Range Minimum Queries

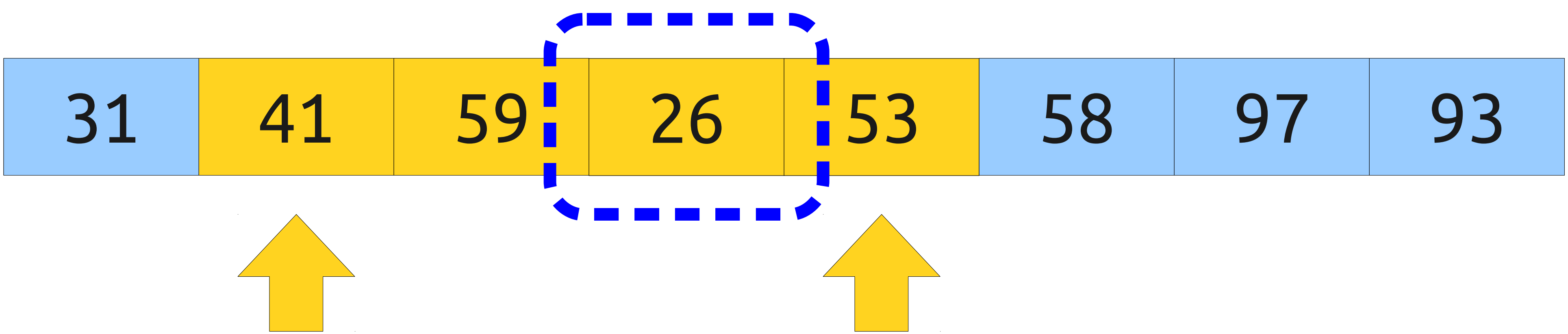
Part Two

Recap from Last Time

The RMQ Problem

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

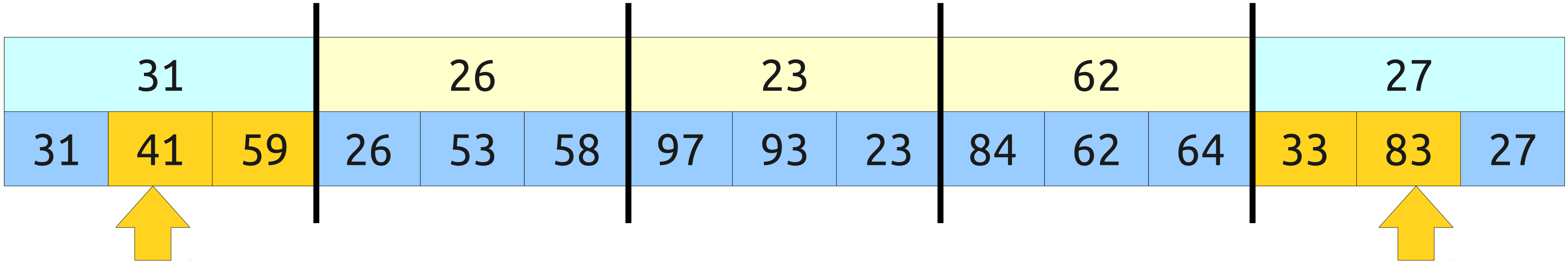
Given a fixed array A and two indices $i \leq j$, what is the smallest element out of $A[i], A[i+1], \ldots, A[j-1], A[j]$?



Some Notation

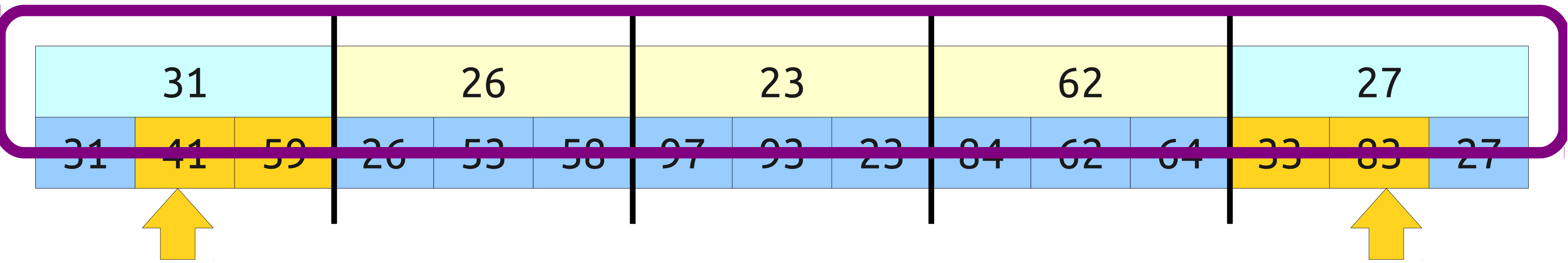
- We'll say that an RMQ data structure has time complexity **$\langle p(n), q(n) \rangle$** if
  - preprocessing takes time at most $p(n)$ and
  - queries take time at most $q(n)$.
- Last time, we saw structures with the following runtimes:
  - $\langle O(n^2), O(1) \rangle$ (full preprocessing)
  - $\langle O(n \log n), O(1) \rangle$ (sparse table)
  - $\langle O(n \log \log n), O(1) \rangle$ (hybrid approach)
  - $\langle O(n), O(n^{1/2}) \rangle$ (blocking)
  - $\langle O(n), O(\log n) \rangle$ (hybrid approach)
  - $\langle O(n), O(\log \log n) \rangle$ (hybrid approach)

Blocking Revisited

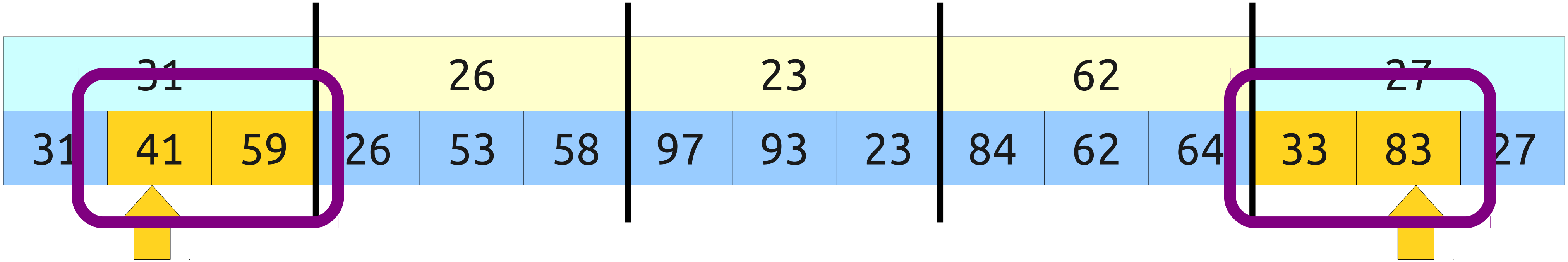


Blocking Revisited

This is just RMQ on the block minimums!



Blocking Revisited



This is just RMQ inside the blocks!

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 Useful Observation

- Sparse tables can be constructed in time O(n log n).
- If we use a sparse table as a top structure, construction time is O((n / b) log n).
  - See last lecture for the math on this.
- **Cute trick:** If we choose $b = \Theta(\log n)$, then the construction time is **O(*n*)**.

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

Yes!

An Observation

The Limits of Hybrids

- The preprocessing time on a hybrid structure is

$$O(n + p_1(n / b) + (n / b)\ p_2(b))$$

- The query time is

$$O(q_1(n / b) + q_2(b))$$

- For this to be ⟨O(*n*), O(1)⟩, we need to have $p_2(n) = O(n)$ and $q_2(n) = O(1)$.
- **We can't build an optimal solution out of the hybrid approach unless we already have one!**
- ***Or can we?***

A Key Difference

- Our original problem is

Solve RMQ on a single array in time $\langle O(n), O(1)\rangle$

- The new problem is

Solve RMQ on a large number of small arrays with O(1) query time and *average* preprocessing time O(*n*).

