Welcome to CS166!

- Course information handout available up front.
- Today:
  - Course overview.
  - Why study data structures?
  - The range minimum query problem.

Course Staff

Keith Schwarz (htiek@cs.stanford.edu)

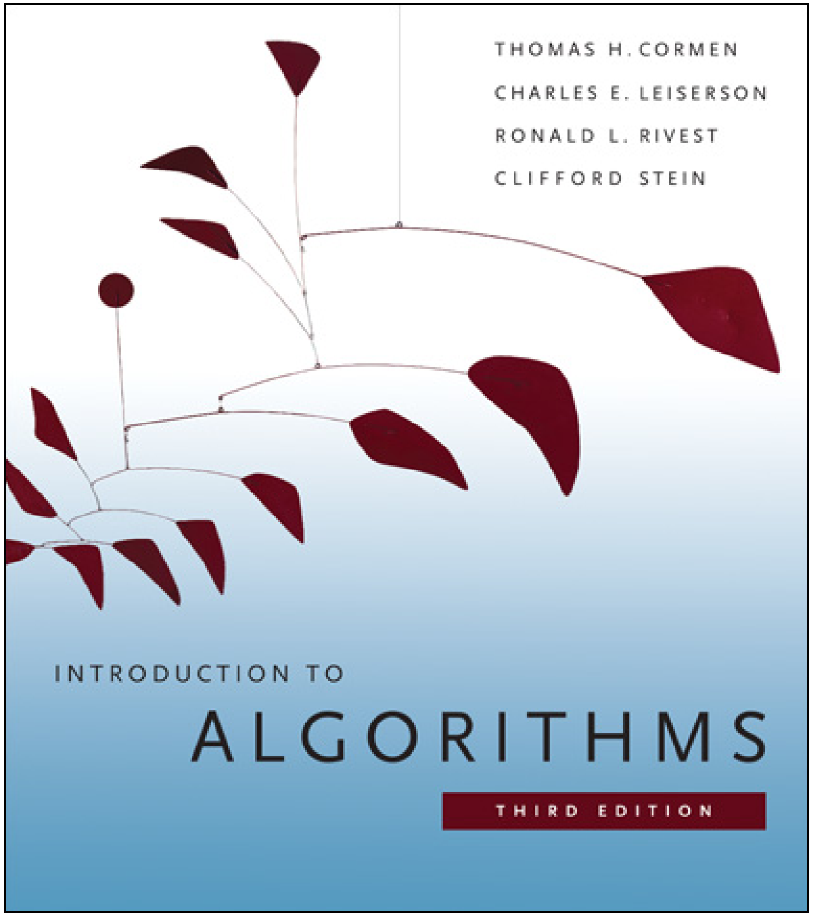
Kyle Brogle (broglek@stanford.edu)
Daniel Hollingshead (dhollingshead@stanford.edu)
Nick Isaacs (nisaacs@stanford.edu)
Aparna Krishnan (aparnak@stanford.edu)
Sen Wu (senwu@stanford.edu)

Course Staff Mailing List:
cs166-spr1314-staff@lists.stanford.edu

The Course Website

http://cs166.stanford.edu

Required Reading



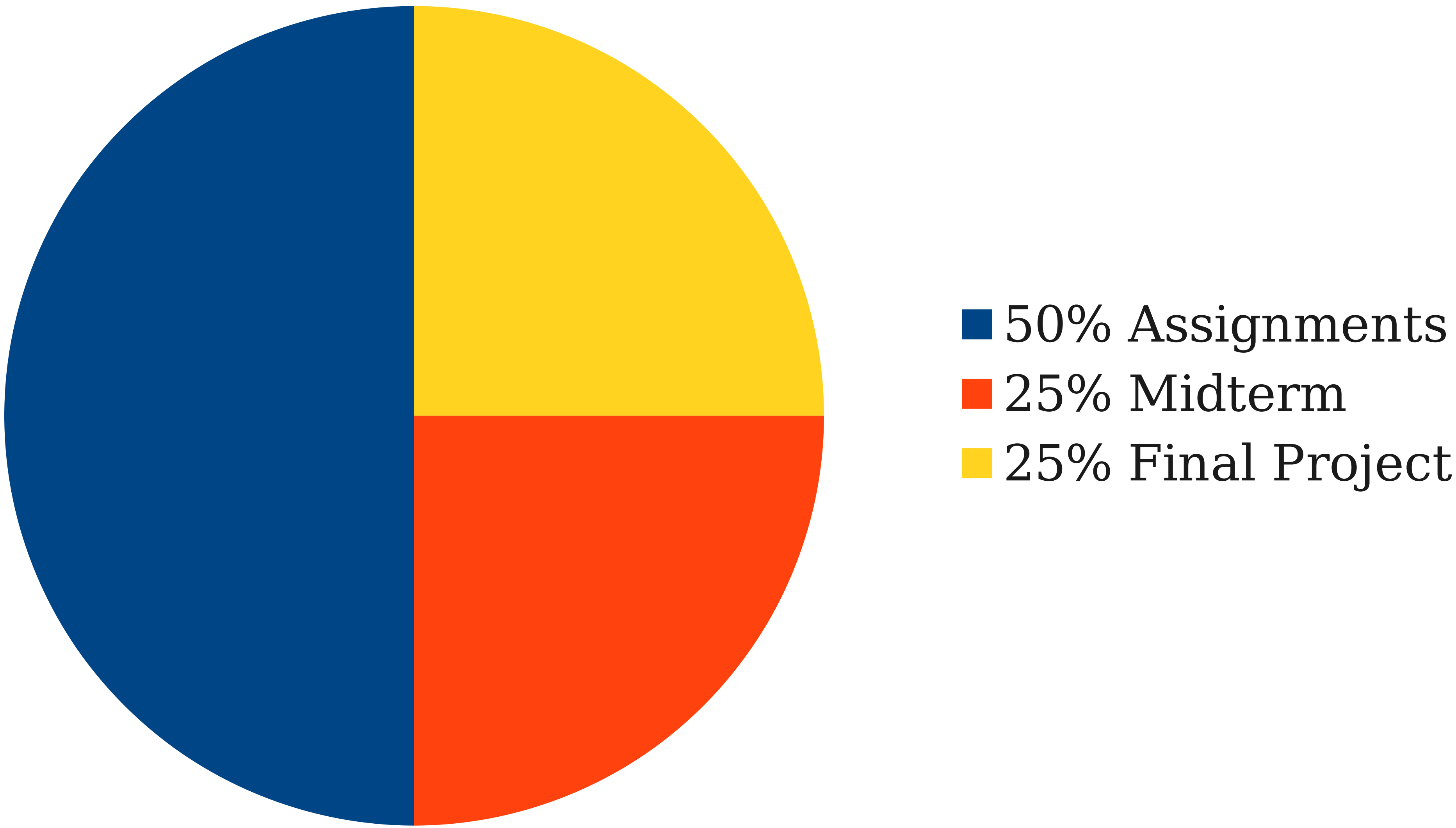
- *Introduction to Algorithms, Third Edition* by Cormen, Leiserson, Rivest, and Stein.
- You'll want the third edition for this course.
- Available in the bookstore; several copies on hold at the Engineering Library.

