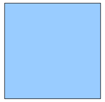
A Faster Approach: **Segment Trees**

A Divide-and-Conquer Approach

- **Idea:** Use divide-and-conquer to speed up RMQ calculations by splitting the overall array in half at each point.

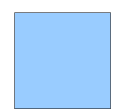
A Divide-and-Conquer Approach

- **Idea:** Use divide-and-conquer to speed up RMQ calculations by splitting the overall array in half at each point.
- **Base case:**



A Divide-and-Conquer Approach

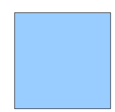
- **Idea:** Use divide-and-conquer to speed up RMQ calculations by splitting the overall array in half at each point.
- **Base case:**



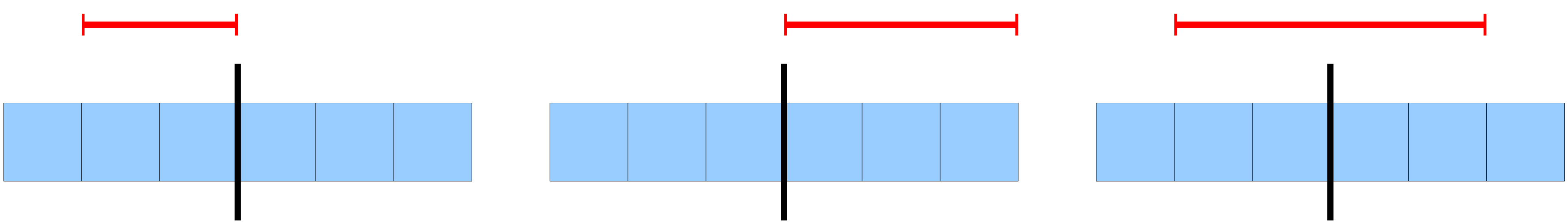
- **Recursive cases:**

A Divide-and-Conquer Approach

- **Idea:** Use divide-and-conquer to speed up RMQ calculations by splitting the overall array in half at each point.
- **Base case:**

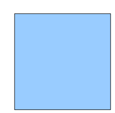


- **Recursive cases:**

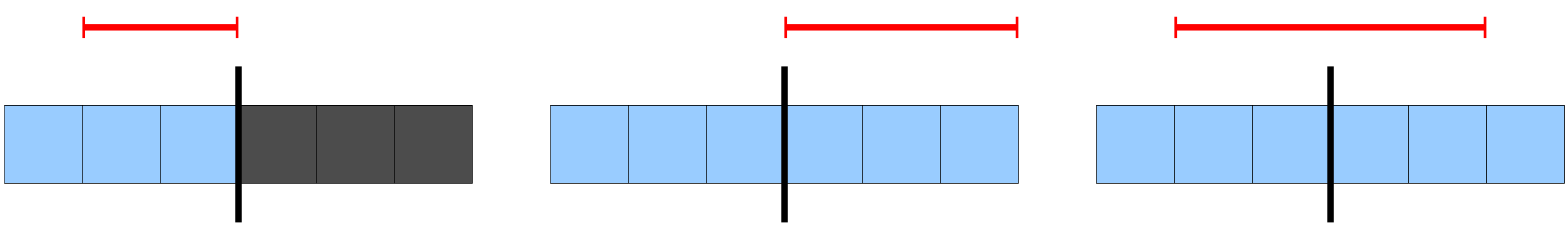


A Divide-and-Conquer Approach

- **Idea:** Use divide-and-conquer to speed up RMQ calculations by splitting the overall array in half at each point.
- **Base case:**

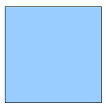


- **Recursive cases:**

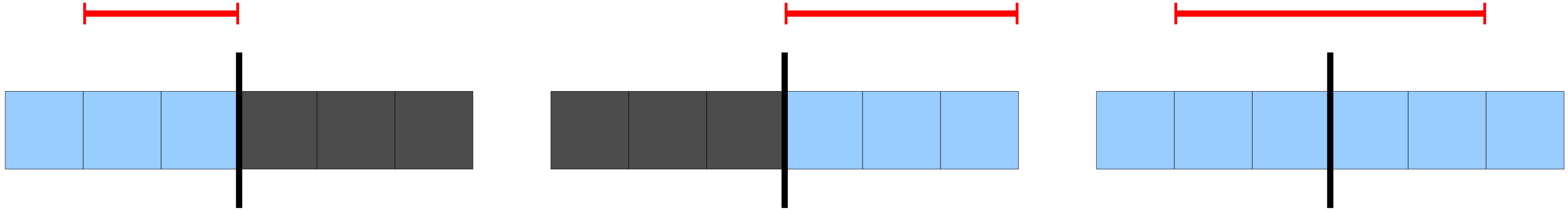


A Divide-and-Conquer Approach

- **Idea:** Use divide-and-conquer to speed up RMQ calculations by splitting the overall array in half at each point.
- **Base case:**

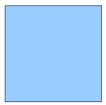


- **Recursive cases:**

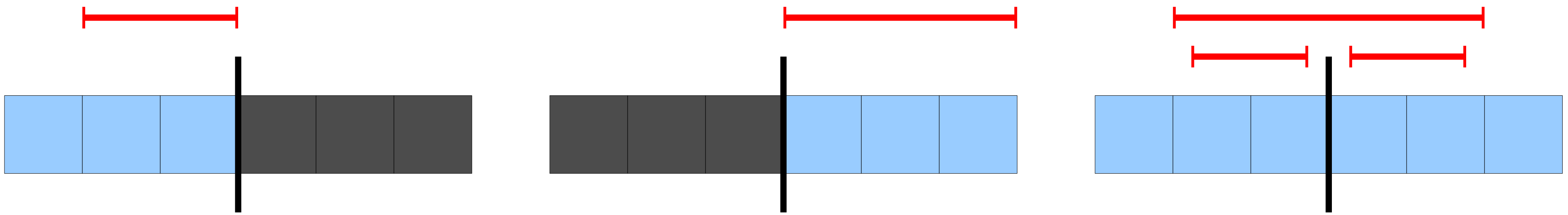


A Divide-and-Conquer Approach

- **Idea:** Use divide-and-conquer to speed up RMQ calculations by splitting the overall array in half at each point.
- **Base case:**