- These are not the same problem.
- **Question:** Why is this second problem any easier than the first?

An Observation

10	30	20	40

166	361	261	464

Claim: The indices of the answers to any range minimum queries on these two arrays are the same.

Modifying RMQ

- From this point forward, let's have $\text{RMQ}_A(i, j)$ denote the **index** of the minimum value in the range rather than the value itself.
- **Observation:** If RMQ structures return indices rather than values, we can use a single RMQ structure for both of these arrays:

10	30	20	40

166	361	261	464

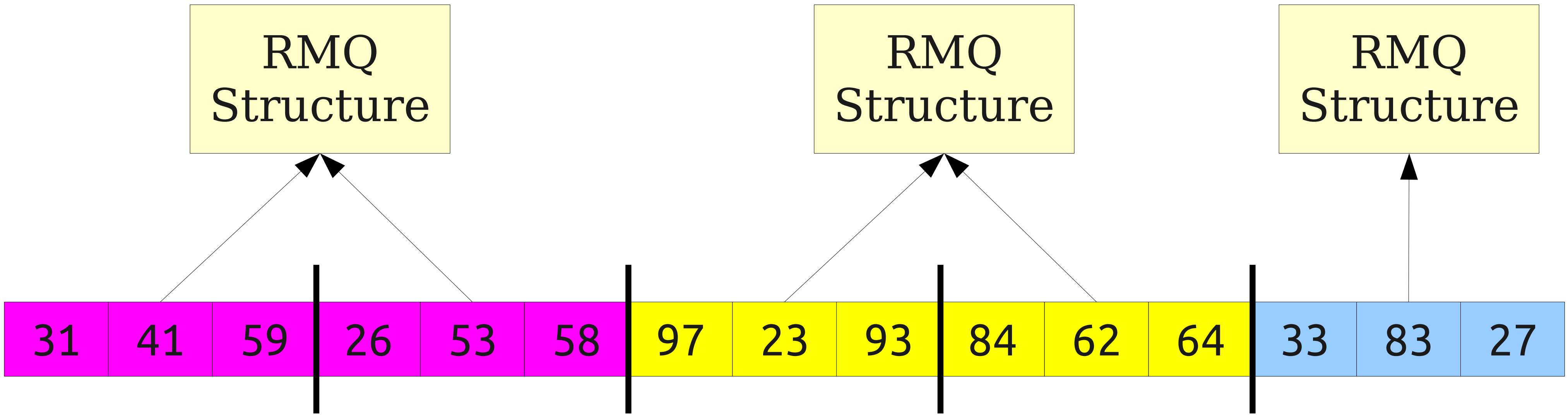
Some Notation

- Let B_1 and B_2 be blocks of length b.
- We'll say that B_1 and B_2 **have the same block type** (denoted $\boldsymbol{B_1 \sim B_2}$) if the following holds:

$$\textbf{For all } 0 \leq i \leq j < b\textbf{:}$$
$$\mathrm{RMQ}_{B_1}(i, j) = \mathrm{RMQ}_{B_2}(i, j)$$

- Intuitively, the RMQ answers for B_1 are always the same as the RMQ answers for B_2.
- If we precompute RMQ over B_1, we can reuse that RMQ structure on B_2 iff $B_1 \sim B_2$.

An Observation



The Big Picture

- We're building up toward a hybrid structure that works as follows. For each block:
  - Determine the type of that block.
  - If an RMQ structure already exists for its block type, just use that structure.
  - Otherwise, compute its RMQ structure and store it for later.
- Need to choose the block size such that
  - there are “not too many” possible block types, which ensures we reuse RMQ structures, but
  - the blocks aren't so small that we can't efficiently build the summary structure on top.

Detecting Block Types

- For this approach to work, we need to be able to check whether two blocks have the same block type.
- **Problem:** Our formal definition of $B_1 \sim B_2$ is defined in terms of RMQ.
  - Not particularly useful *a priori*; we don't want to have to compute RMQ structures on B_1 and B_2 to decide whether they have the same block type!
- Is there a simpler way to determine whether two blocks have the same type?

An Initial Idea

- Since the elements of the array are ordered and we're looking for the smallest value in certain ranges, we might look at the permutation types of the blocks.

31	41	59	16	18	3	27	18	28	66	73	84
1	2	3	2	3	1	2	1	3	1	2	3
12	2	5	66	26	6	60	22	14	72	99	27
3	1	2	3	2	1	3	1	2	2	3	1

- **Claim:** If B_1 and B_2 have the same permutation on their elements, then $B_1 \sim B_2$.

Some Problems

- There are two main problems with this approach.
- **Problem One:** It's possible for two blocks to have different permutations but the same block type.
- All three of these blocks have the same block type but different permutation types:

261	268	161	167	166	167	261	161	268	166	166	268	161	261	167
4	5	1	3	2	3	4	1	5	2	2	5	1	4	3

- **Problem Two:** The number of possible permutations of a block is $b!$.
  - b has to be absolutely minuscule for $b!$ to be small.
- Is there a better criterion we can use?

An Observation

- **Claim:** If $B_1 \sim B_2$, the minimum elements of B_1 and B_2 must occur at the same position.