Prerequisites

- **CS161** (Design and Analysis of Algorithms)
  - We'll assume familiarity with asymptotic notation, correctness proofs, algorithmic strategies (e.g. divide-and-conquer), classical algorithms, recurrence relations, etc.
- **CS107** (Computer Organization and Systems)
  - We'll assume comfort working from the command-line, designing and testing nontrivial programs, and manipulating bitwise representations of data. You should have some knowledge of the memory hierarchy.
- Not sure whether you're in the right place? Please feel free to ask!

Grading Policies

Grading Policies



Axess: “Enrollment Limited”

- Because this is a new course, we're limiting enrollment in CS166 to 100.
- If you are interested in taking the course, please sign up on Axess as soon as possible so that we can get an approximate headcount.
- If enrollment is under 100, then everything will work as a normal course.
- If enrollment exceeds 100, we'll send out an application. Sorry for the inconvenience!

Why Study Data Structures?

Why Study Data Structures?

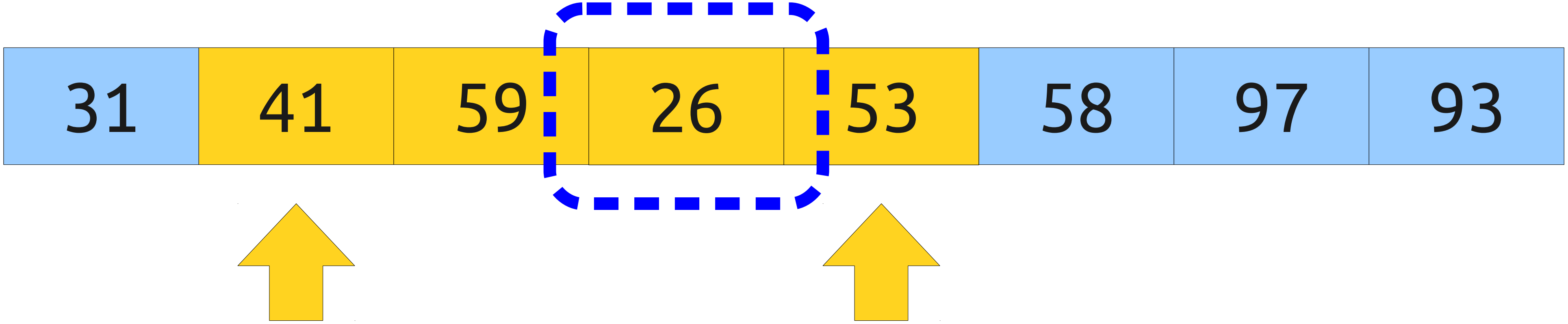
- **Explore the intersection between theory and practice.**
- **Learn new approaches to modeling and solving problems.**
- **Expand your sense of what can be done efficiently.**

Range Minimum Queries

The RMQ Problem

- The **Range Minimum Query** (**RMQ**) problem is the following:

  Given a fixed array A and two indices $i \leq j$, what is the smallest element out of A[i], A[i + 1], …, A[j – 1], A[j]?