- **Recursive cases:**



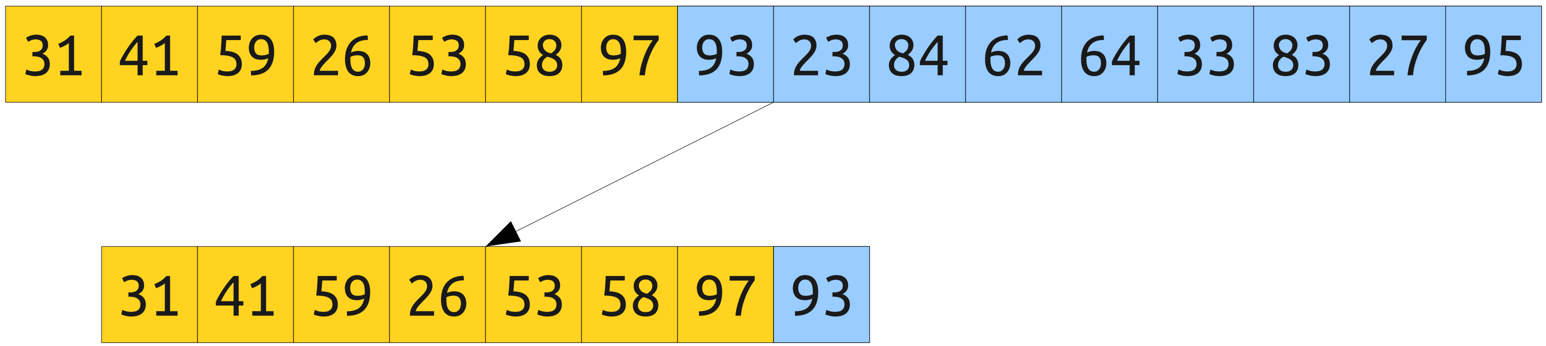
A Divide-and-Conquer Approach

31	41	59	26	53	58	97	93	23	84	62	64	33	83	27	95

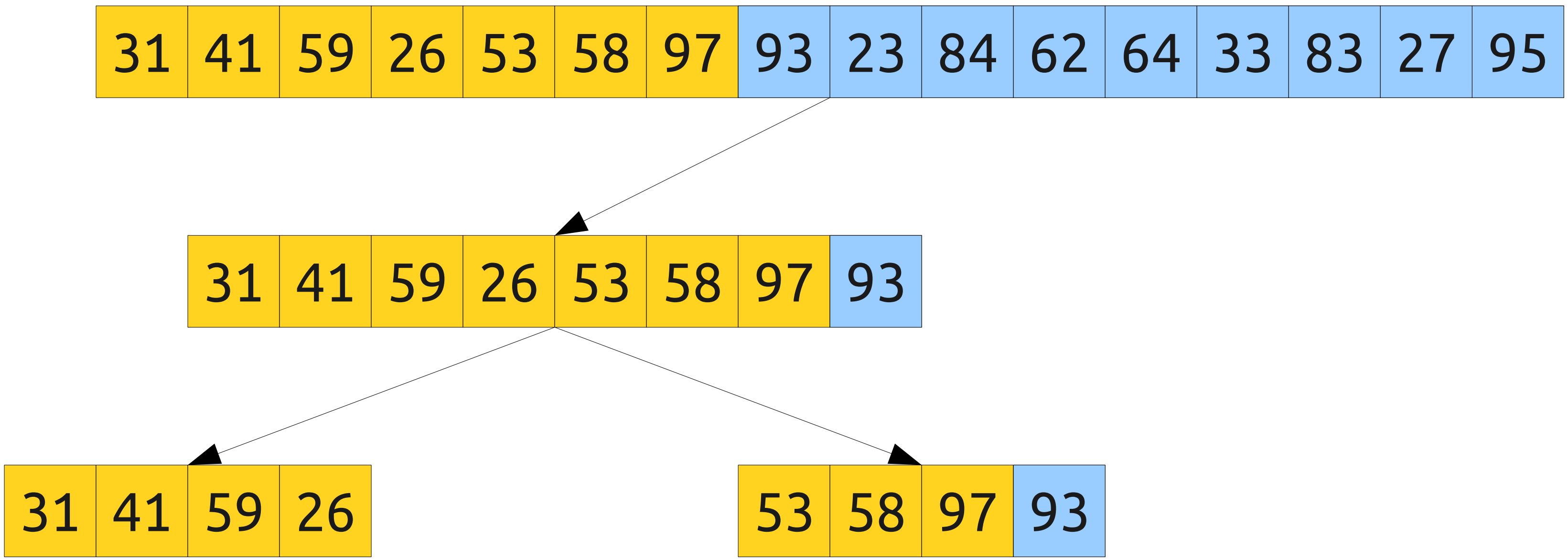
A Divide-and-Conquer Approach

31	41	59	26	53	58	97	93	23	84	62	64	33	83	27	95

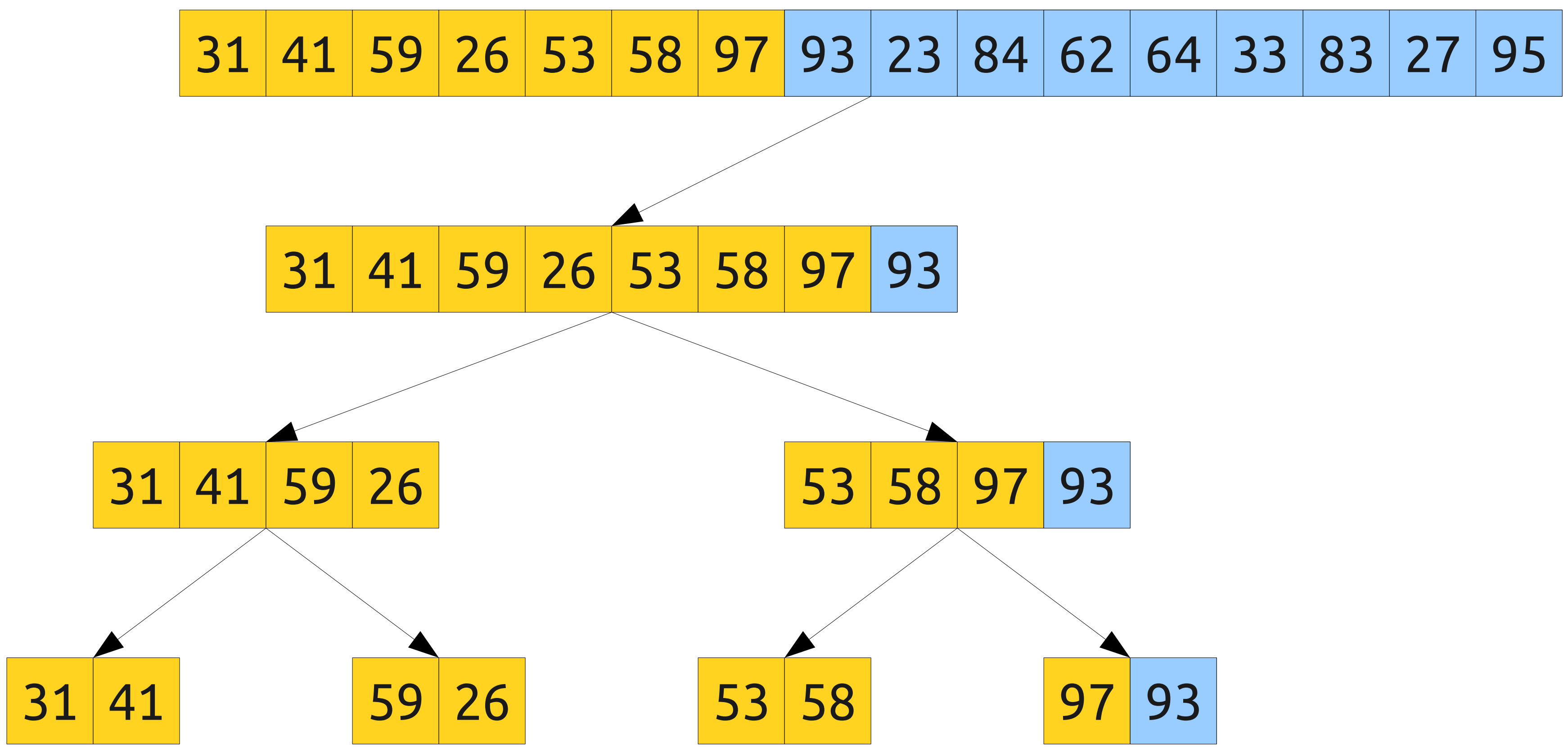
A Divide-and-Conquer Approach



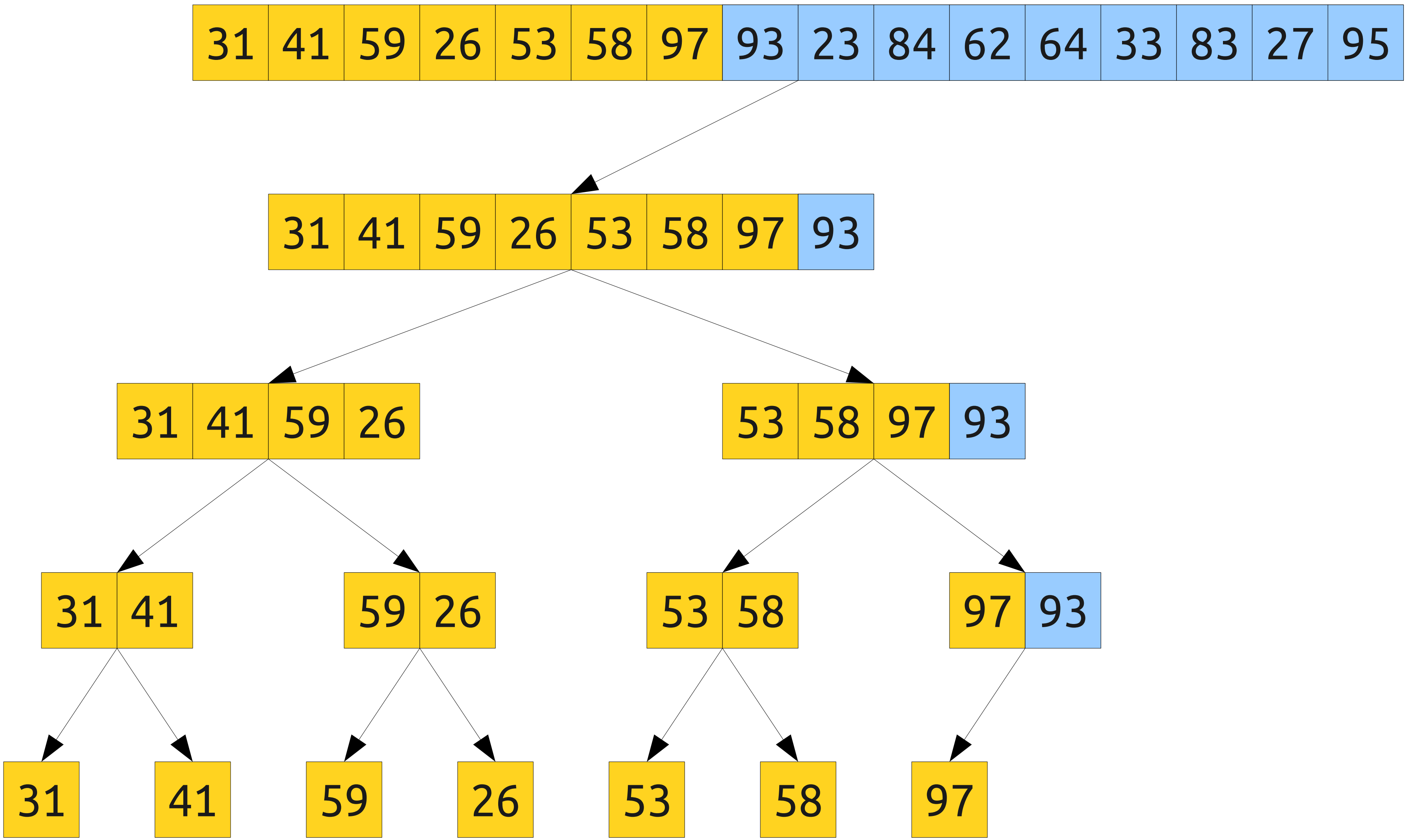
A Divide-and-Conquer Approach



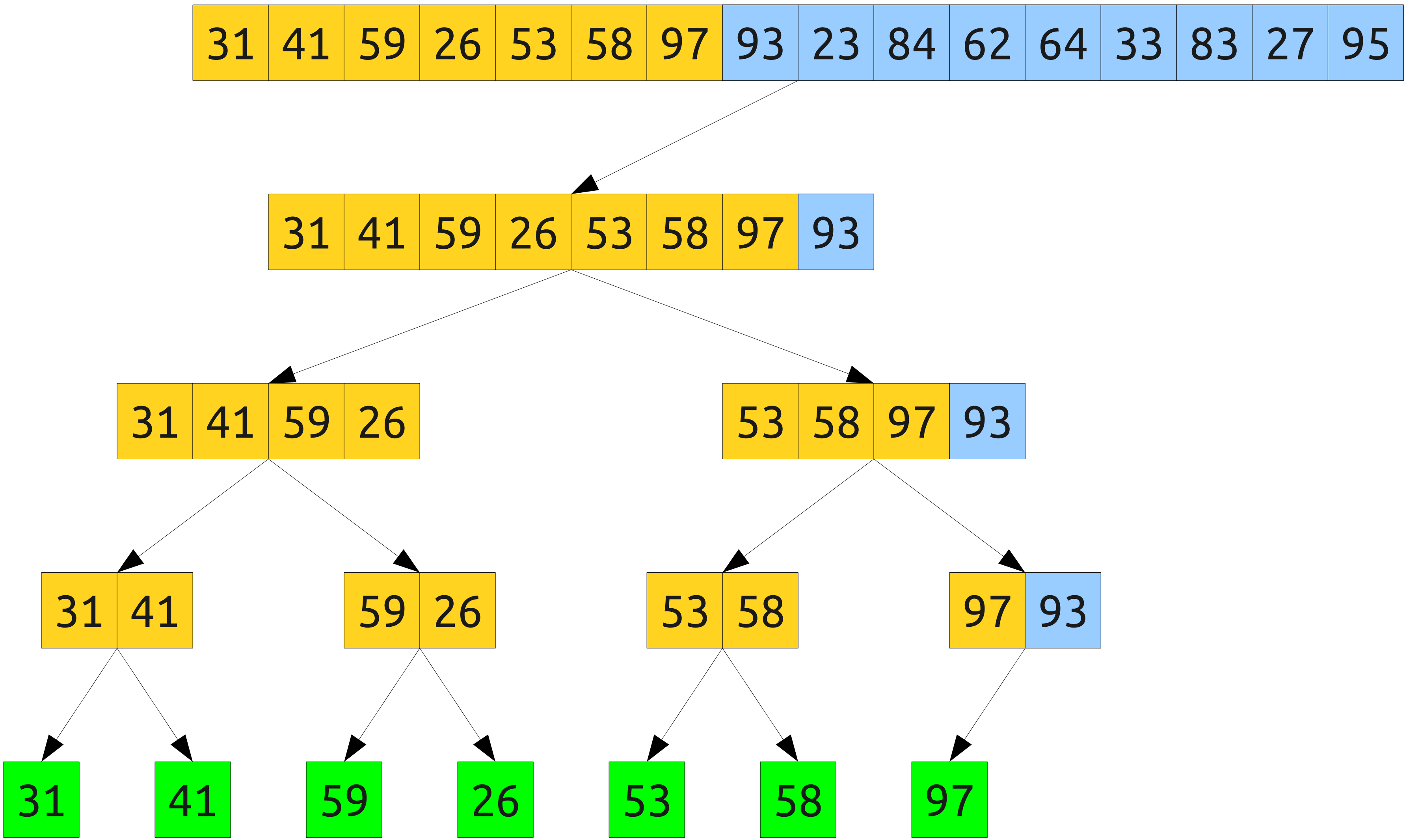
A Divide-and-Conquer Approach



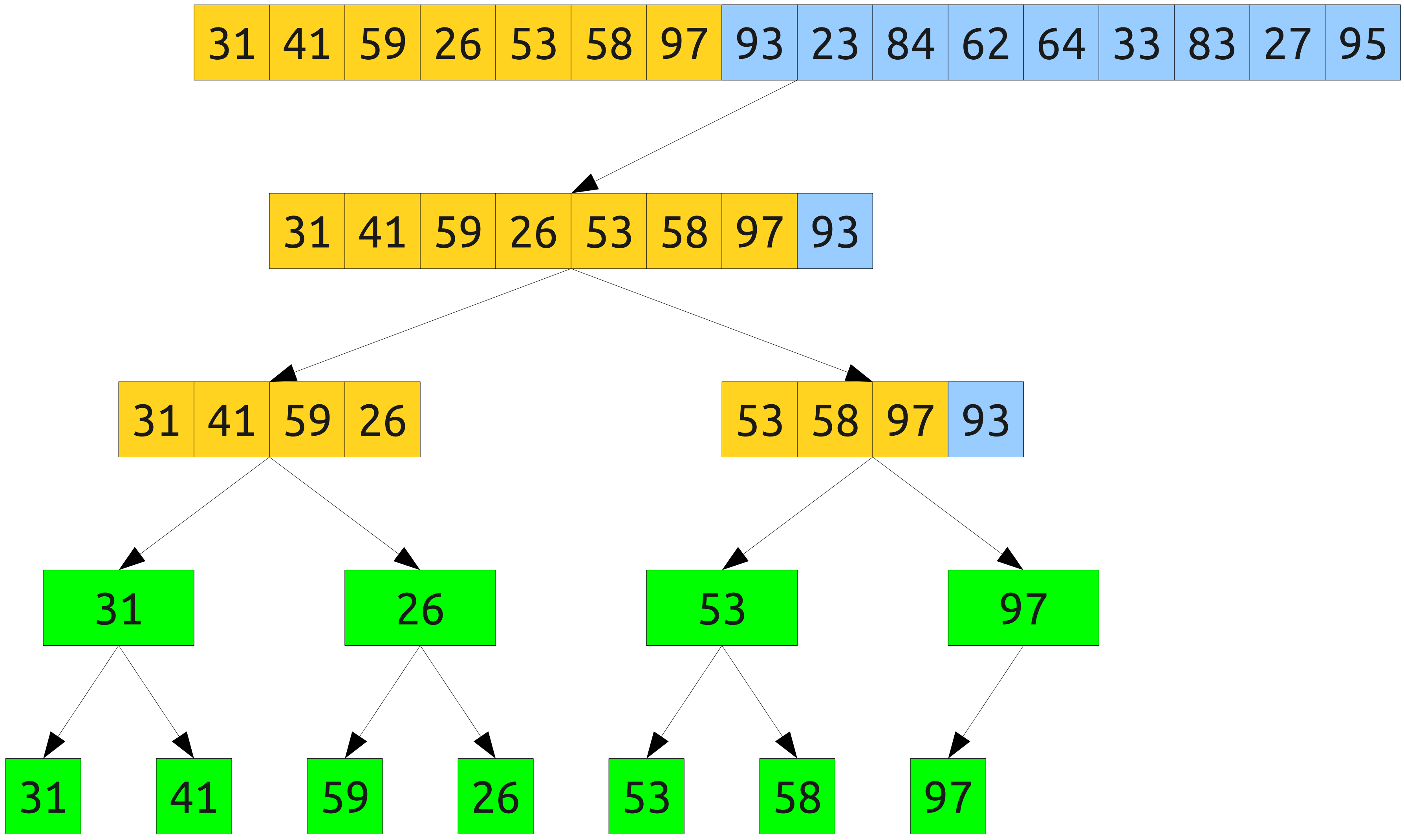
A Divide-and-Conquer Approach