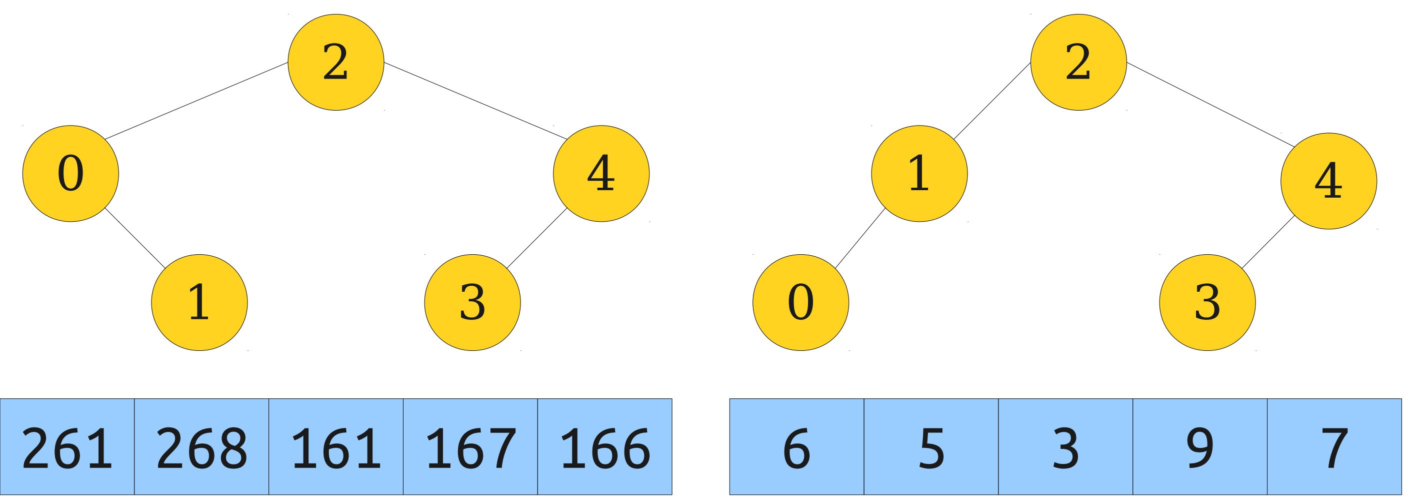
261	268	161	167	166

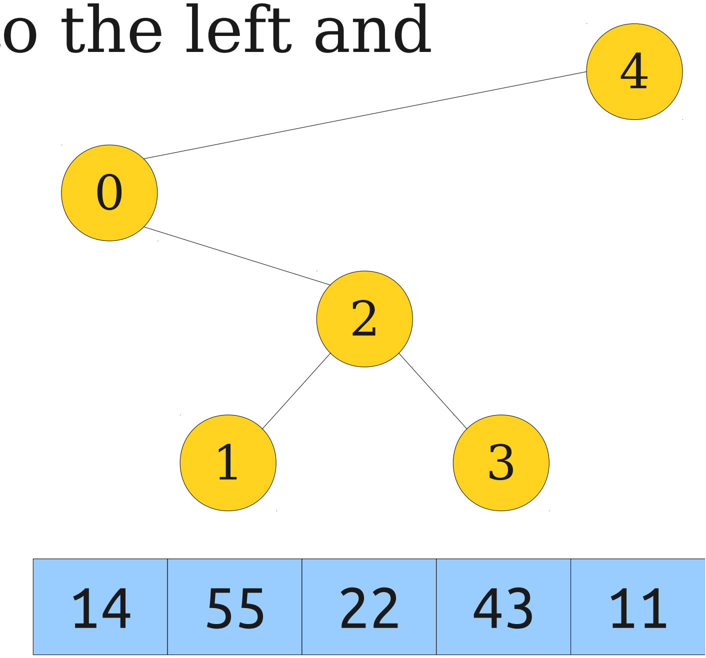
14	22	11	43	35

- **Claim:** This property must hold recursively on the subarrays to the left and right of the minimum.

Cartesian Trees

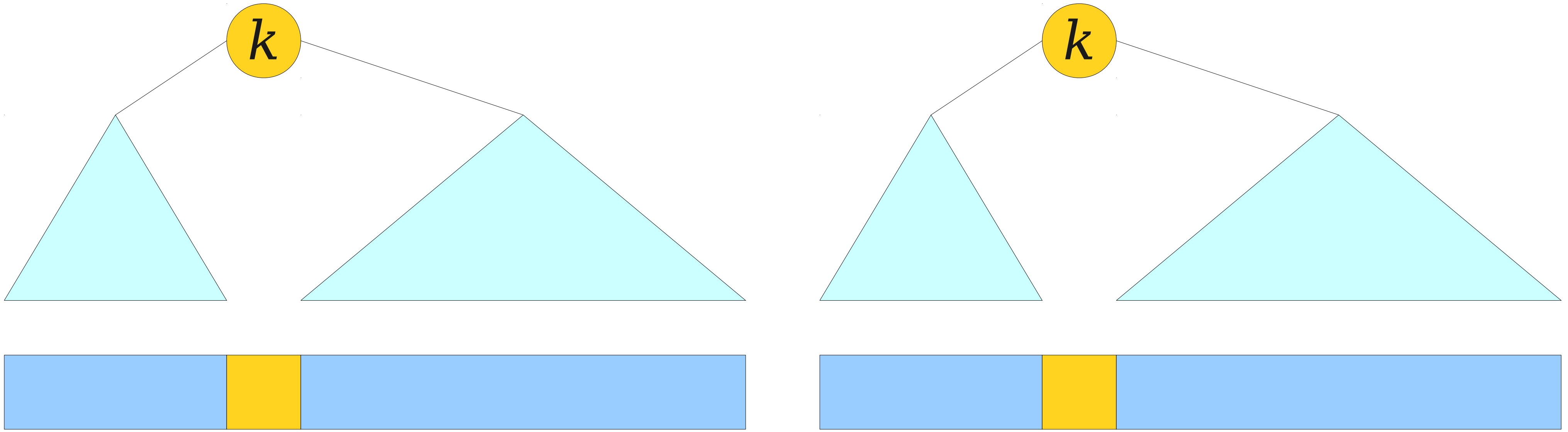
- A **Cartesian tree** is a binary tree derived from an array and defined as follows:
  - The empty array has an empty Cartesian tree.
  - For a nonempty array, the root stores the index of the minimum value. Its left and right children are Cartesian trees for the subarrays to the left and right of the minimum.





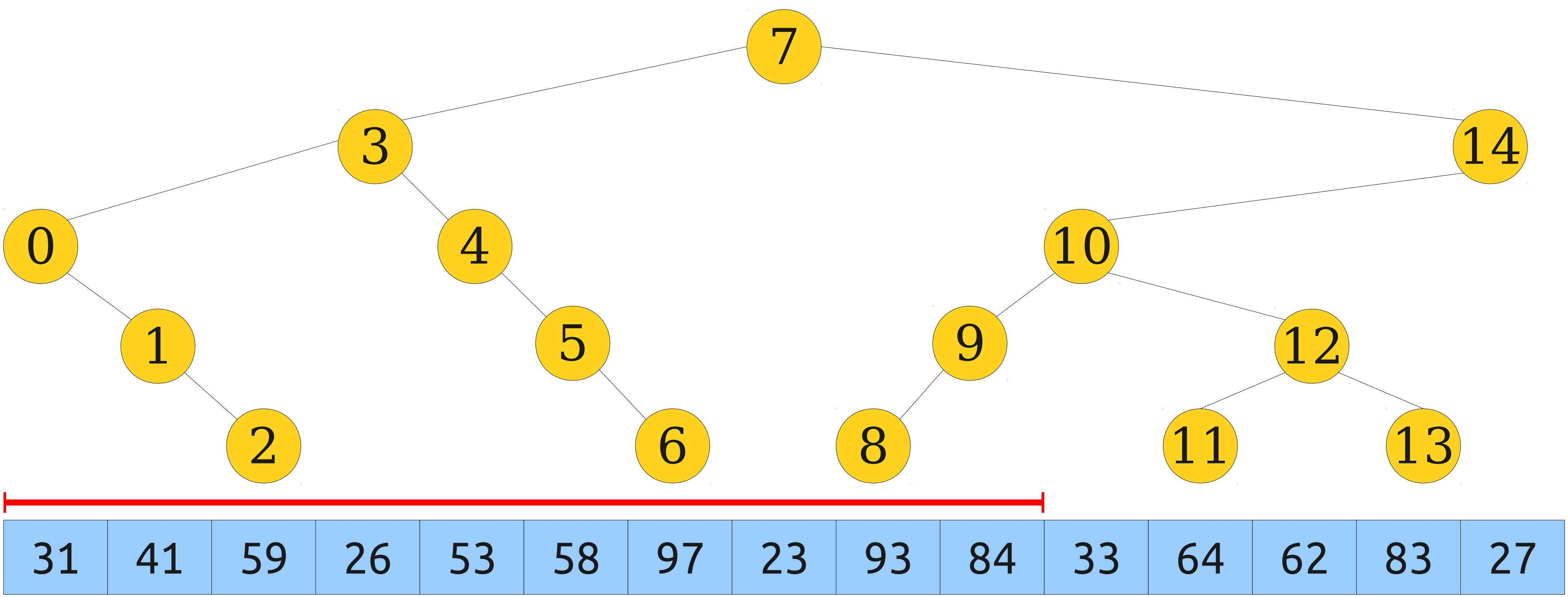
Cartesian Trees and RMQ

- **Theorem:** Let B_1 and B_2 be blocks of length b. Then $B_1 \sim B_2$ iff B_1 and B_2 have equal Cartesian trees.
- **Proof sketch:**
  - (⇒) Induction. B_1 and B_2 have equal RMQs, so corresponding ranges have the same minima.



