\left(n \log\log n \cdot \left[\frac{1}{c'} + \frac{c'}{t'}\right]\right) + O\left(2^{t'} \cdot t' \cdot \log t'\right) + n + o(n)$$

Four Russians table

Second-level tables

Original bit array, first-level tables.

# Our New Constraints

- Query Time: O(1).
- Space Usage:

$$O\left(n \log\log n \cdot \left[\frac{1}{c'} + \frac{c'}{t'}\right]\right) + O\left(2^{t'} \cdot t' \cdot \log t'\right) + n + o(n)$$

*t'* needs to be at most ½ lg *n*.

*c'* needs to be large relative to log log *n*.

*c'* needs to be small relative to *t'*.

- ***Better Choices:***

$$c' = \sqrt{\lg n}$$

$$t' = \frac{1}{2} \lg n$$

# Our New Constraints

- Query Time: O(1).
- Space Usage:

$$\mathrm{O}\left(n \log\log n \cdot \left[\frac{1}{c'} + \frac{c'}{t'}\right]\right) + \mathrm{O}\left(2^{t'} \cdot t' \cdot \log t'\right) + n + o(n)$$

- ***Better Choices:***

$$c' = \sqrt{\lg n}$$

$$t' = \frac{1}{2} \lg n$$

# Our New Constraints

- Query Time: O(1).
- Space Usage:

$$O\left(n \log\log n \cdot \left[\frac{1}{c'} + \frac{c'}{t'}\right]\right) + O\left(2^{t'} \cdot t' \cdot \log t'\right) + n + o(n)$$

- ***Better Choices:***

$$c' = \sqrt{\lg n}$$

$$t' = \frac{1}{2} \lg n$$

# Our New Constraints

- Query Time: O(1).
- Space Usage:

$$O\left(n \log\log n \cdot \left[\frac{1}{c'} + \frac{c'}{t'}\right]\right) + O\left(2^{½ \lg n} \cdot \lg n \cdot \lg\lg n\right) + n + o(n)$$

- ***Better Choices:***

$$c' = \sqrt{\lg n}$$

$$t' = \frac{1}{2} \lg n$$

# Our New Constraints

- Query Time: O(1).
- Space Usage:

$$O\left(n \log\log n \cdot \left[\frac{1}{c'} + \frac{c'}{t'}\right]\right) + O\left(\sqrt{n} \cdot \lg n \cdot \lg\lg n\right) + n + o(n)$$

- ***Better Choices:***

$$c' = \sqrt{\lg n}$$

$$t' = \frac{1}{2}\lg n$$

# Our New Constraints

- Query Time: O(1).
- Space Usage:

$$O\left(n \log\log n \cdot \left[\frac{1}{c'} + \frac{c'}{t'}\right]\right) + o(n) + n + o(n)$$

- ***Better Choices:***

$$c' = \sqrt{\lg n}$$

$$t' = \frac{1}{2}\lg n$$

# Our New Constraints

- Query Time: O(1).
- Space Usage:

$$\mathrm{O}\left(n \log\log n \cdot \left[\frac{1}{c'} + \frac{c'}{t'}\right]\right) + o(n) + n + o(n)$$

- ***Better Choices:***

$$c' = \sqrt{\lg n}$$

$$t' = \frac{1}{2}\lg n$$

# Our New Constraints

- Query Time: O(1).
- Space Usage:

$$\mathrm{O}\left(n \log\log n \cdot \left\lceil \frac{1}{\sqrt{\log n}} \right\rceil\right) + o(n) + n + o(n)$$

- ***Better Choices:***

$$c' = \sqrt{\lg n}$$

$$t' = \frac{1}{2} \lg n$$

# Our New Constraints

- Query Time: O(1).
- Space Usage:

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

- ***Better Choices:***

$$c' = \sqrt{\lg n}$$

$$t' = \frac{1}{2}\lg n$$

# Our New Constraints

- Query Time: O(1).
- Space Usage:

$$o(n) + o(n) + n + o(n)$$

- ***Better Choices:***

$$c' = \sqrt{\lg n}$$

$$t' = \frac{1}{2} \lg n$$

# Our New Constraints

- Query Time: O(1).
- Space Usage:

$$o(n) \quad + \quad o(n) \quad + \quad n \quad + \quad o(n)$$

- ***Better Choices:***

$$c' = \sqrt{\lg n}$$

$$t' = \frac{1}{2} \lg n$$

# Our New Constraints

- Query Time: O(1).
- Space Usage: ***n + o(n)***.

- ***Better Choices:***

$$c' = \sqrt{\lg n}$$

$$t' = \frac{1}{2} \lg n$$

# The Final Scorecard

| | Bits Needed | Query Time |
|---|---|---|
| Binary Search w/Jacobson Rank | $n + o(n)$ | $O(\log n)$ |
| Precompute-All | $O(m \log n)$ | $O(1)$ |
| Hybrid Precompute + Binary Search | $n + o(n)$ | $O(\log \log n)$ |
| Two-Level Hybrid + Binary Search | $n + o(n)$ | $O(\log \log \log n)$ |
| Two-Level Hybrid + Four Russians. * | $n + o(n)$ | $O(1)$ |
| Two-Level Hybrid + Four Russians (Clark) | $n + o(n)$ | $O(1)$ |

** technically*

# Summary for Today

- Splitting a problem into smaller pieces is a great way to reduce space in succinct structures, and those pieces do not necessarily have to be fixed-sized blocks.
- Recognizing that different styles of solutions work at different scales makes it possible to solve hard problems by combining each approach together.
- Four-Russians speedups are common in succinct data structures.
- Adding an intermediate level of subdivision between the top-level structure and Four Russians problems can substantially reduce space usage.

# Next Time

- ***Pairwise Independent Hashing***
  - Families of hash functions with lovely Theoryland properties.
- ***Frequency Estimation***
  - Guessing how many times we've seen items without writing them all down.
- ***Count-Min Sketches***
  - A simple and beautiful data structure.