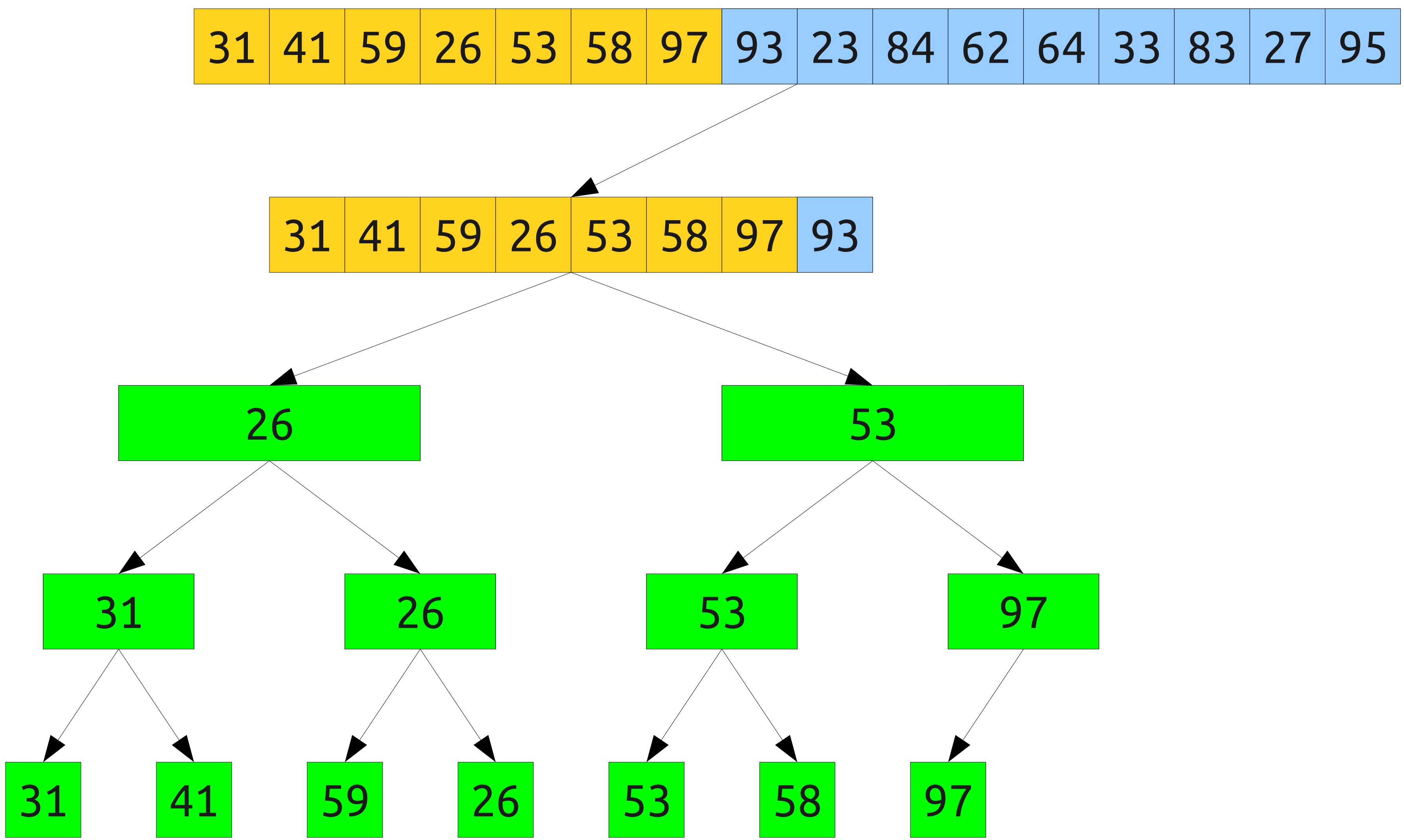
A Divide-and-Conquer Approach



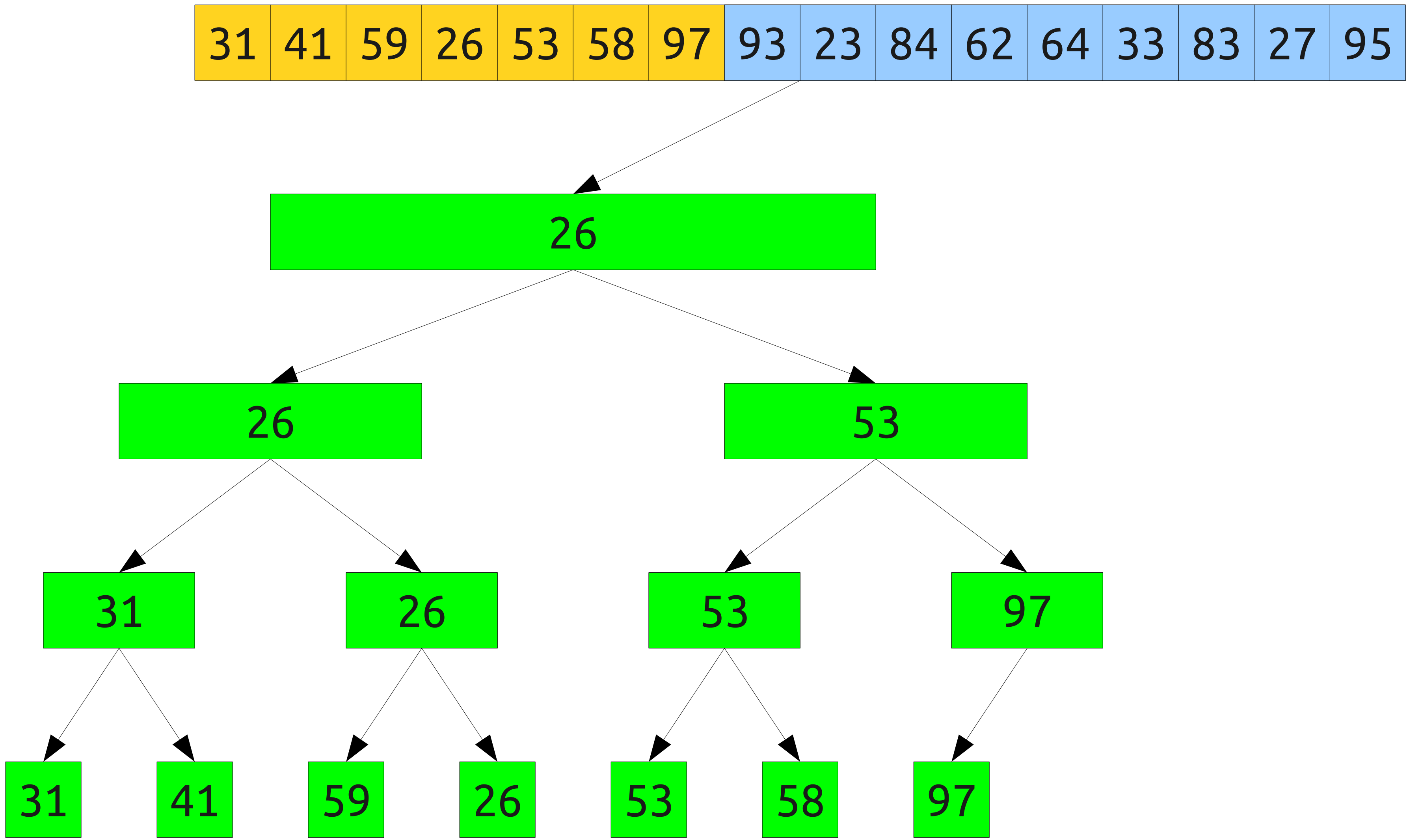
A Divide-and-Conquer Approach



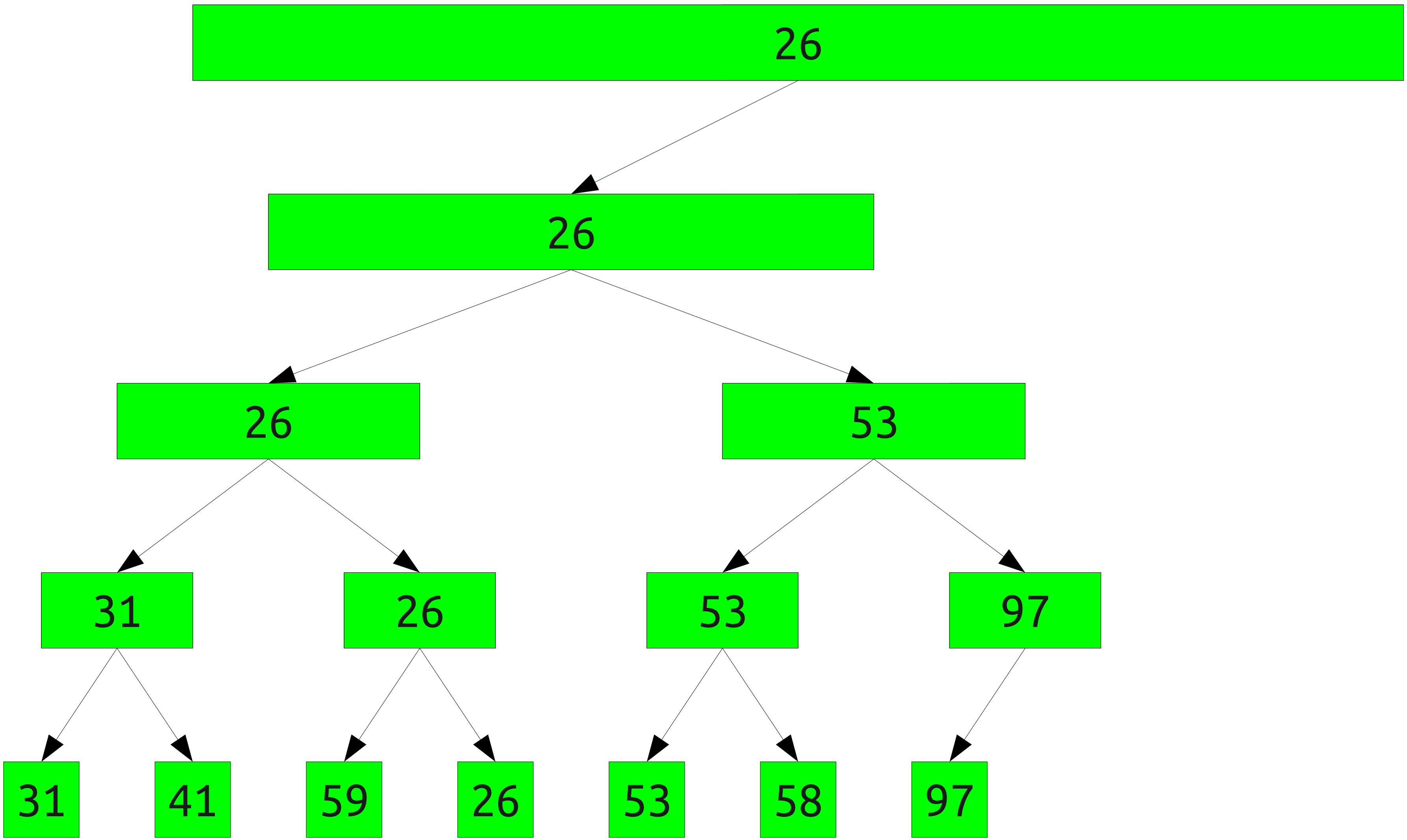
A Divide-and-Conquer Approach



A Divide-and-Conquer Approach



A Divide-and-Conquer Approach



Analyzing Efficiency

- Each recursive call fires off at most two recursive calls and does O(1) work to combine them.
- Recurrence relation:

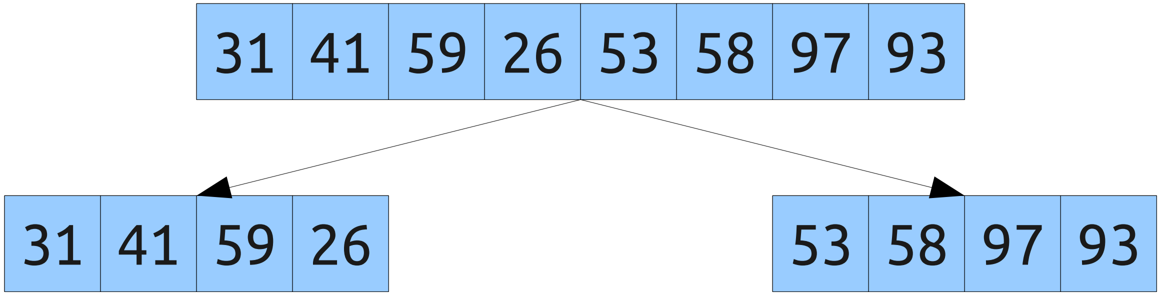
$$T(n) = 2T(n / 2) + O(1)$$

- Using the Master Theorem, this solves to **O(*n*)**.
- This is no better than our initial solution!