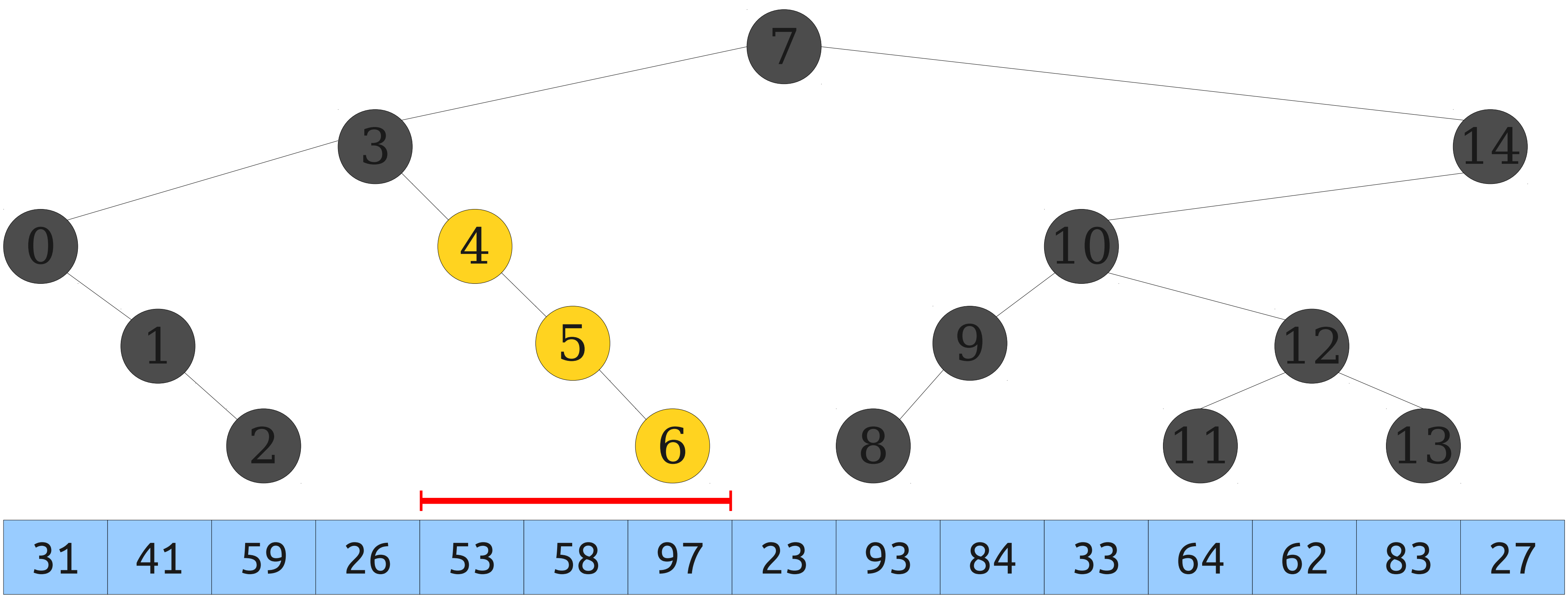
Cartesian Trees and RMQ

- **Theorem:** Let B_1 and B_2 be blocks of length b. Then $B_1 \sim B_2$ iff B_1 and B_2 have equal Cartesian trees.
- **Proof sketch:**
  - (⇐) Induction. It's possible to answer RMQ using a recursive walk on the Cartesian tree.



Cartesian Trees and RMQ

- **Theorem:** Let B_1 and B_2 be blocks of length b. Then $B_1 \sim B_2$ iff B_1 and B_2 have equal Cartesian trees.
- **Proof sketch:**
  - (⇐) Induction. It's possible to answer RMQ using a recursive walk on the Cartesian tree.



Time-Out for Announcements!

Problem Set One

- **Problem Set One** goes out today. It's due next Wednesday at 2:15PM at the start of class.
- You can work individually or in pairs.
  - If you work in pairs, submit a single problem set with both your names on it. You'll each earn the same score.
  - If you work individually, we will grade the problem set out of 19 points. (There are 23 possible points on the problem set).
- **Please read the handout on the Honor Code and the Problem Set Policies before starting this problem set.**

Office Hours

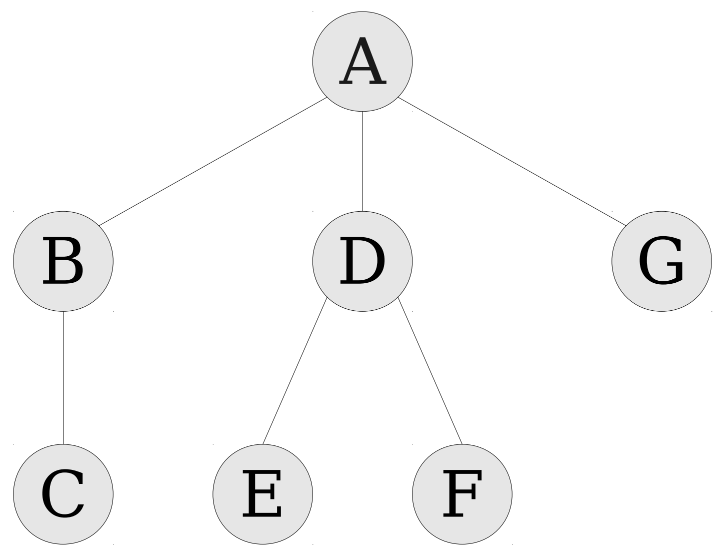
- Office hours schedule:
  - TAs: Thursday, 2PM – 4PM, location TBA.
  - Keith: Monday, 3:30PM – 5:30PM, location TBA.
- Office hours start this week. We'll email out locations and post them on the course website as well.

Your Questions

“If you were a data structure, what would you be and why?”

“What real world problems can we solve with RMQ?”

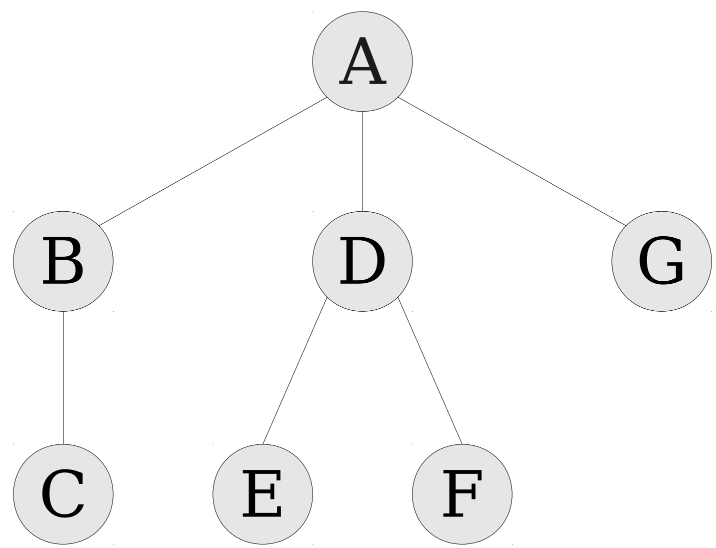
Lowest Common Ancestors



A	B	C	B	A	D	E	D	F	D	A	G	A

This is called an **Euler tour** of the tree. We'll talk about this more in a few weeks.

Lowest Common Ancestors



A	B	C	B	A	D	E	D	F	D	A	G	A
0	1	2	1	0	1	2	1	2	1	0	1	0

“Will there be a Piazza forum for this class?”

“Can Keith post slides before lecture starts (so we can print them to take notes on)?”

“Will slides be posted online after lecture?”

Back to CS166!

Building Cartesian Trees

- The previous theorem lets us check whether $B_1 \sim B_2$ by testing whether they have the same Cartesian tree.
- How efficiently can we actually build these trees?