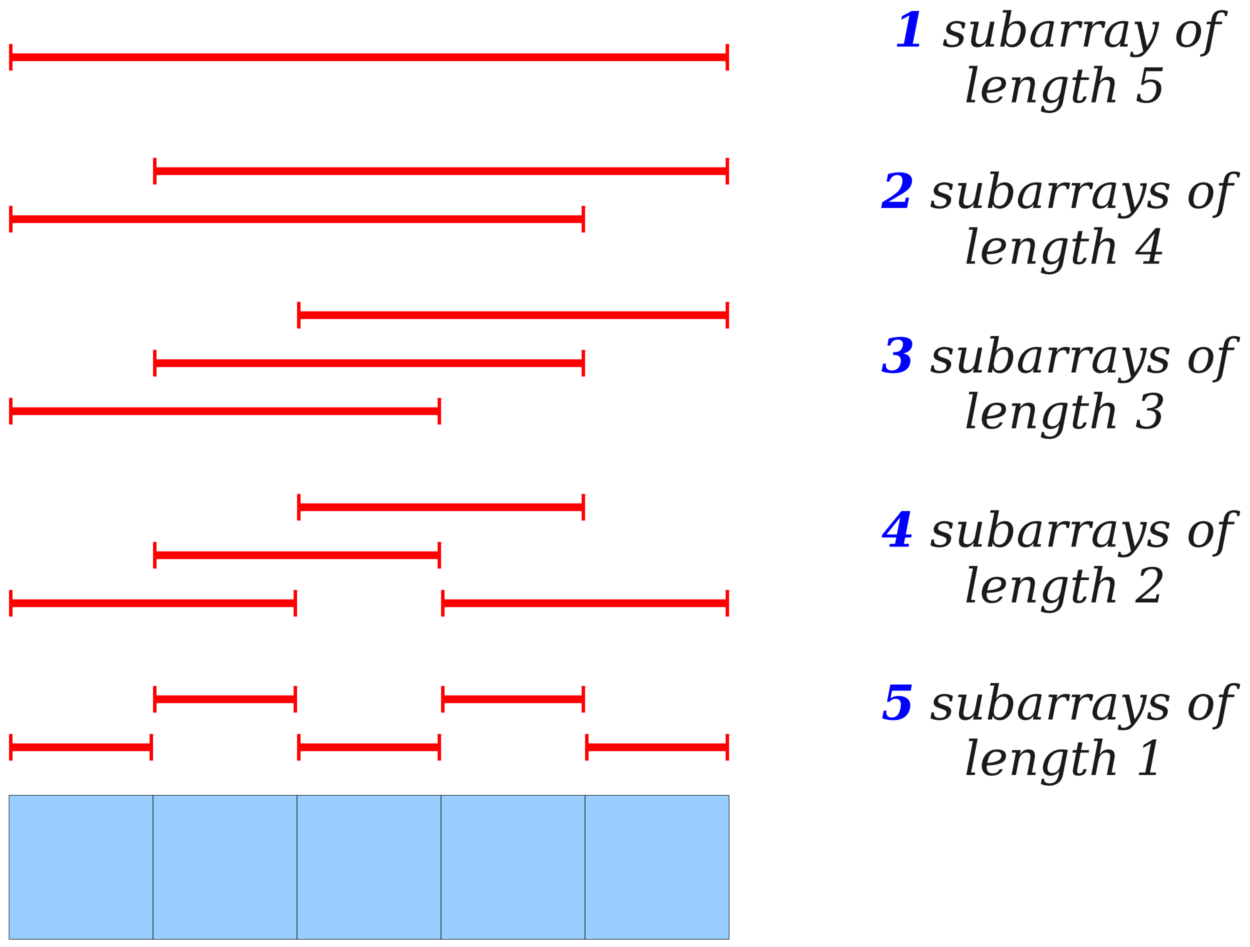


A Trivial Solution

- There's a simple O(n)-time algorithm for evaluating $\mathrm{RMQ}_A(i, j)$: just iterate across the elements between i and j, inclusive, and take the minimum!
- Why is this problem at all algorithmically interesting?
- Suppose that the array A is fixed and we'll make k queries on it.
- Can we do better than the naïve algorithm?

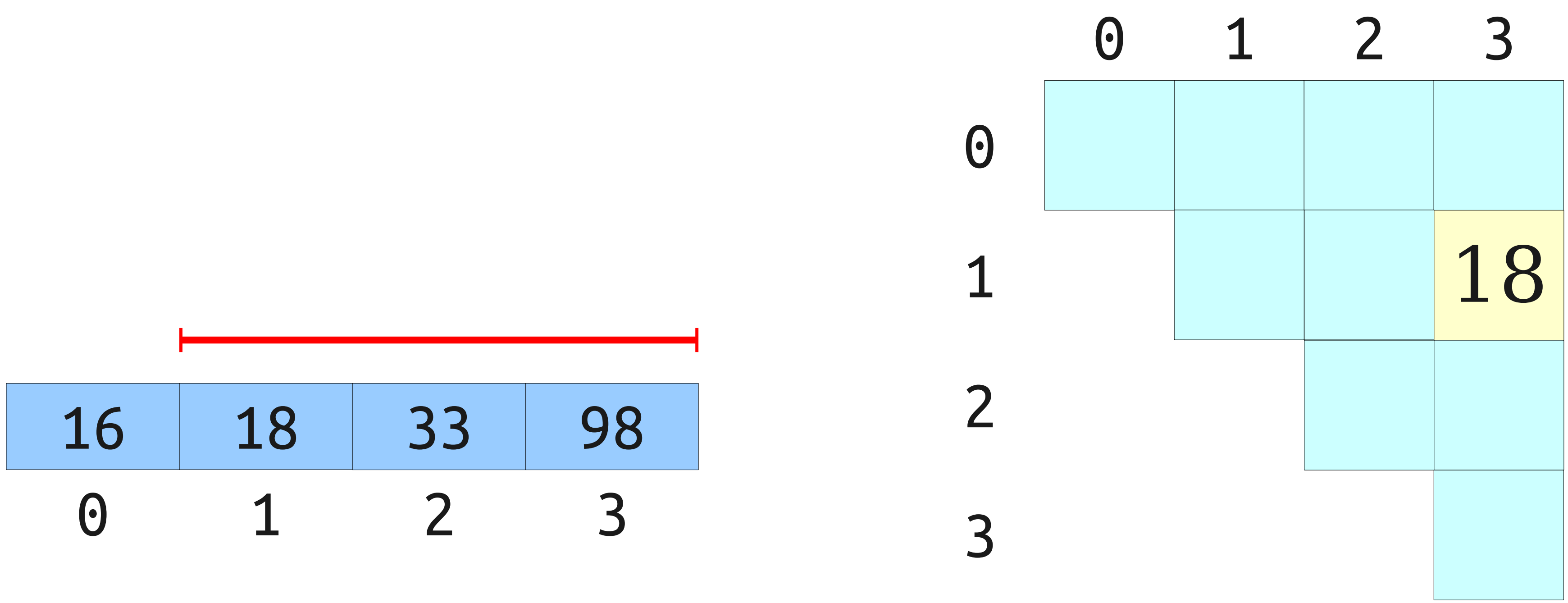
An Observation

- In an array of length *n*, there are only $\Theta(n^2)$ possible queries.
- Why?