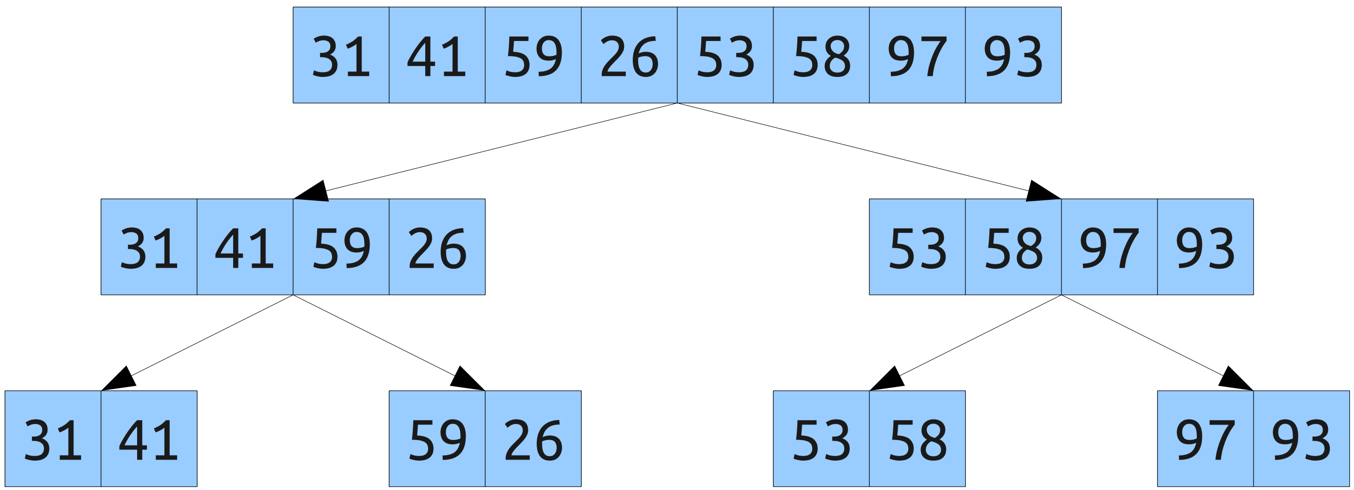
An Observation

31	41	59	26	53	58	97	93

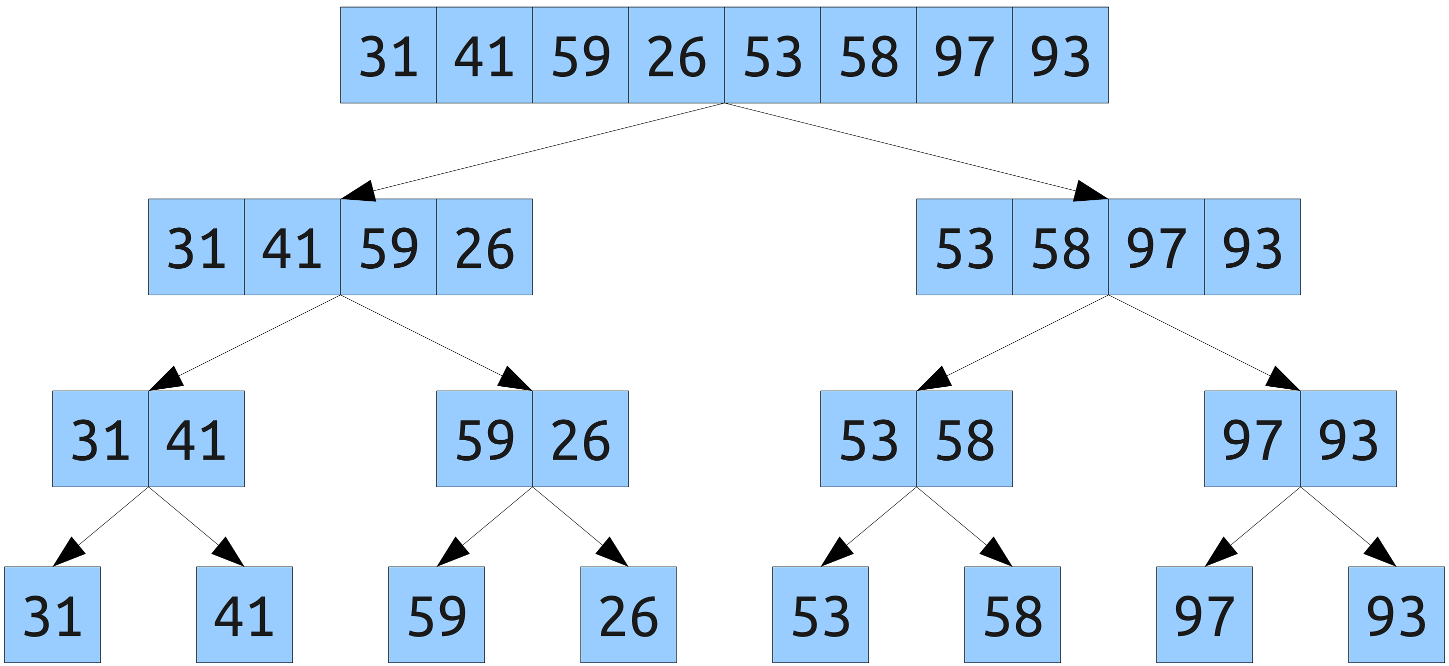
An Observation



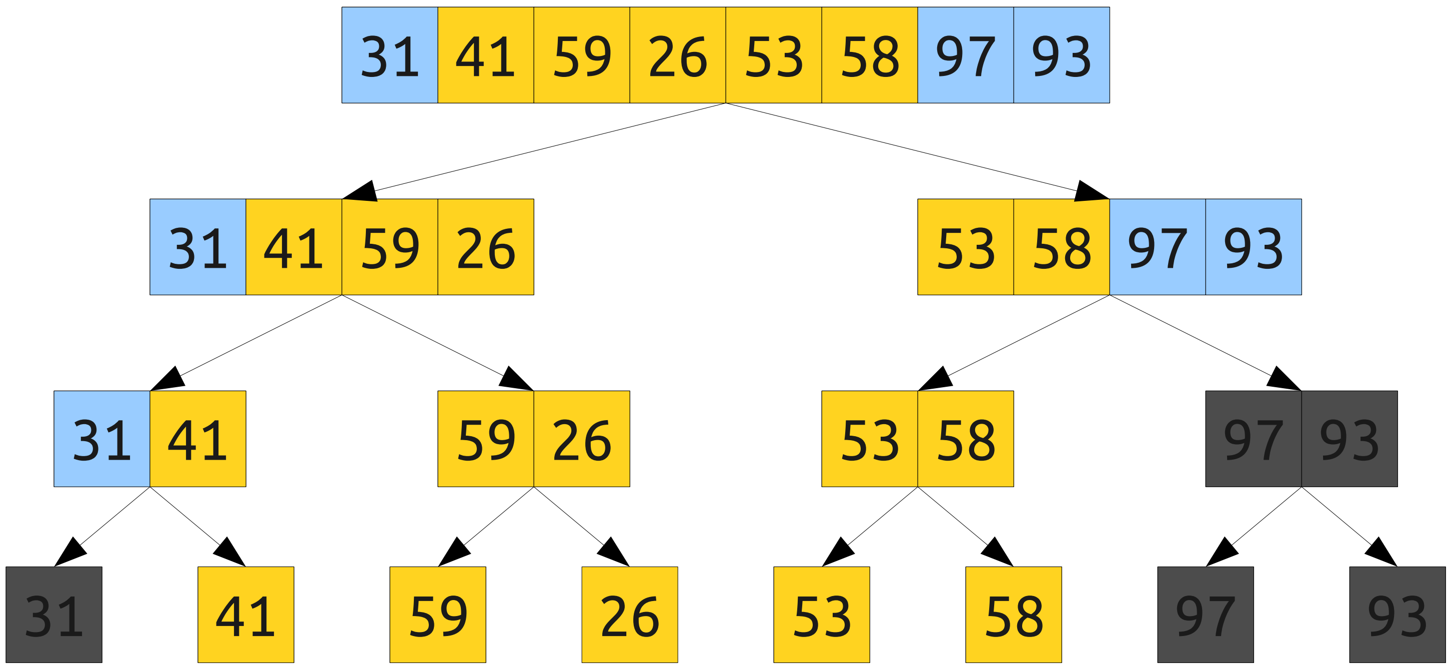
An Observation



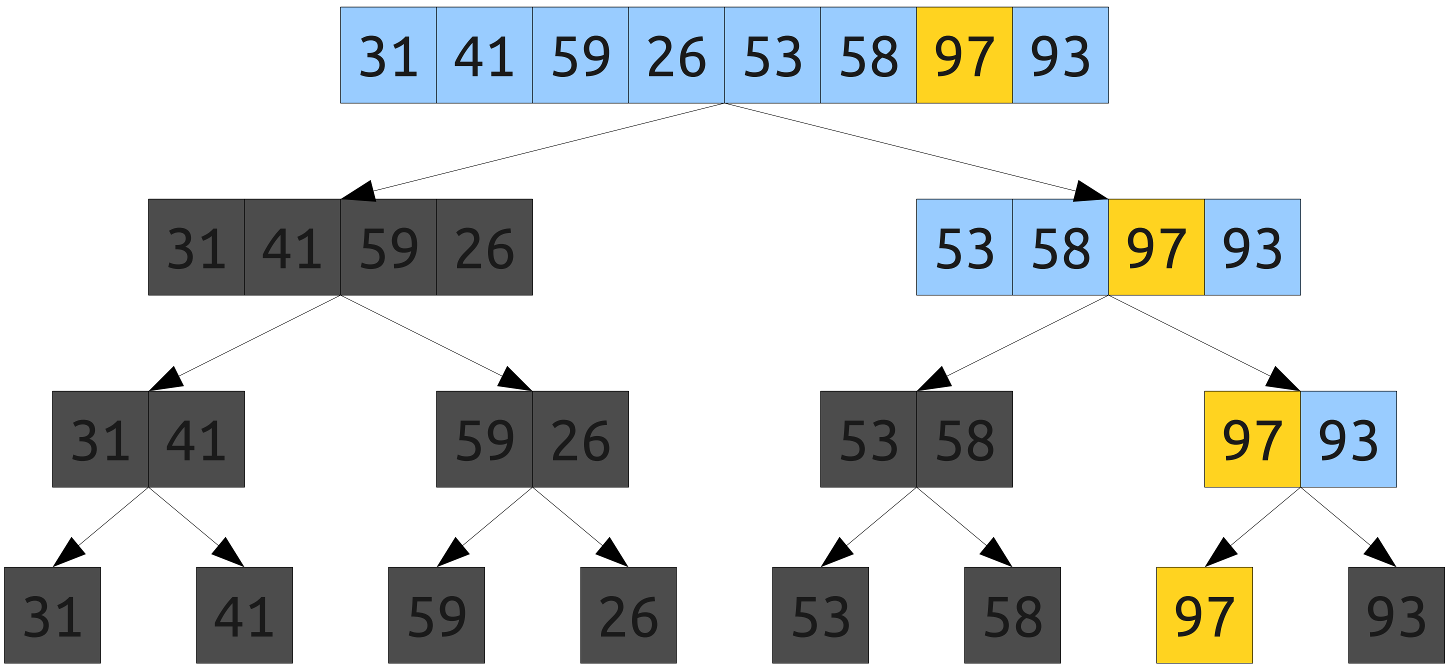
An Observation



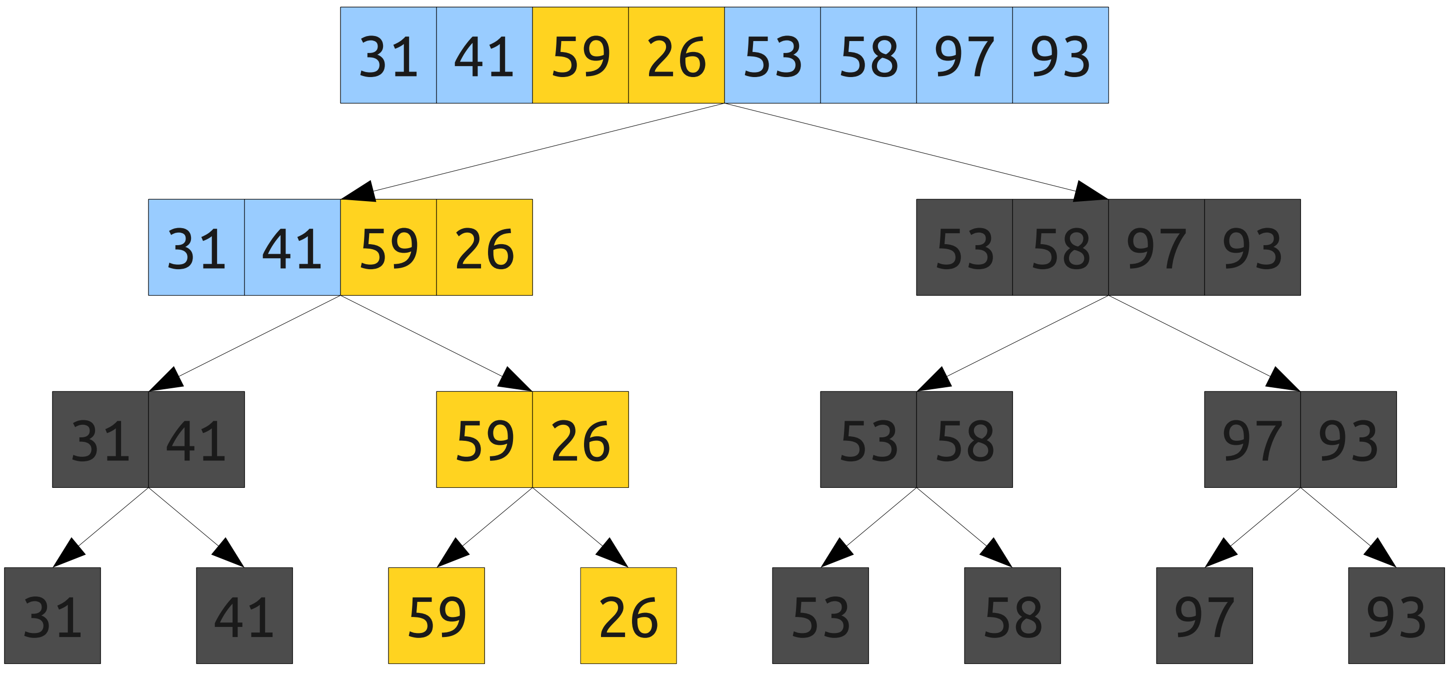