Building Cartesian Trees

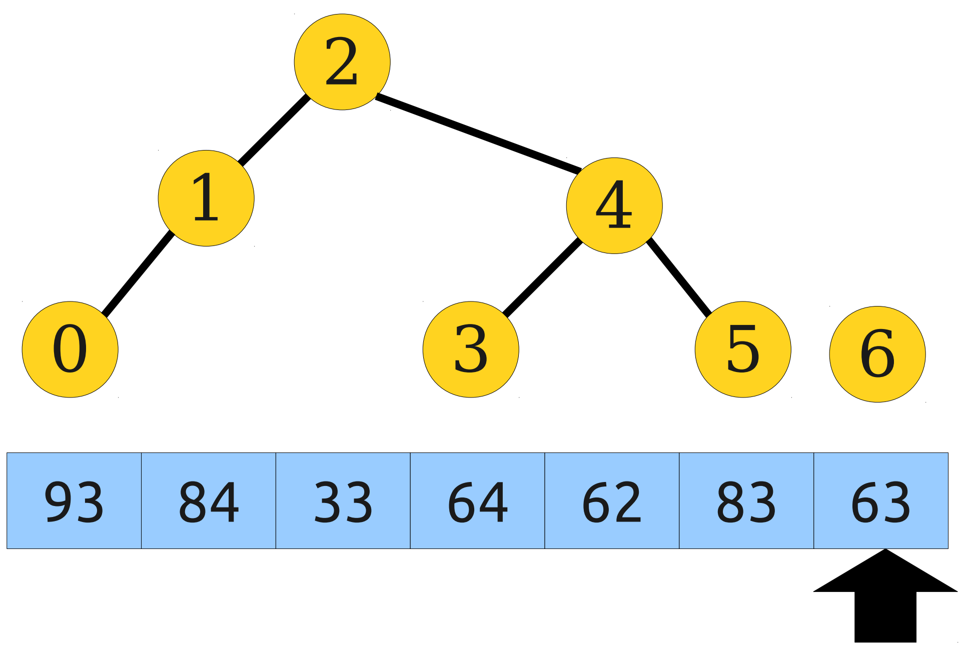
- Here's a naïve algorithm for constructing Cartesian trees:
  - Find the minimum value.
  - Recursively build a Cartesian tree with the elements to the left.
  - Recursively build a Cartesian tree with the elements to the right.
  - Return the overall tree.
- What's the runtime of this operation?

Building Cartesian Trees

- This algorithm works by
  - doing a linear scan over the array,
  - identifying the minimum at whatever position it occupies, then
  - recursively processing the left and right halves on the array.
- Similar to the recursion in quicksort: it depends on where the minima are.
  - Get a good split: O(n log n).
  - Get bad splits: O(n^2).
- We're going to need to be faster than this.

A Better Approach

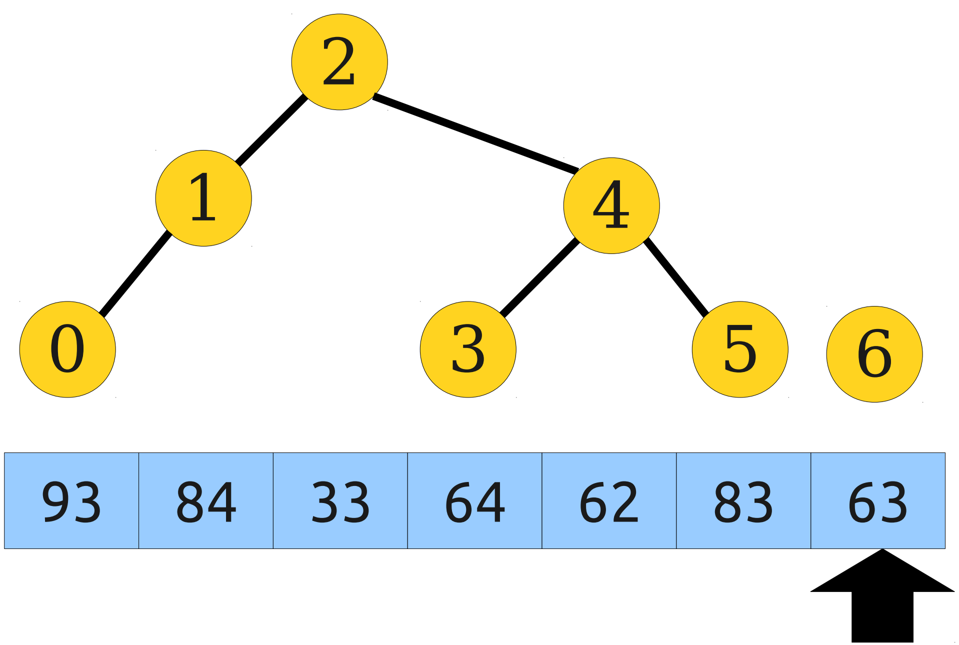
- It turns out that it's possible to build a Cartesian tree over an array of length k in time $O(k)$.
- **High-Level Idea:** Build a Cartesian tree for the first element, then the first two, then the first three, then the first four, etc.



Observation 1: This new node cannot end up as the left child of any node in the tree.

A Better Approach

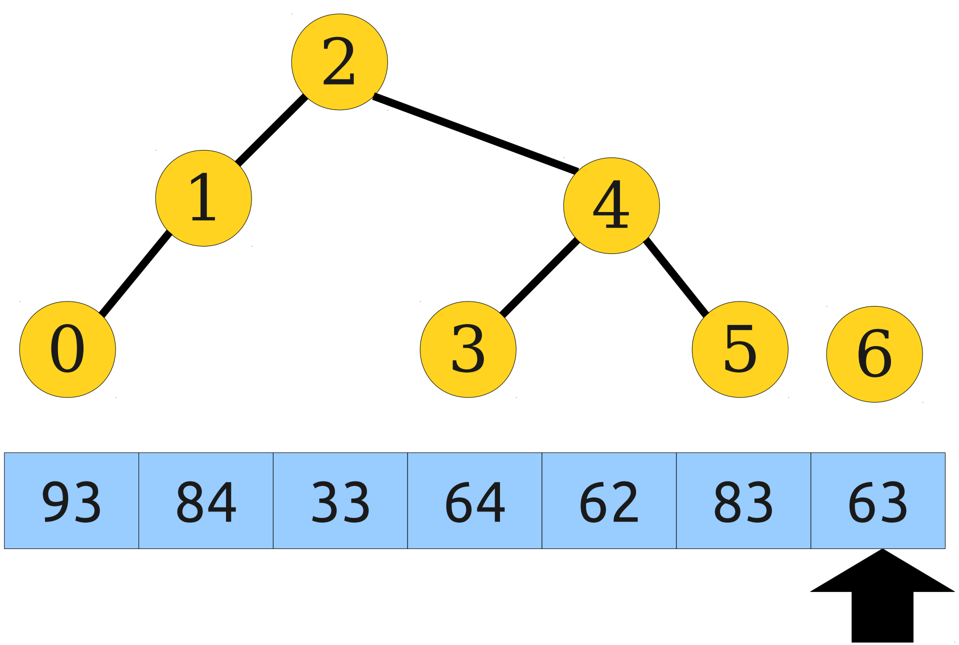
- It turns out that it's possible to build a Cartesian tree over an array of length k in time O(k).
- **High-Level Idea:** Build a Cartesian tree for the first element, then the first two, then the first three, then the first four, etc.



Observation 2: This new node will end up on the right spine of the tree.