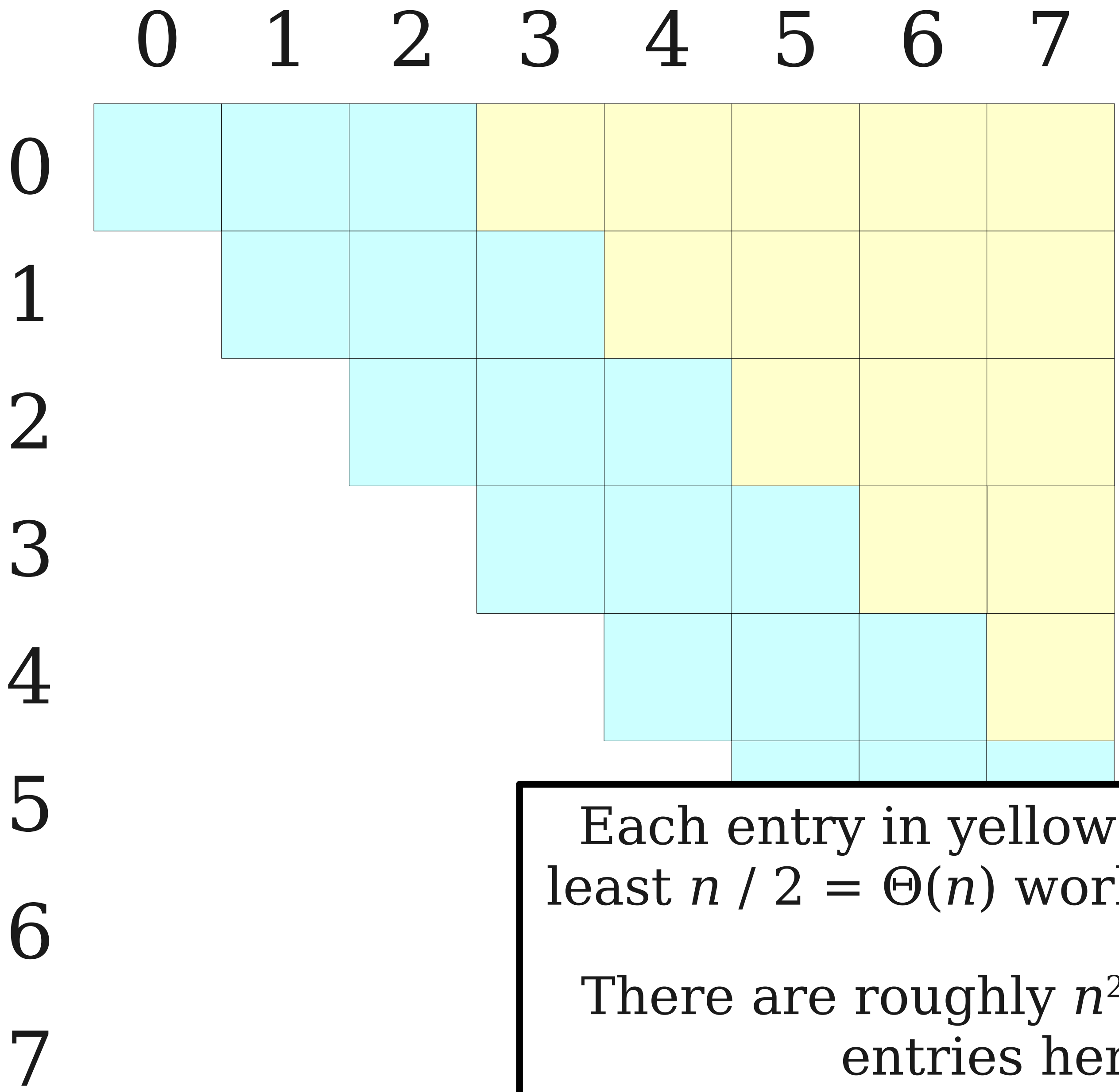
A Different Approach

- There are only $\Theta(n^2)$ possible RMQs in an array of length n.
- If we precompute all of them, we can answer RMQ in time O(1) per query.



Building the Table

- One simple approach: for each entry in the table, iterate over the range in question and find the minimum value.
- How efficient is this?
  - Number of entries: $\Theta(n^2)$.
  - Time to evaluate each entry: $O(n)$.
  - Time required: $O(n^3)$.
- The runtime is $O(n^3)$ using this approach. Is it also $\Theta(n^3)$?



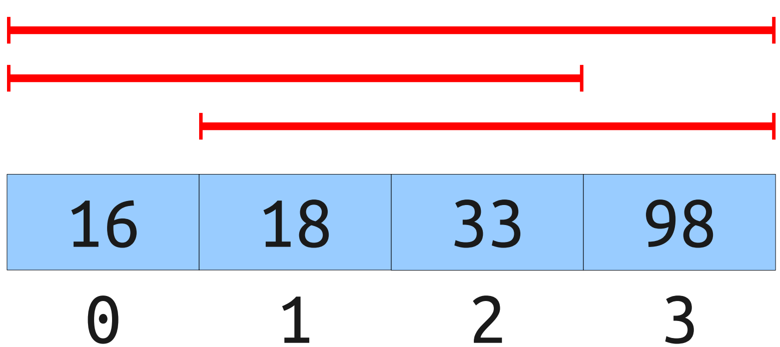
Each entry in yellow requires at least n / 2 = $\Theta(n)$ work to evaluate.

There are roughly n^2 / 8 = $\Theta(n^2)$ entries here.

Total work required: $\Theta(n^3)$

A Different Approach

- Naïvely precomputing the table is inefficient.
- Can we do better?
- **Claim:** Can precompute all subarrays in time $\Theta(n^2)$ using dynamic programming.



	0	1	2	3
0	16	16	16	★
1		18	18	18
2			33	33
3				98

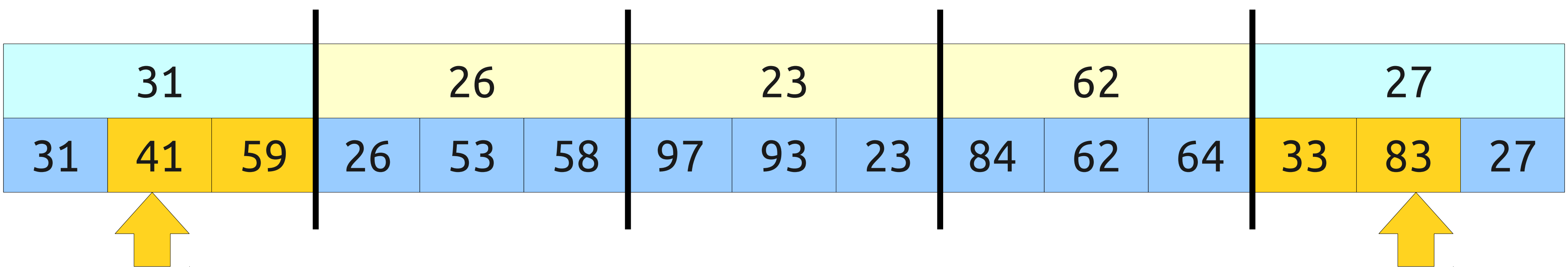
Some Notation

- We'll say that an RMQ data structure has time complexity **$\langle p(n), q(n) \rangle$** if
  - preprocessing takes time at most $p(n)$ and
  - queries take time at most $q(n)$.
- We now have two RMQ data structures:
  - $\langle O(1), O(n) \rangle$ with no preprocessing.
  - $\langle O(n^2), O(1) \rangle$ with full preprocessing.
- These are two extremes on a curve of tradeoffs: no preprocessing versus full preprocessing.
- **Question:** *Is there a "golden mean" between these extremes?*