An Observation



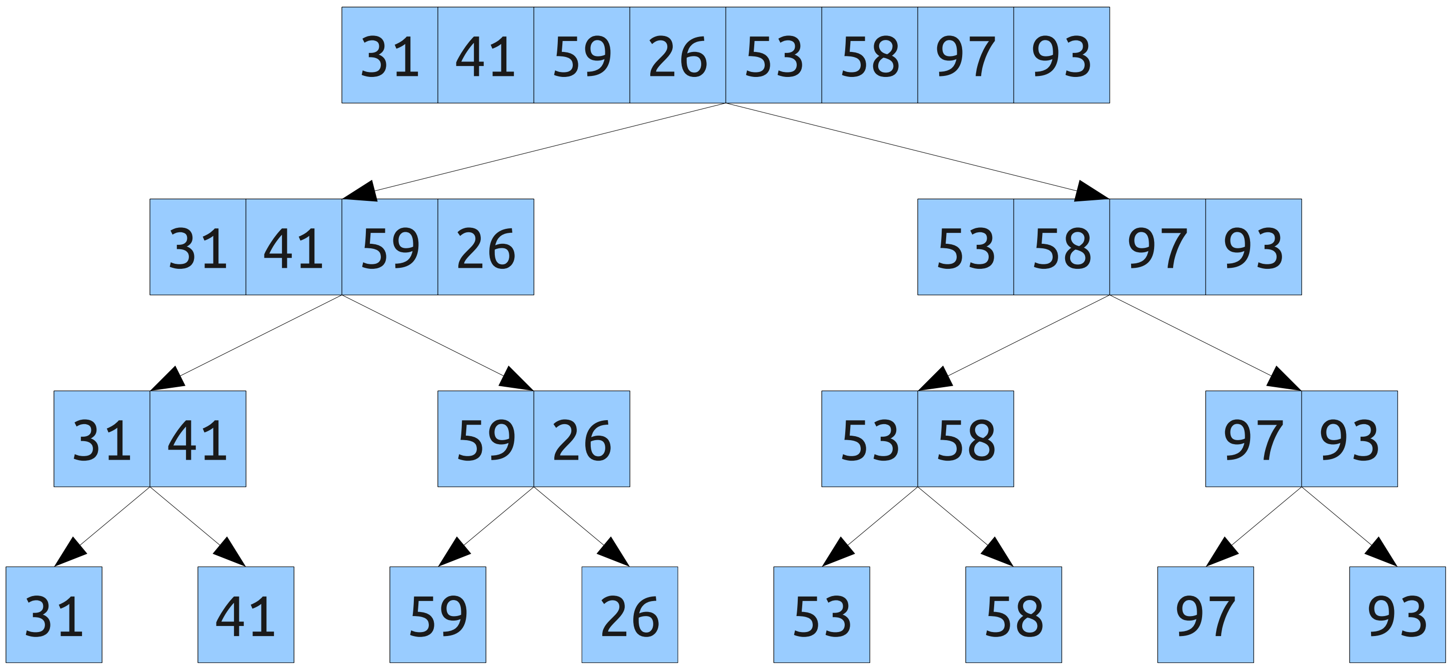
An Observation



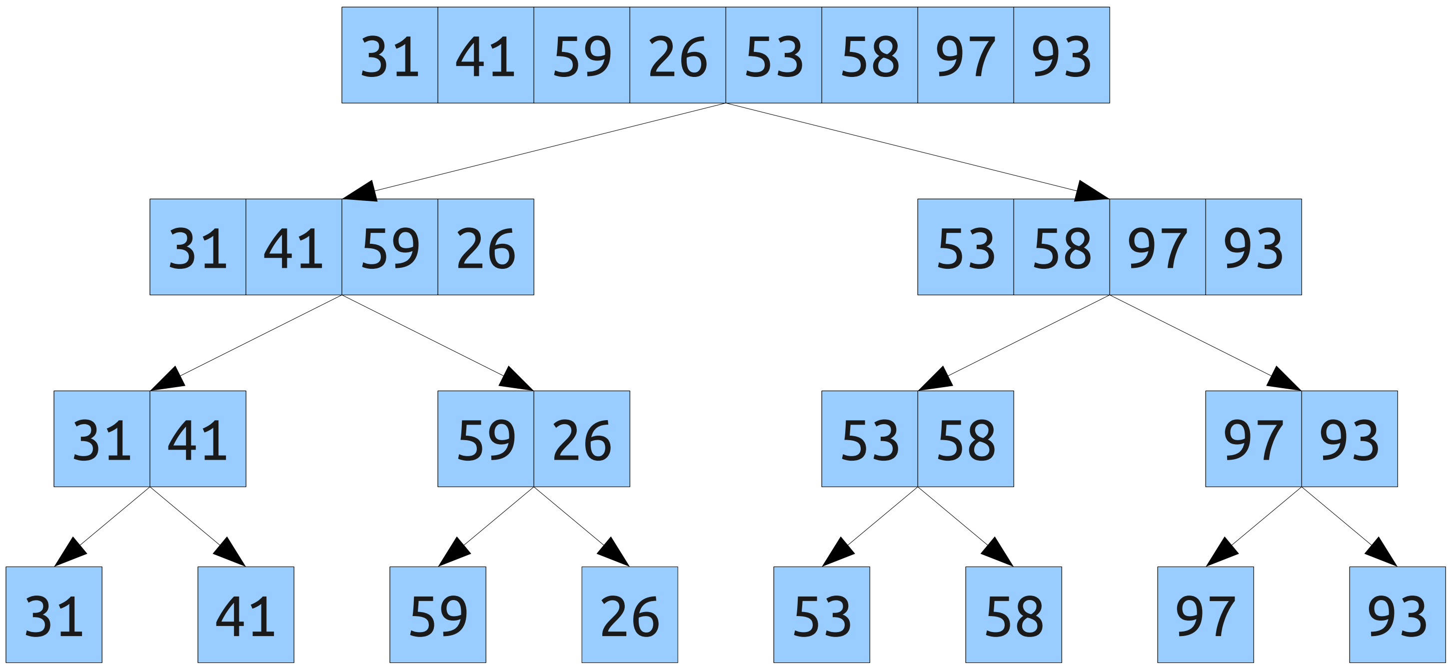
An Observation



An Observation



An Observation



Observation 1: Every recursive call that will ever be made doing RMQ this way must use one of the subarrays given here.