A Better Approach

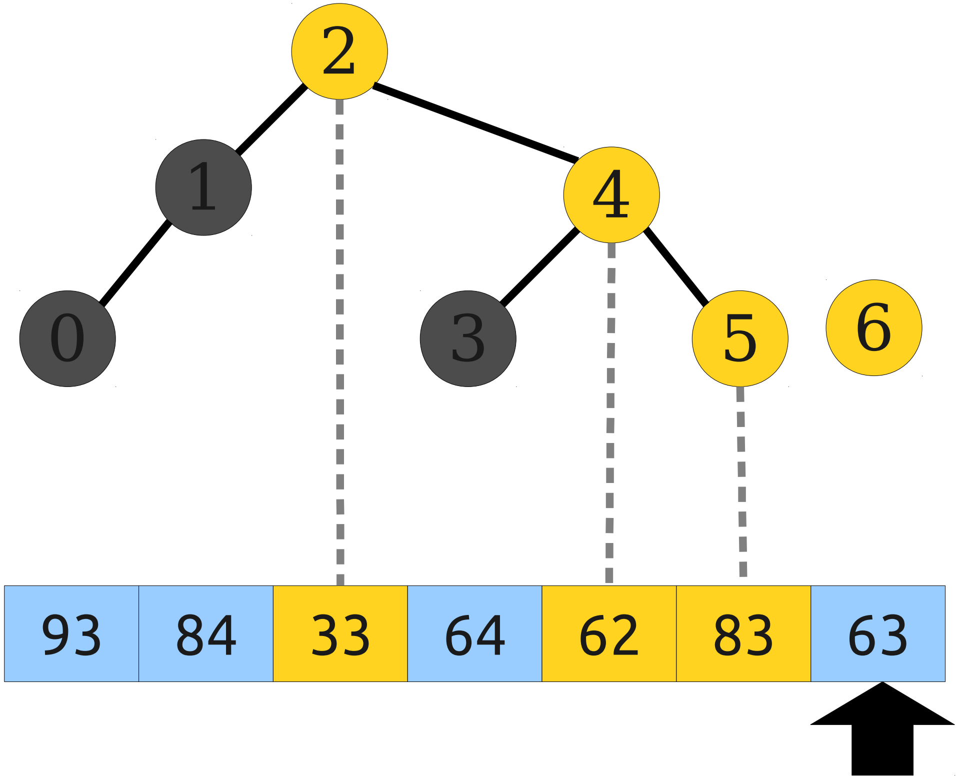
- It turns out that it's possible to build a Cartesian tree over an array of length k in time $O(k)$.
- **High-Level Idea:** Build a Cartesian tree for the first element, then the first two, then the first three, then the first four, etc.



Observation 3: Cartesian trees are min-heaps with respect to the elements in the original array.

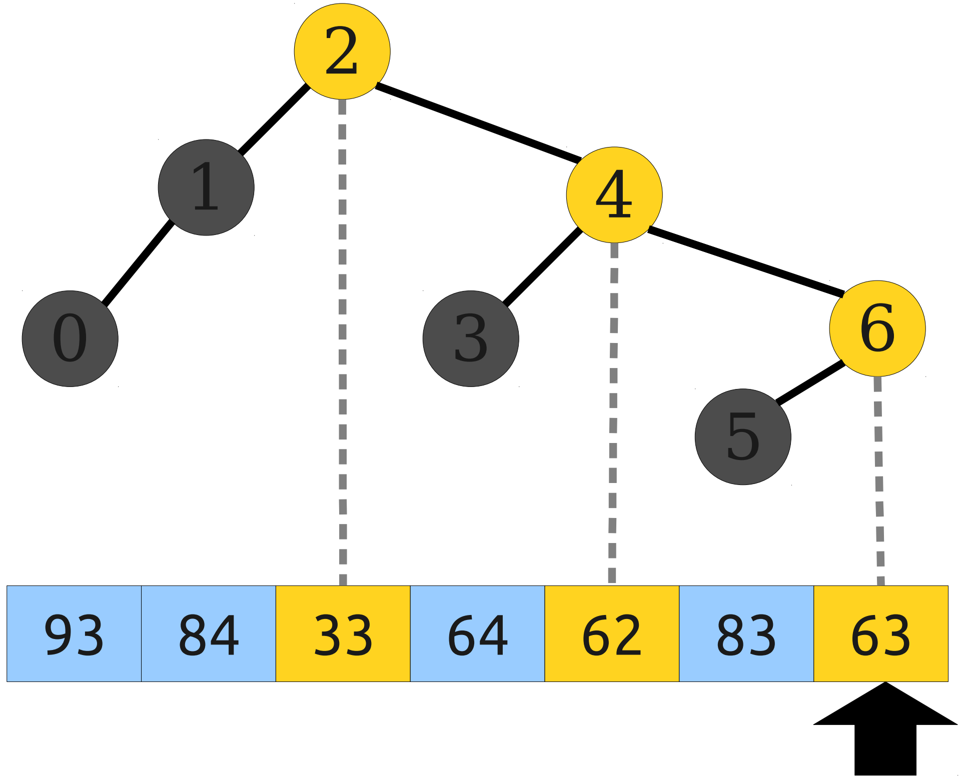
A Better Approach

- It turns out that it's possible to build a Cartesian tree over an array of length k in time O(k).
- **High-Level Idea:** Build a Cartesian tree for the first element, then the first two, then the first three, then the first four, etc.



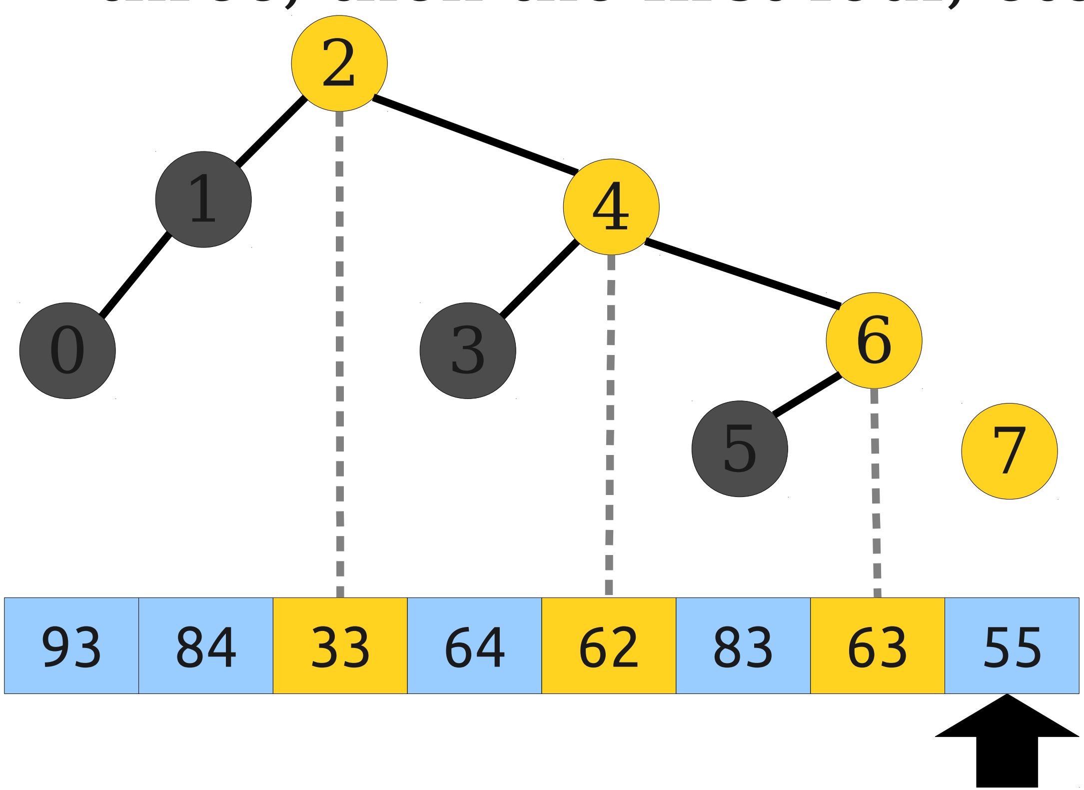
A Better Approach

- It turns out that it's possible to build a Cartesian tree over an array of length k in time O(k).
- **High-Level Idea:** Build a Cartesian tree for the first element, then the first two, then the first three, then the first four, etc.