Another Approach: **Block Decomposition**

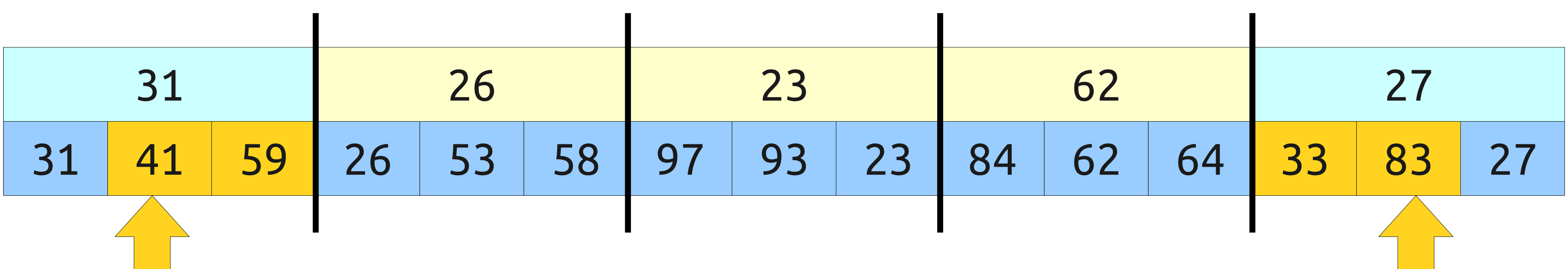
A Block-Based Approach

- Split the input into O(n / b) blocks of some “block size” b.
  - Here, b = 3.
- Compute the minimum value in each block.



Analyzing the Approach

- Let's analyze this approach in terms of n and b.
- Preprocessing time:
  - $O(b)$ work on $O(n / b)$ blocks to find minimums.
  - Total work: **$O(n)$**.
- Time to query $\mathrm{RMQ}_A(i, j)$:
  - $O(1)$ work to find block indices (divide by block size).
  - $O(b)$ work to scan inside i and j's blocks.
  - $O(n / b)$ work looking at block minimums between i and j.
  - Total work: **$O(b + n / b)$**.



Intuiting O($\boldsymbol{b}$ + $\boldsymbol{n / b}$)

- As b increases:
  - The $\boldsymbol{b}$ term rises (more elements to scan within each block).
  - The $\boldsymbol{n / b}$ term drops (fewer blocks to look at).
- As b decreases:
  - The $\boldsymbol{b}$ term drops (fewer elements to scan within a block).
  - The $\boldsymbol{n / b}$ term rises (more blocks to look at).
- Is there an optimal choice of b given these constraints?

Optimizing *b*

- What choice of b minimizes $b + n / b$?
- Start by taking the derivative:

$$\frac{d}{db}\left(b+n/b\right) \quad = \quad 1-\frac{n}{b^2}$$

- Setting the derivative to zero:

$$\begin{array}{ccc} 1-n/b^2 & = & 0 \\ 1 & = & n/b^2 \\ b^2 & = & n \\ b & = & \sqrt{n} \end{array}$$

- Asymptotically optimal runtime is when $b = n^{1/2}$.
- In that case, the runtime is

  $O(b + n / b) = O(n^{1/2} + n / n^{1/2}) = O(n^{1/2} + n^{1/2}) = \mathbf{O(n^{1/2})}$

Summary of Approaches

- Three solutions so far:
  - No preprocessing: $\langle O(1), O(n)\rangle$.
  - Full preprocessing: $\langle O(n^2), O(1)\rangle$.
  - Block partition: $\langle O(n), O(n^{1/2})\rangle$.
- Modest preprocessing yields modest performance increases.
- **Question:** Can we do better?

A Second Approach: **Sparse Tables**

An Intuition

- The $\langle O(n^2), O(1) \rangle$ solution gives fast queries because every range we might look up has already been precomputed.
- This solution is slow overall because we have to compute the minimum of every possible range.
- **Question:** Can we still get O(1) queries without preprocessing all possible ranges?