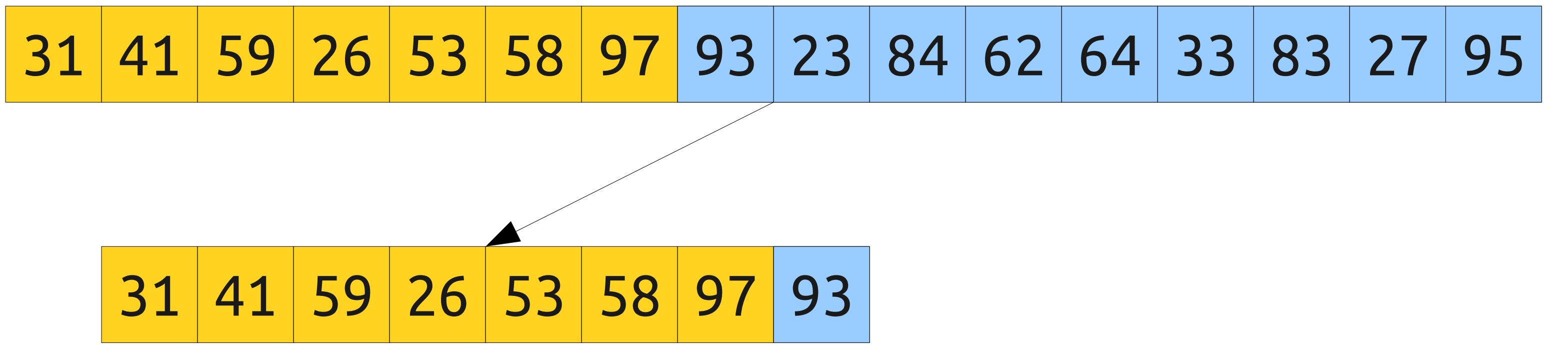
A Second Observation

31	41	59	26	53	58	97	93	23	84	62	64	33	83	27	95

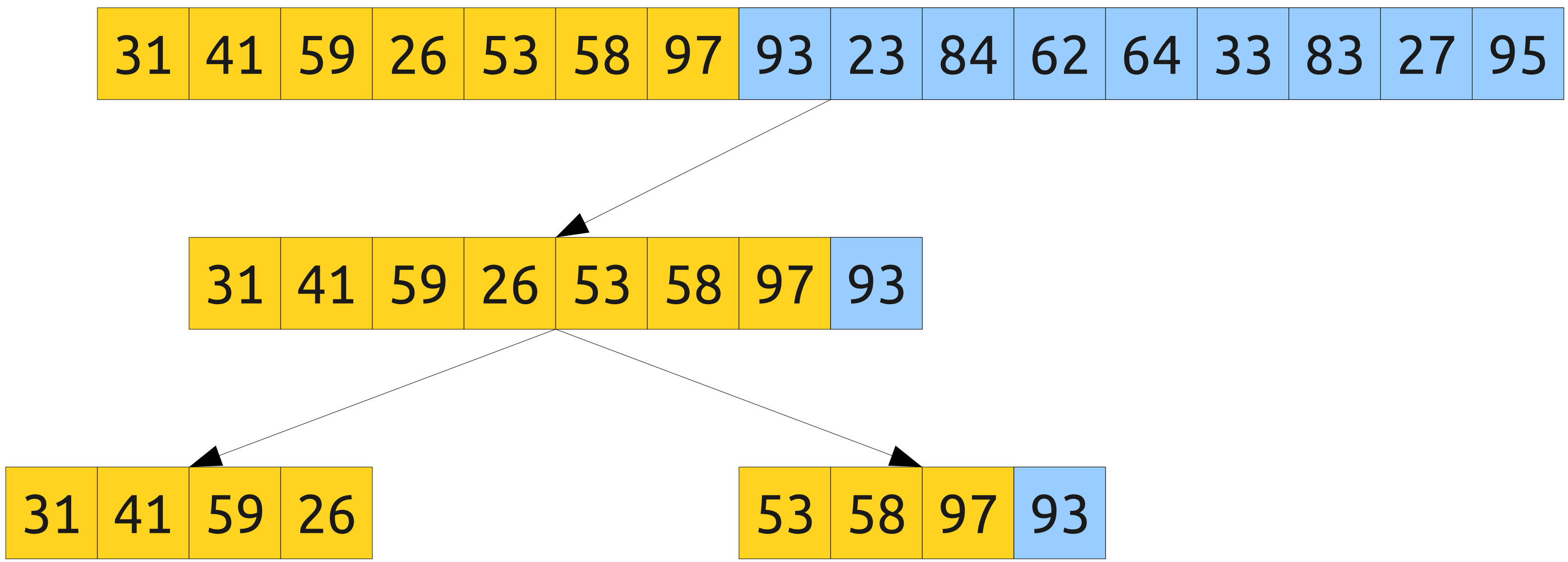
A Second Observation

31	41	59	26	53	58	97	93	23	84	62	64	33	83	27	95

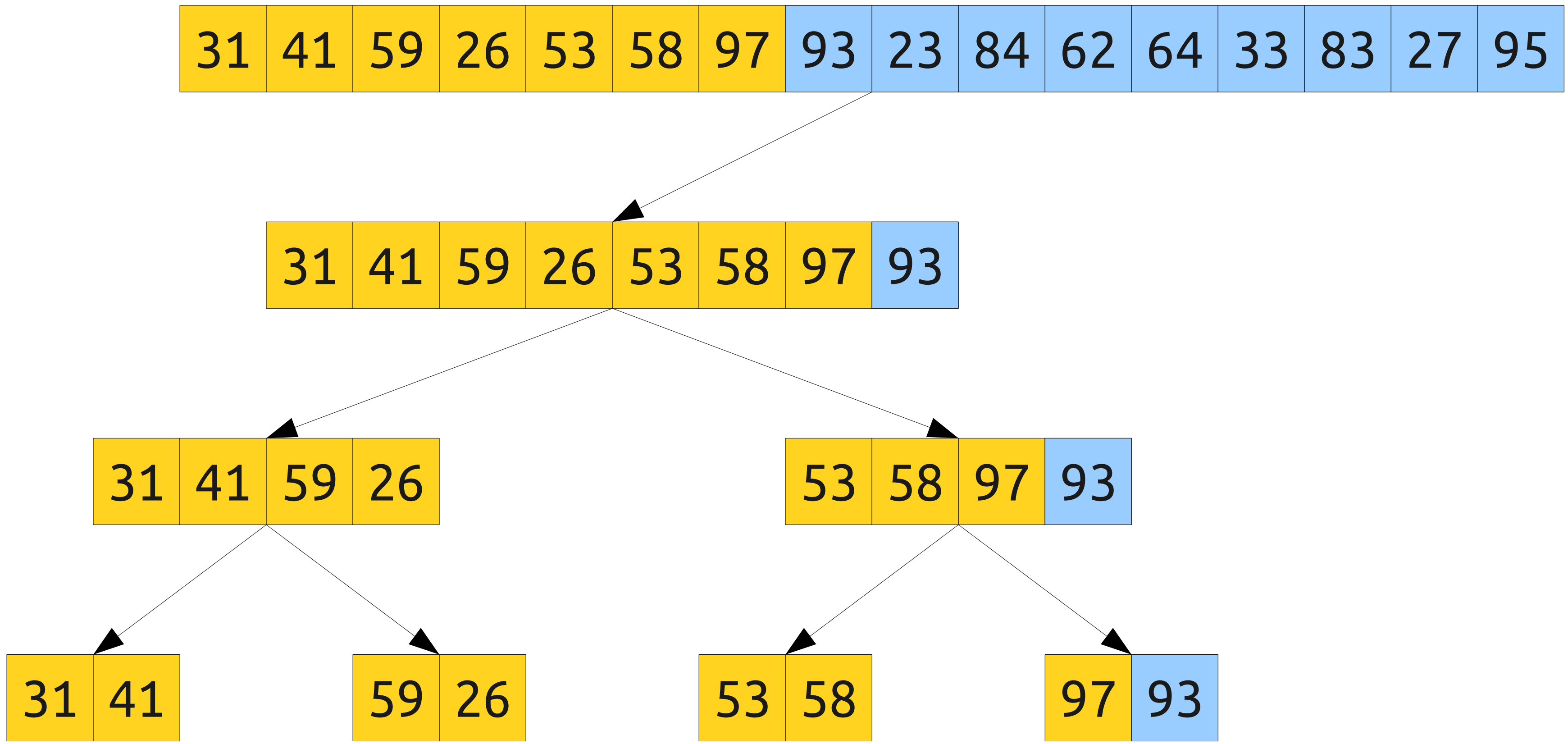
A Second Observation



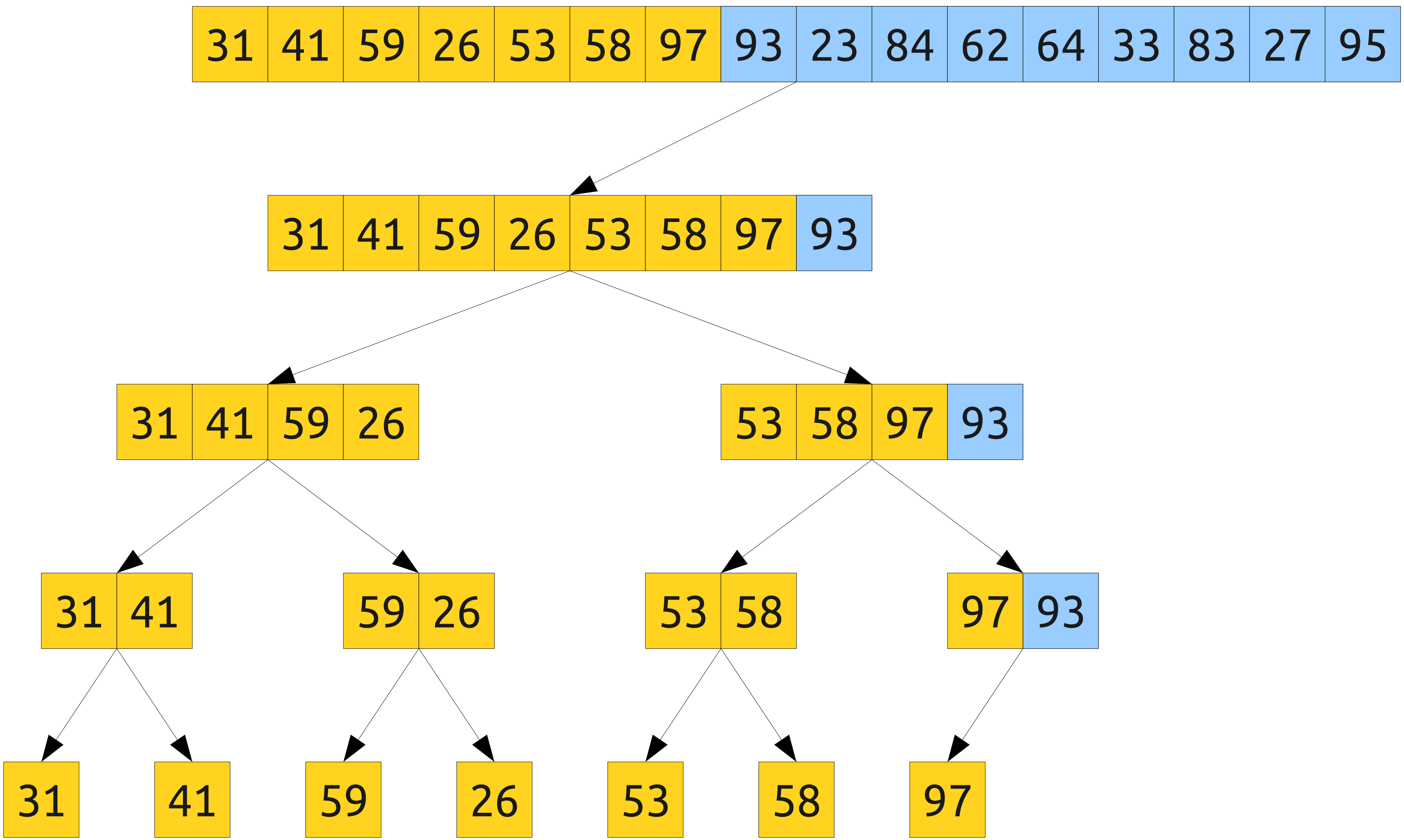
A Second Observation



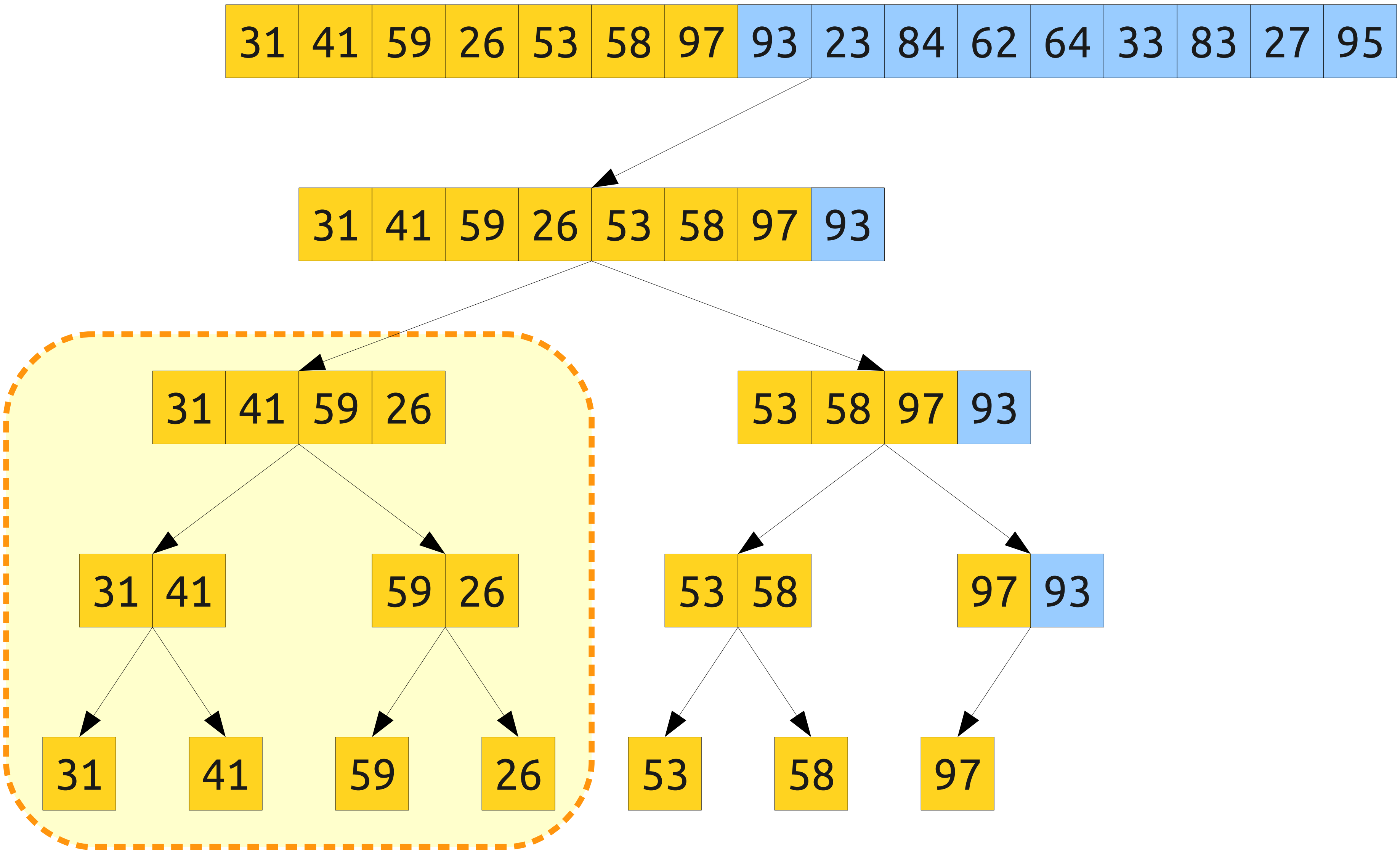
A Second Observation



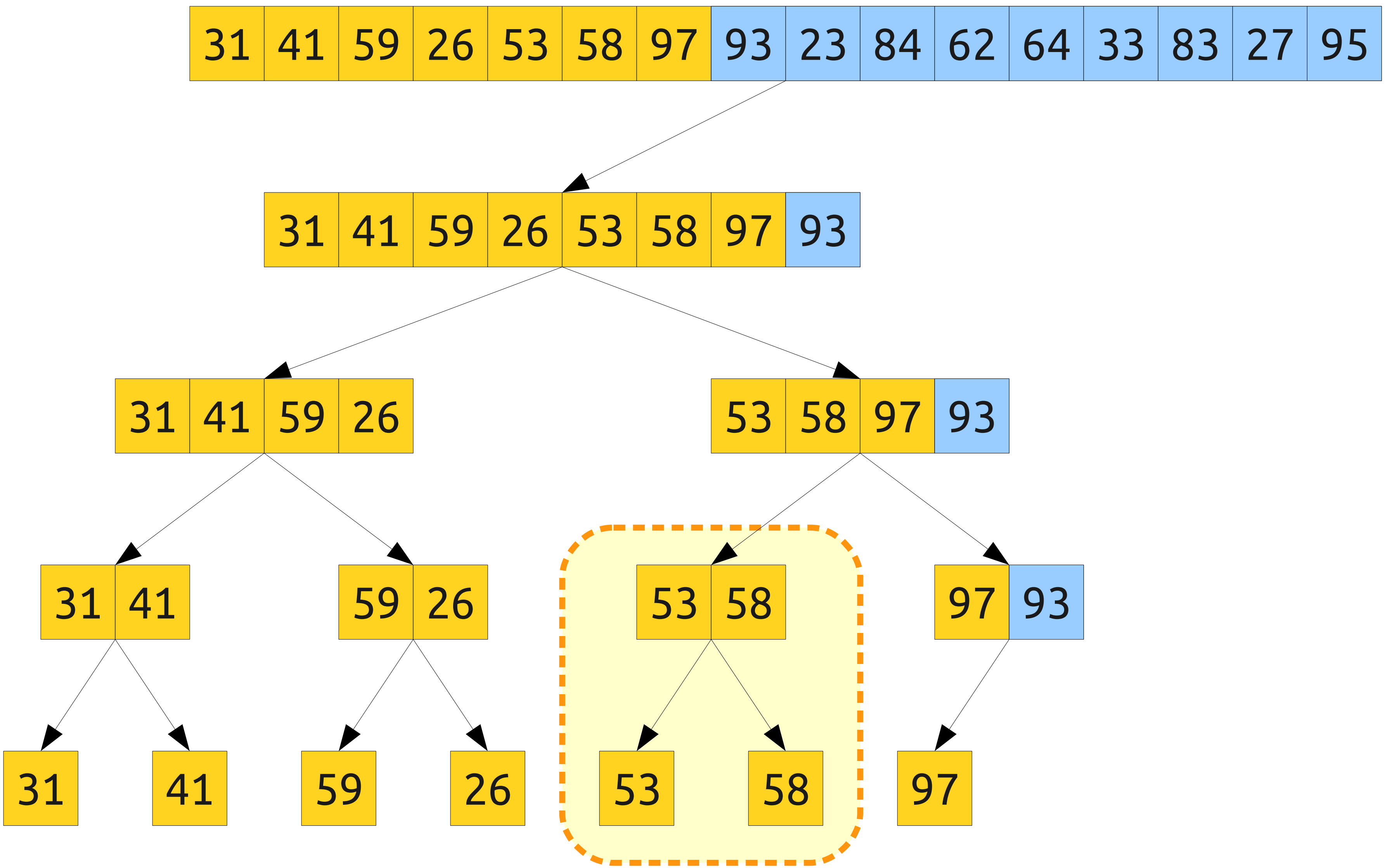
A Second Observation



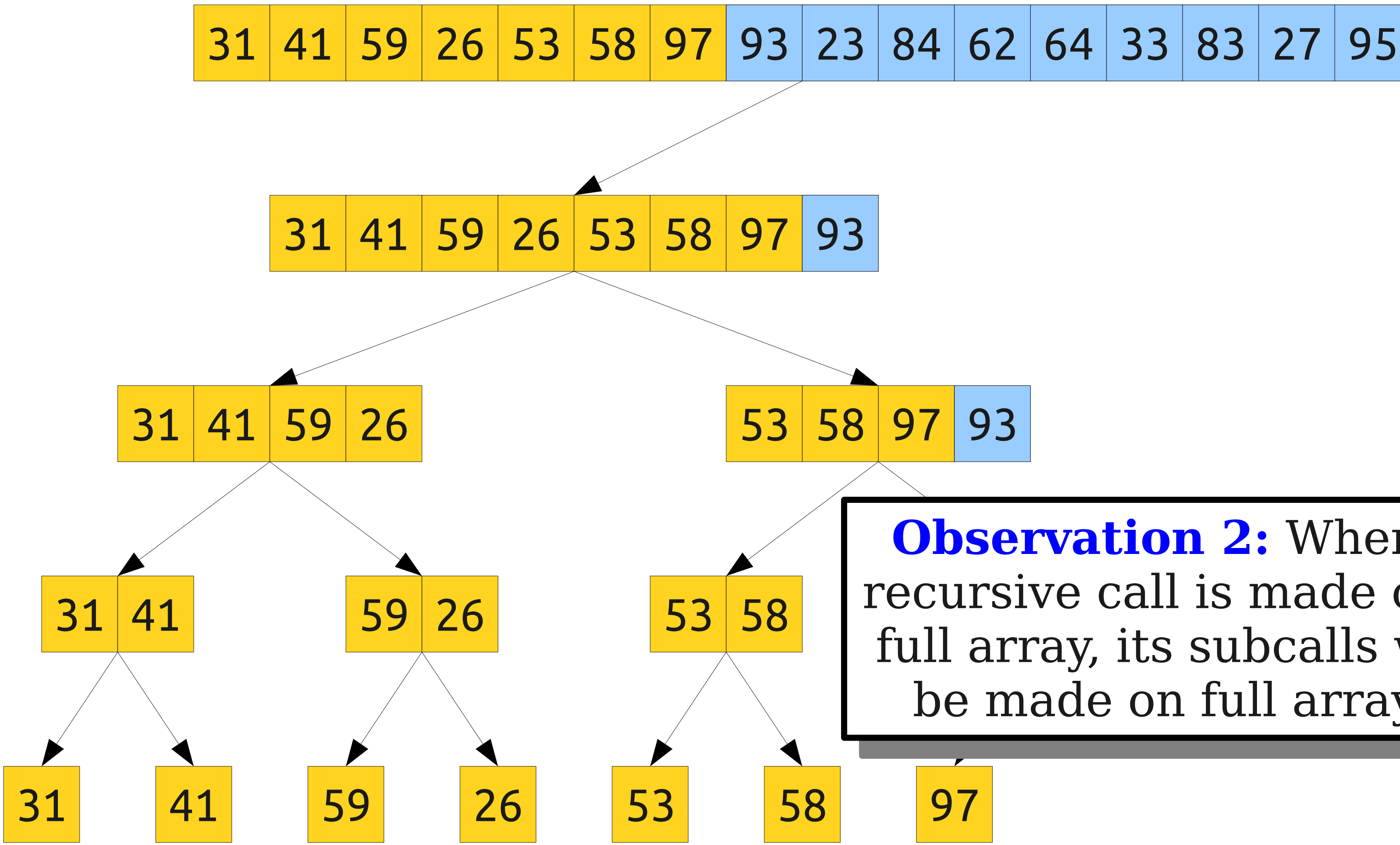
A Second Observation