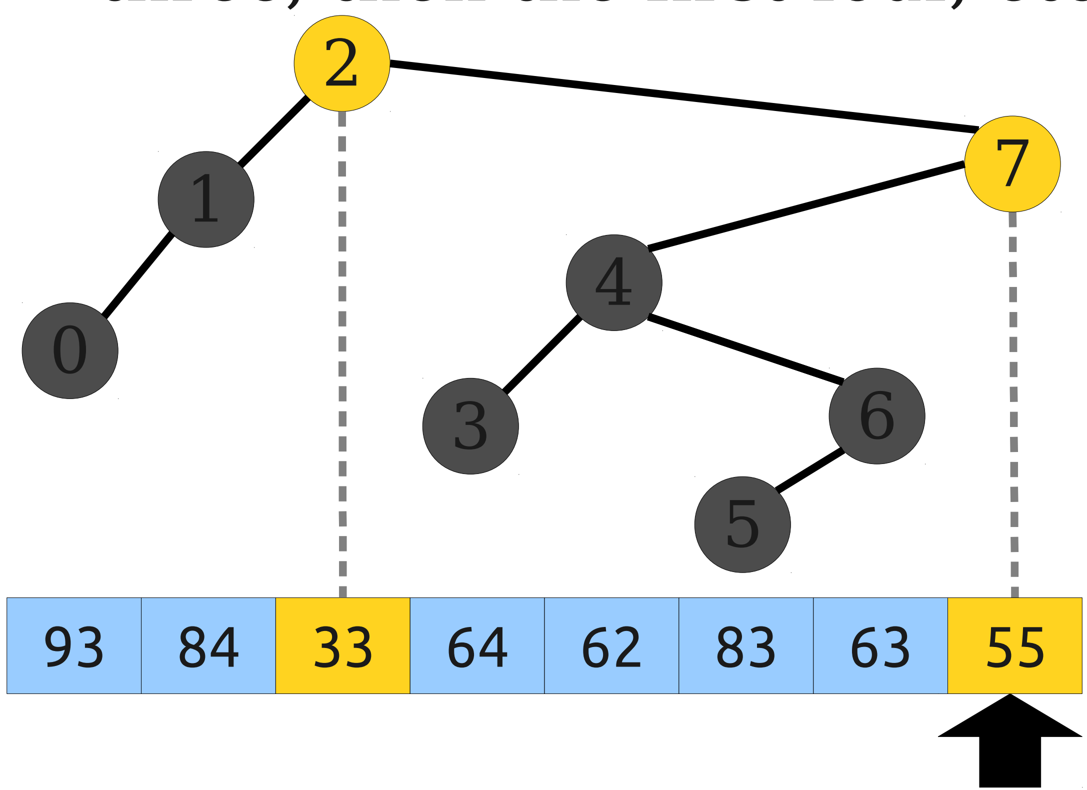
A Better Approach

- It turns out that it's possible to build a Cartesian tree over an array of length k in time $O(k)$.
- **High-Level Idea:** Build a Cartesian tree for the first element, then the first two, then the first three, then the first four, etc.



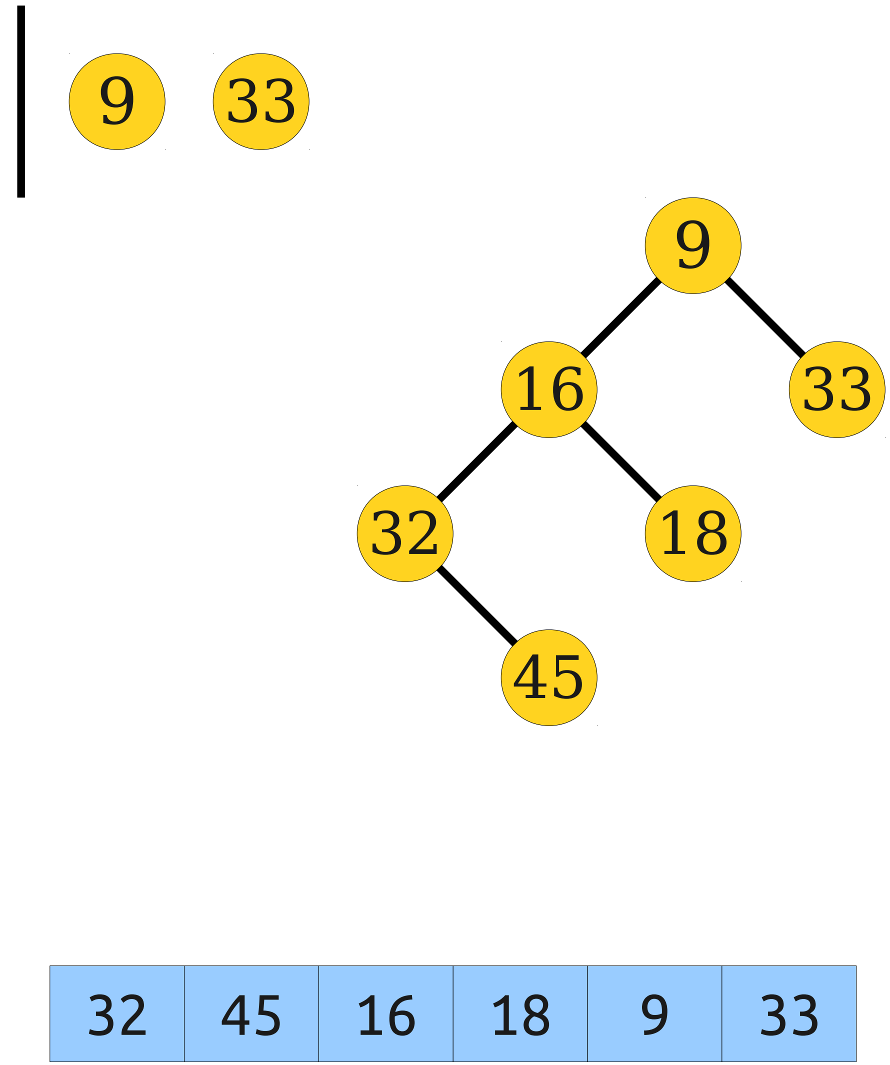
A Better Approach

- It turns out that it's possible to build a Cartesian tree over an array of length k in time $O(k)$.
- **High-Level Idea:** Build a Cartesian tree for the first element, then the first two, then the first three, then the first four, etc.



A Stack-Based Algorithm

- Maintain a stack of the nodes on the right spine of the tree.
- To insert a new node:
  - Pop the stack until it's empty or the top node has a lower value than the current value.
  - Set the new node's left child to be the last value popped (or **`null`** if nothing was popped).
  - Set the new node's parent to be the top node on the stack (or **`null`** if the stack is empty).
  - Push the new node onto the stack.



Analyzing the Runtime

- Pushing each node might take time O(n), since we might have to pop everything off the stack.
- Runtime is therefore O(n^2).
- **Claim:** Runtime is actually Θ(n).
- **Proof:** Work done per node is directly proportional to the number of stack operations performed when that node was processed.
- Total number of stack operations is at most $2n$.
  - Every node is pushed once.
  - Every node is popped at most once.
- Total runtime is therefore Θ(n).

The Story So Far

- Since we can build Cartesian trees in linear time, we can test if two blocks have the same type in linear time.
- **Goal:** Choose a block size that's small enough that there are duplicated blocks, but large enough that the top-level RMQ can be computed efficiently.
- So how many Cartesian trees are there?

Theorem: The number of Cartesian trees for an array of length b is at most 4^b.