An Observation

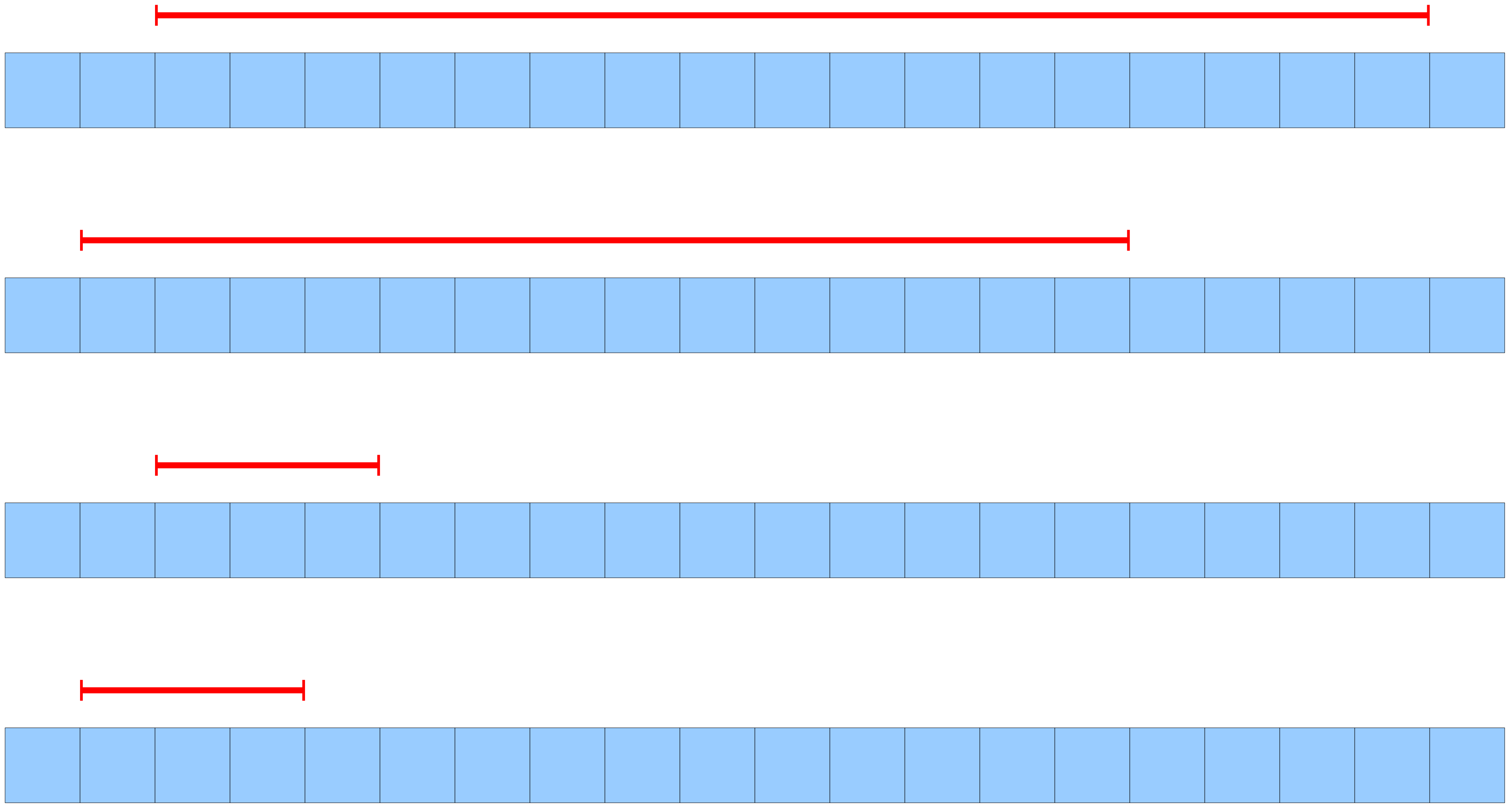
31	41	59	26	53	58	97	93
0	1	2	3	4	5	6	7

	0	1	2	3	4	5	6	7
0	31	31	31	26				
1		41	41	26	26			
2			59	26	26	26		★
3				26	26	26	26	
4					53	53	53	53
5						58	58	58
6							97	93
7								93

The Intuition

- It's still possible to answer any query in time O(1) without precomputing RMQ over all ranges.
- If we precompute the answers over too many ranges, the preprocessing time will be too large.
- If we precompute the answers over too few ranges, the query time won't be O(1).
- **Goal:** Precompute RMQ over a set of ranges such that
  - There are $o(n^2)$ total ranges, but
  - there are enough ranges to support O(1) query times.

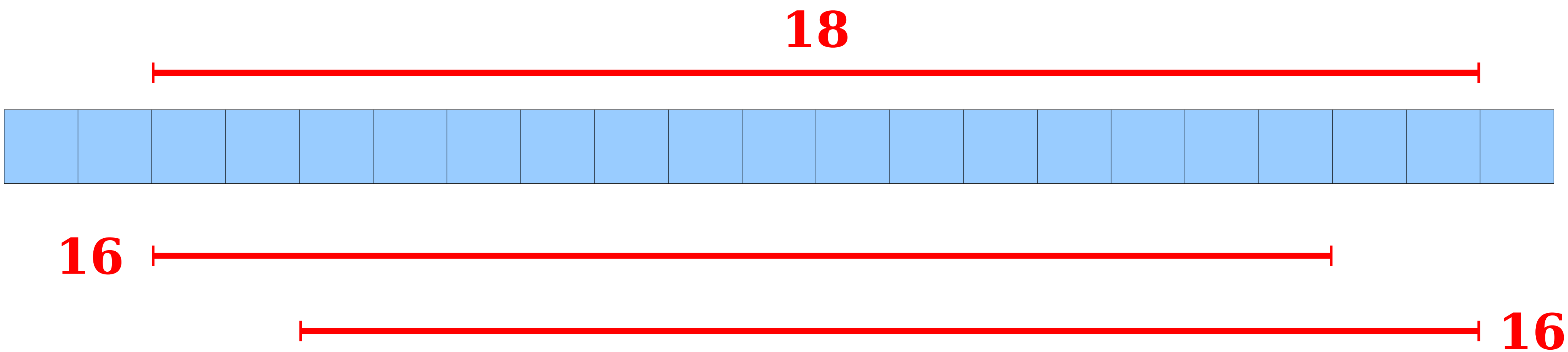
Some Observations



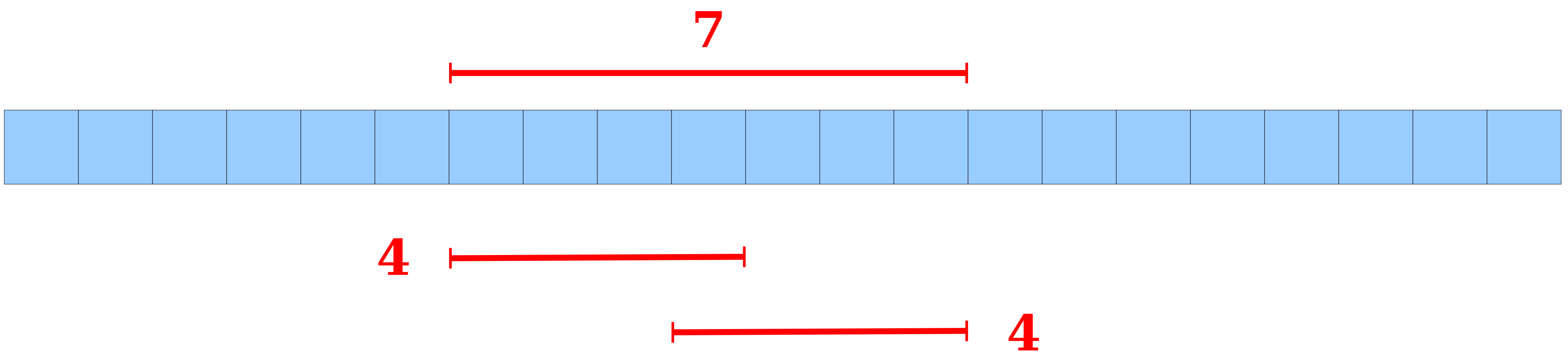
The Approach

- For each index i, compute RMQ for ranges starting at i of size 1, 2, 4, 8, 16, …, 2^k as long as they fit in the array.
  - Gives both large and small ranges starting at any point in the array.
  - Only O(log n) ranges computed for each array element.
  - Total number of ranges: O(n log n).
- **Claim:** Any range in the array can be formed as the union of two of these ranges.