A Second Observation



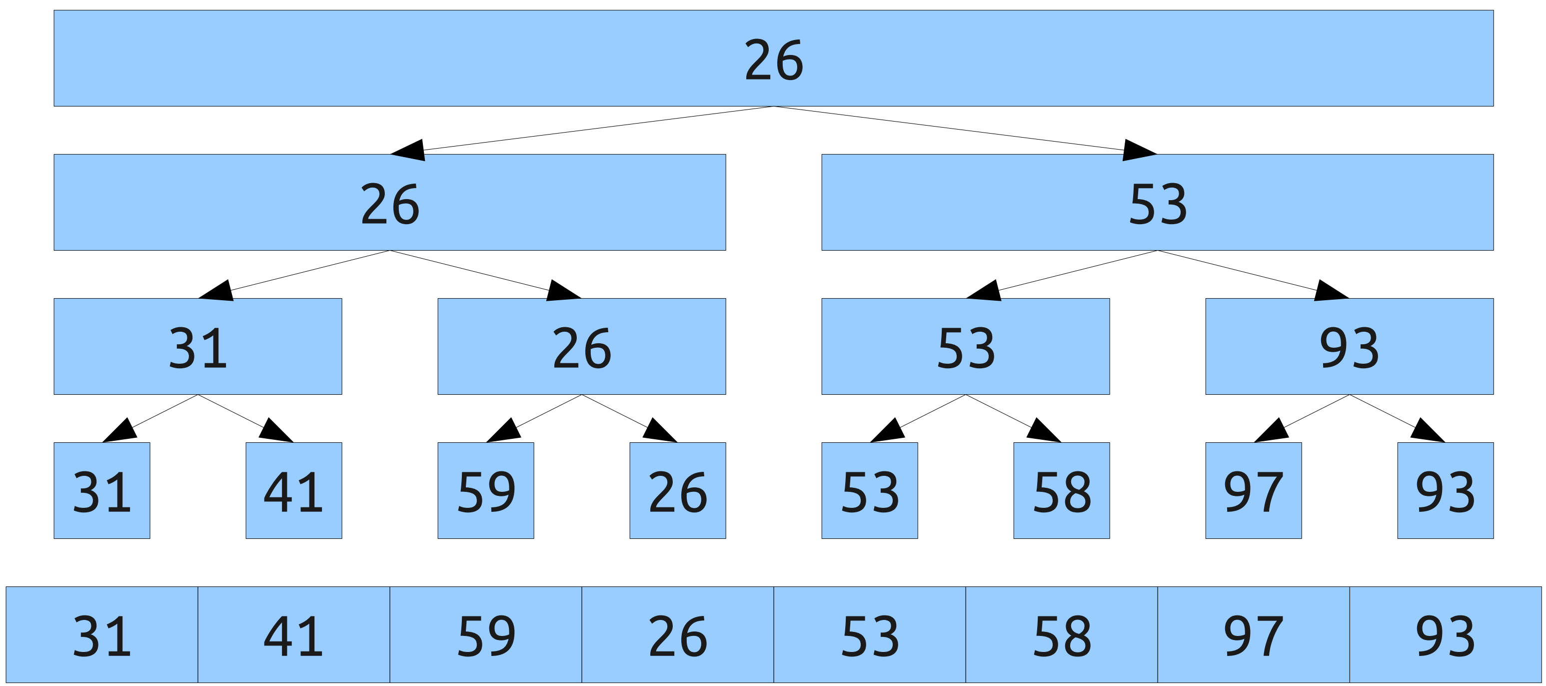
A Second Observation



Observation 2: When a recursive call is made on a full array, its subcalls will be made on full arrays.

A Revised Idea

- For each subarray that could *ever* be visited by a recursive call, compute the minimum of that subarray and store it.
- Store result as a **segment tree**:



- Can be built with O(n) preprocessing. *(why)?*

A Revised Idea

- Modify the recursive algorithm to use the segment tree.
- If range to search equals the range at the current node, return the minimum value in that range.