In case you're curious, the actual number is

$$\frac{1}{b+1}\binom{2b}{b}$$

which is roughly

$$\frac{4^b}{b^{3/2}\sqrt{\pi}}$$

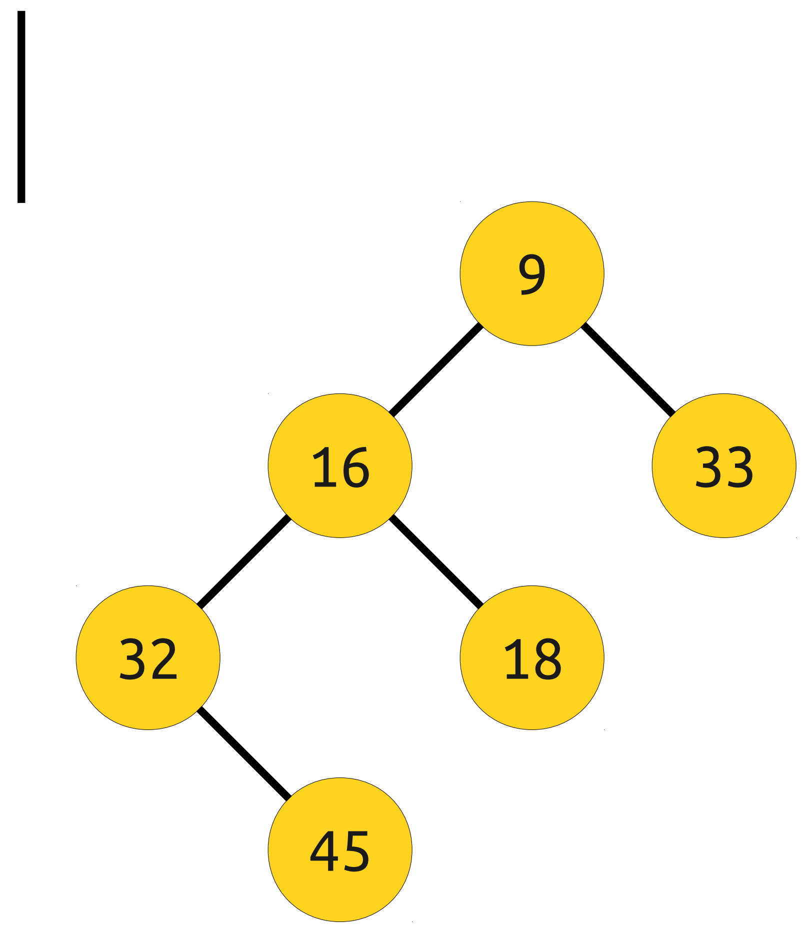
Proof Approach

- Our stack-based algorithm for generating Cartesian trees is capable of producing a Cartesian tree for every possible input array.
- Therefore, if we can count the number of possible executions of that algorithm, we can count the number of Cartesian trees.
- Using a simple counting scheme, we can show that there are at most 4^b possible executions.

The Insight

- **Claim:** The Cartesian tree produced by the stack-based algorithm is uniquely determined by the sequence of pushes and pops made on the stack.
- There are at most $2b$ stack operations during the execution of the algorithm: b pushes and no more than b pops.
- Can represent the execution as a $2b$-bit number, where 1 means “push” and 0 means “pop.” We'll pad the end with 0's (pretend we pop everything from the stack.)
  - We'll call this number the **Cartesian tree number** of a particular block.
- There are at most $2^{2b} = 4^b$ possible $2b$-bit numbers, so there are at most 4^b possible Cartesian trees.

Cartesian Tree Numbers



32	45	16	18	9	33

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

We don't actually need to build the Cartesian tree – we can just simulate the stack!

Cartesian Tree Numbers

27	18	28	18	28	45	90	45	23	53	60	28	74	71	35

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

Finishing Things Up

- Using the previous algorithm, we can compute the Cartesian tree number of a block in time O(b) and without actually building the tree.
- Gives a simple and efficient linear-time algorithm for testing whether two blocks have the same block type.
- And, we bounded the number of Cartesian trees at 4^b using this setup!

The Fischer-Heun Structure

- In 2005, Fischer and Heun introduced a (slight variation on) the following RMQ data structure.
- Use a hybrid approach with block size b (we'll choose b later), a sparse table as a top RMQ structure, and the full precomputation data structure for the blocks.
- However, make the following modifications:
  - Make a table of length 4^b holding RMQ structures. The index corresponds to the Cartesian tree number. Initially, the array is empty.
  - When computing the RMQ for a particular block, first compute its Cartesian tree number t.
  - If there's an RMQ structure for t in the array, use it.
  - Otherwise, compute the RMQ structure for the current block, store it in the array and index t, then use it.

Analyzing the Runtime

- What is the query time on the Fischer-Heun structure?
  - O(1) queries in top-level structure.
  - O(1) queries in each block.
- Total query time: **O(1)**.

Analyzing the Runtime

- Splitting the input into blocks and computing the block mins takes time $O(n)$ regardless of b.
- Creating the sparse table takes time $O((n / b) \log n)$.
- Computing the Cartesian tree number of each block takes time $O(n)$ in total.
  - $O(b)$ work $O(n / b)$ times.
- Maximum possible work constructing RMQ structures is $O(4^b b^2)$:
  - Takes time $O(b^2)$ to compute an RMQ structure.
  - Done at most 4^b times, one per possible Cartesian tree.
- Total runtime:

$$\mathbf{O(n + (n / b) \log n + 4^b b^2)}$$

The Finishing Touch

- The runtime is

$$\mathbf{O(n + (n / b) \log n + 4^b b^2)}$$

- As we saw earlier, if we set $b = \Theta(\log n)$, then

$$(n / b) \log n = O(n)$$

- Suppose we set $b = \log_4 (n^{1/2}) = ¼\log_2 n$. Then

$$4^b b^2 = n^{1/2} (\log_2 n)^2 = o(n)$$

- With $b = ¼\log_2 n$, the preprocessing time is

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

- **We finally have an ⟨O(*n*), O(1)⟩ RMQ solution!**

Practical Concerns

- This structure is actually reasonably efficient; preprocessing is relatively fast.
- In practice, the ⟨O(n), O(log n)⟩ hybrid is a bit faster:
  - Constant factor in the Fischer-Heun O(n) and O(1) are high.
  - Constant factor in the hybrid approach's O(n) and O(log n) are very low.
- Check the Fischer-Heun paper for details.

Wait a Minute...

- This approach assumes that the Cartesian tree numbers will fit into individual machine words!
- If $b = ¼ \log_2 n$, then each Cartesian tree number will have $½ \log_2 n$ bits.
- Cartesian tree numbers will fit into a machine word if n fits into a machine word.
- In the **transdichotomous machine model**, we assume the problem size always fits into a machine word.
  - Reasonable – think about how real computers work.
- So there's nothing to worry about.

The Method of Four Russians

- The technique employed here is an example of the **Method of Four Russians**.
- Idea:
  - Split the input apart into blocks of size $\Theta(\log n)$.
  - Using the fact that there can only be polynomially many different blocks of size $\Theta(\log n)$, evaluate the blocks more efficiently than evaluating each one independently.
  - Combine the results together using a top-level structure on an input of size $\Theta(n / \log n)$.
- This technique is used frequently to shave log factors off of runtimes.

Why Study RMQ?

- I chose RMQ as our first problem for a few reasons:
  - **See different approaches to the same problem**. Different intuitions produced different runtimes.
  - **Build data structures out of other data structures**. Many modern data structures use other data structures as building blocks, and it's very evident here.
  - **See the Method of Four Russians**. This trick looks like magic the first few times you see it and shows up in lots of places.
  - **Explore modern data structures**. This is relatively recent data structure (2005), and I wanted to show you that the field is still very active!
- So what's next?

Next Time

- **Balanced Trees**
  - One of the most versatile and useful data structures around.
- **B-Trees**
  - Data structures for storing sorted information on disk.
- **Red/Black Trees**
  - They're not as scary as they might look. Trust me!