Creating Ranges



Creating Ranges



Doing a Query

- To answer $\mathrm{RMQ}_A(i, j)$:
  - Find the largest k such that $2^k \leq j - i + 1$.
    - With the right preprocessing, this can be done in time O(1); you'll figure out how in the problem set!
  - The range $[i, j]$ can be formed as the overlap of the ranges $[i, i + 2^k - 1]$ and $[j - 2^k + 1, j]$.
  - Each range can be looked up in time O(1).
  - Total time: **O(1)**.

Precomputing the Ranges

- There are O(n log n) ranges to precompute.
- Using dynamic programming, we can compute all of them in time O(n log n).

31	41	59	26	53	58	97	93
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**

	2^0	2^1	2^2	2^3
0	31	31	★	
1	41	41		
2	59	26		
3	26	26		
4	53	53		
5	58	58		
6	97	93		
7	93			

Sparse Tables

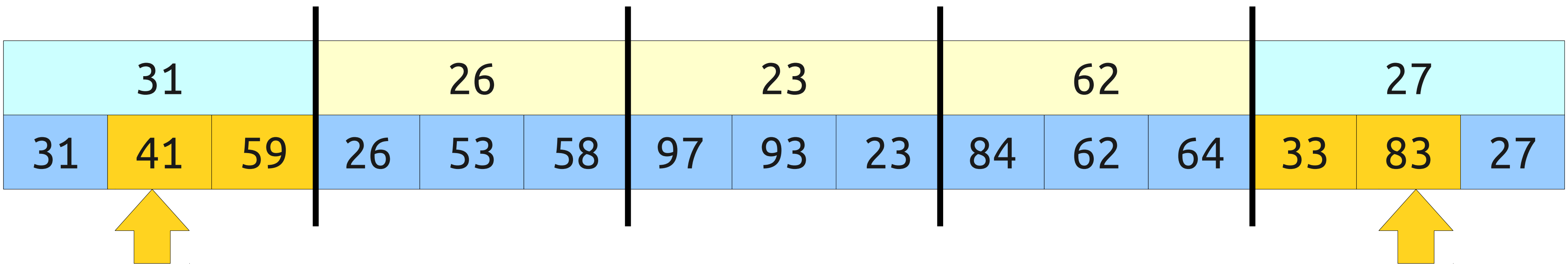
- This data structure is called a **sparse table**.
- Gives an **$\langle O(n \log n), O(1) \rangle$** solution to RMQ.
- Asymptotically better than precomputing all possible ranges!

The Story So Far

- We now have the following solutions for RMQ:
  - Precompute all: $\langle O(n^2), O(1) \rangle$.
  - Precompute none: $\langle O(1), O(n) \rangle$.
  - Blocking: $\langle O(n), O(n^{1/2}) \rangle$.
  - Sparse table: $\langle O(n \log n), O(1) \rangle$.
- ***Can we do better?***

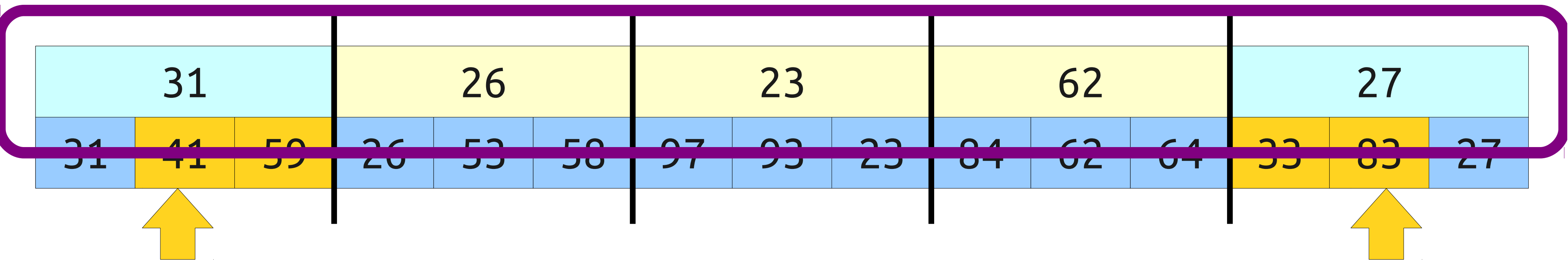
A Third Approach: **Hybrid Strategies**

Blocking Revisited

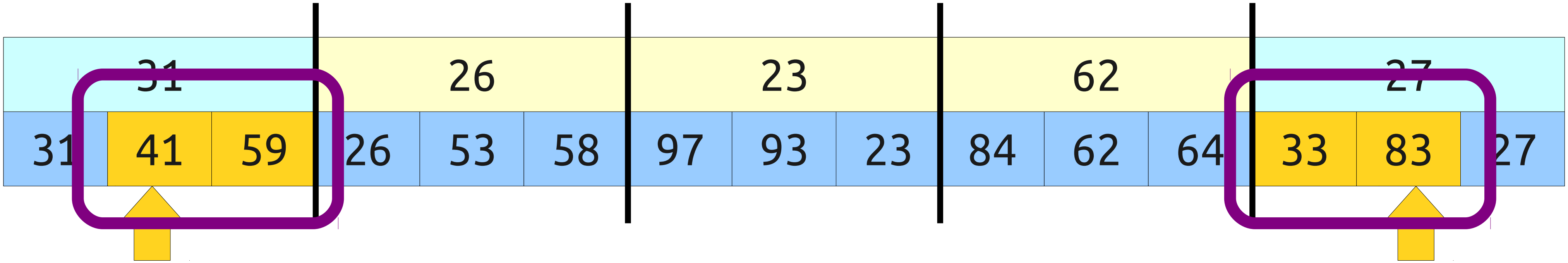


Blocking Revisited

This is just RMQ on the block minimums!



Blocking Revisited



This is just RMQ inside the blocks!

The Setup

- Here's a new possible route for solving RMQ:
  - Split the input into blocks of some block size b.
  - For each of the $O(n / b)$ blocks, compute the minimum.
  - **Construct an RMQ structure on the block minimums.**
  - **Construct RMQ structures on each block.**
  - Combine the RMQ answers to solve RMQ overall.
- This approach of segmenting a structure into a high-level structure and many low-level structures is sometimes called a **macro/micro decomposition**.

Combinations and Permutations

- The macro/micro decomposition isn't a single data structure; it's a *framework* for data structures.
- We get to choose
  - the block size,
  - which RMQ structure to use on top, and
  - which RMQ structure to use for the blocks.
- Summary and block RMQ structures don't have to be the same type of RMQ data structure – we can combine different structures together to get different results.

The Framework

- Suppose we use a $\langle p_1(n), q_1(n)\rangle$-time RMQ solution for the block minimums and a $\langle p_2(n), q_2(n)\rangle$-time RMQ solution within each block.
- Let the block size be b.
- In the hybrid structure, the preprocessing time is

$$\mathbf{O}(n + p_1(n / b) + (n / b)\ p_2(b))$$

- The query time is

$$\mathbf{O}(q_1(n / b) + q_2(b))$$

31			26			23			62			27		
31	41	59	26	53	58	97	93	23	84	62	64	33	83	27

A Sanity Check

- The $\langle O(n), O(n^{1/2}) \rangle$ block-based structure from earlier uses this framework with the $\langle O(1), O(n) \rangle$ no-preprocessing RMQ structure and $b = n^{1/2}$.
- According to our formulas, the preprocessing time should be

$$\begin{aligned} & O(n + p_1(n / b) + (n / b)\, p_2(b)) \\ &= O(n + 1 + n / b) \\ &= \mathbf{O(n)} \end{aligned}$$

- The query time should be

$$\begin{aligned} & O(q_1(n / b) + q_2(b)) \\ &= O(n / b + b) \\ &= \mathbf{O(n^{1/2})} \end{aligned}$$

- Looks good so far!

For Reference

$$p_1(n) = 1$$
$$q_1(n) = n$$

$$p_2(n) = 1$$
$$q_2(n) = n$$

$$b = n^{1/2}$$

An Observation

- A sparse table takes time $O(n \log n)$ to construct on an array of n elements.
- With block size b, there are $O(n / b)$ total blocks.
- Time to construct a sparse table over the block minimums: $O((n / b) \log (n / b))$.
- Since $\log (n / b) = O(\log n)$, the time to build the sparse table is at most $O((n / b) \log n)$.
- **Cute trick:** If $b = \Theta(\log n)$, the time to construct a sparse table over the minimums is

$$O((n / b) \log n) = O((n / \log n) \log n) = \mathbf{O(n)}$$

One Possible Hybrid

- Set the block size to log n.
- Use a sparse table for the top-level structure.
- Use the "no preprocessing" structure for each block.
- Preprocessing time:

$$\begin{aligned} & O(n + p_1(n / b) + (n / b)\ p_2(b)) \\ = {} & O(n + n + n / \log n) \\ = {} & \mathbf{O(n)} \end{aligned}$$

- Query time:

$$\begin{aligned} & O(q_1(n / b) + q_2(b)) \\ = {} & O(1 + \log n) \\ = {} & \mathbf{O(\log n)} \end{aligned}$$

- An **⟨O(n), O(log n)⟩** solution!

For Reference

$p_1(n) = n \log n$
$q_1(n) = 1$

$p_2(n) = 1$
$q_2(n) = n$

$b = \log n$

Another Hybrid

- Let's suppose we use the $\langle O(n \log n), O(1)\rangle$ sparse table for both the top and bottom RMQ structures with a block size of $\log n$.
- The preprocessing time is

$$\begin{aligned} & O(n + p_1(n / b) + (n / b)\ p_2(b)) \\ = \ & O(n + n + (n / \log n)\ b \log b) \\ = \ & O(n + (n / \log n) \log n \log \log n) \\ = \ & \mathbf{O(n \log \log n)} \end{aligned}$$

- The query time is

$$\begin{aligned} & O(q_1(n / b) + q_2(b)) \\ = \ & \mathbf{O(1)} \end{aligned}$$

- We have an $\mathbf{\langle O(n \log \log n), O(1)\rangle}$ solution to RMQ!

For Reference

$p_1(n) = n \log n$
$q_1(n) = 1$

$p_2(n) = n \log n$
$q_2(n) = 1$

$b = \log n$

One Last Hybrid

- Suppose we use a sparse table for the top structure and the ⟨O(n), O(log n)⟩ solution for the bottom structure. Let's choose b = log n.
- The preprocessing time is

$$\begin{aligned} & O(n + p_1(n / b) + (n / b)\, p_2(b)) \\ = \ & O(n + n + (n / \log n)\, b) \\ = \ & O(n + n + (n / \log n) \log n) \\ = \ & \mathbf{O(n)} \end{aligned}$$

- The query time is

$$\begin{aligned} & O(q_1(n / b) + q_2(b)) \\ = \ & O(1 + \log \log n) \\ = \ & \mathbf{O(\log \log n)} \end{aligned}$$

- We have an **⟨O(n), O(log log n)⟩** solution to RMQ!

For Reference

$p_1(n) = n \log n$
$q_1(n) = 1$

$p_2(n) = n$
$q_2(n) = \log n$

$b = \log n$

Where We Stand

- We've seen a bunch of RMQ structures today:
  - No preprocessing: $\langle O(1), O(n) \rangle$
  - Full preprocessing: $\langle O(n^2), O(1) \rangle$
  - Block partition: $\langle O(n), O(n^{1/2}) \rangle$
  - Sparse table: $\langle O(n \log n), O(1) \rangle$
  - Hybrid 1: $\langle O(n), O(\log n) \rangle$
  - Hybrid 2: $\langle O(n \log \log n), O(1) \rangle$
  - Hybrid 3: $\langle O(n), O(\log \log n) \rangle$

Is there an $\langle O(n), O(1) \rangle$ solution to RMQ?

Yes!

Next Time

- **Cartesian Trees**
  - A data structure closely related to RMQ.
- **The Method of Four Russians**
  - A technique for shaving off log factors.
- **The Fischer-Heun Structure**
  - A deceptively simple, asymptotically optimal RMQ structure.