  - We precomputed this; takes time O(1).
- Otherwise:
  - If range is purely in the first or second half, recurse on that subrange.
  - Otherwise, split the range in half, then recursively search the left and right halves and take the minimum.

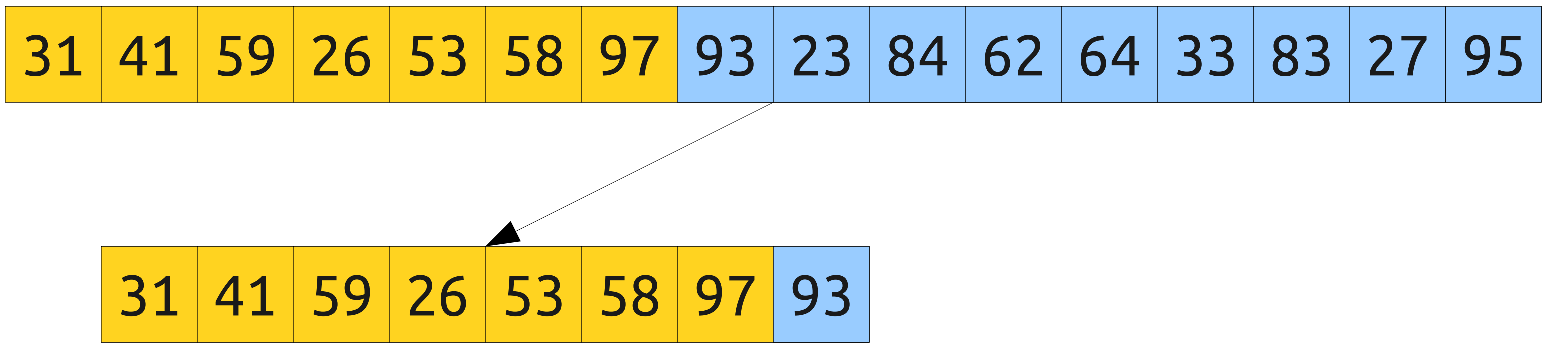
Segment Trees

31	41	59	26	53	58	97	93	23	84	62	64	33	83	27	95

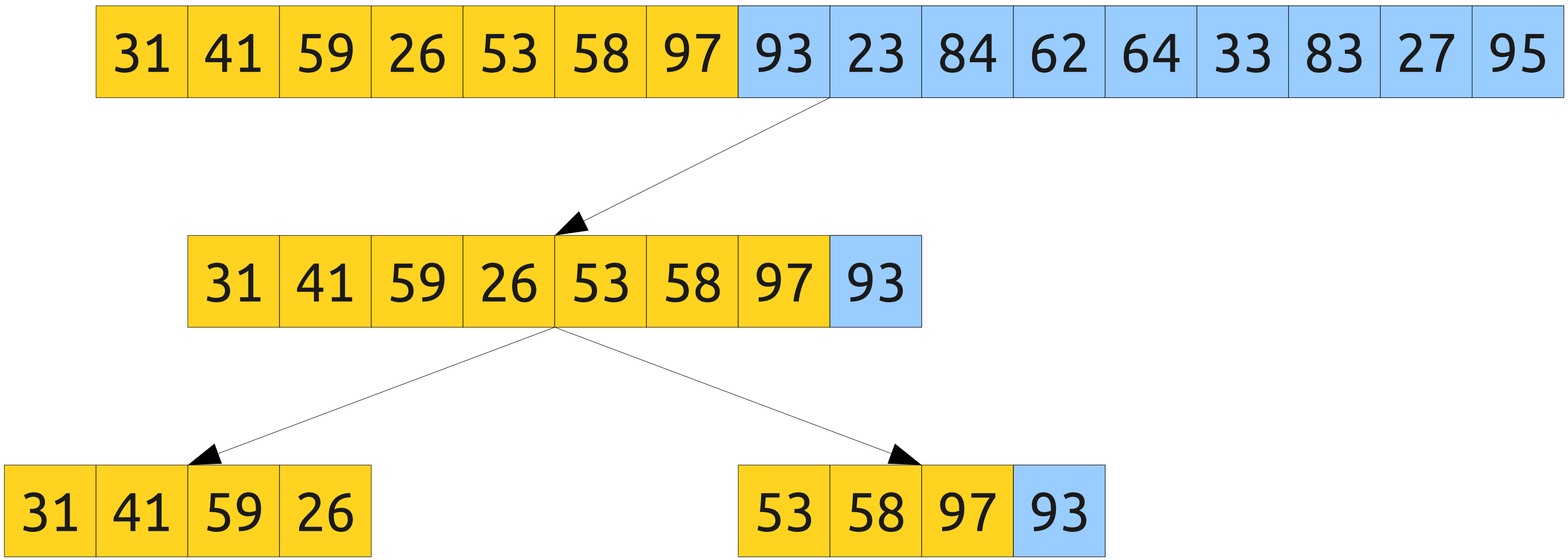
Segment Trees

31	41	59	26	53	58	97	93	23	84	62	64	33	83	27	95

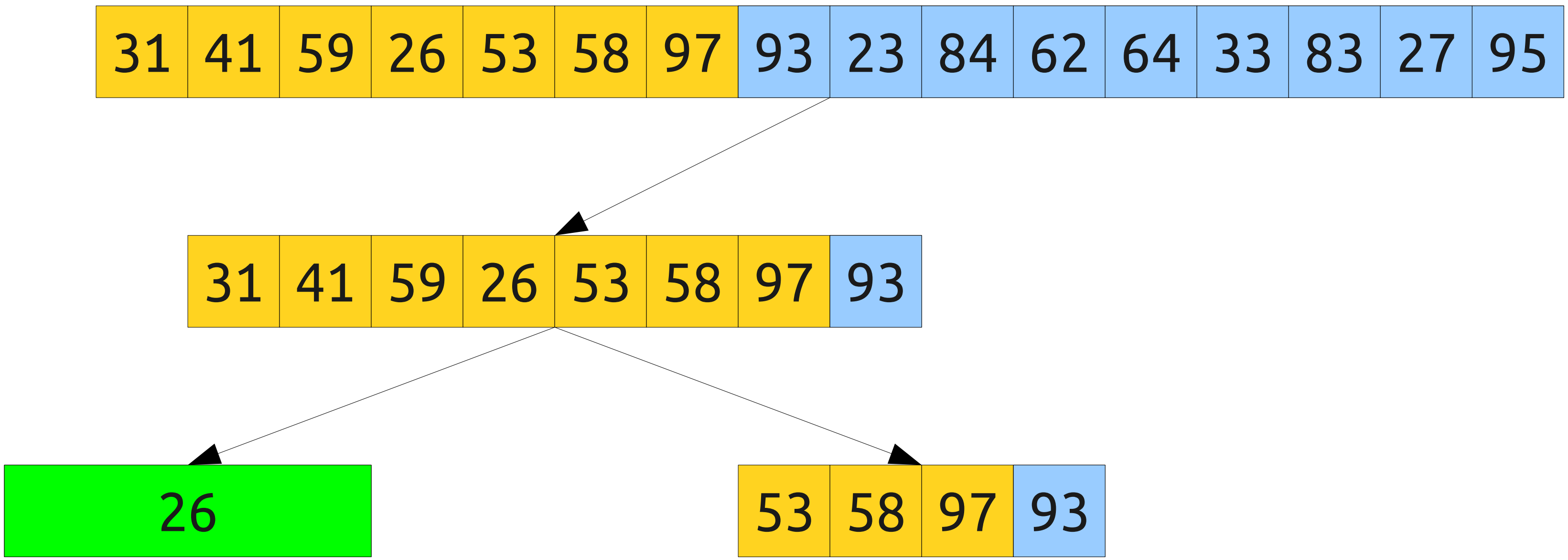
Segment Trees



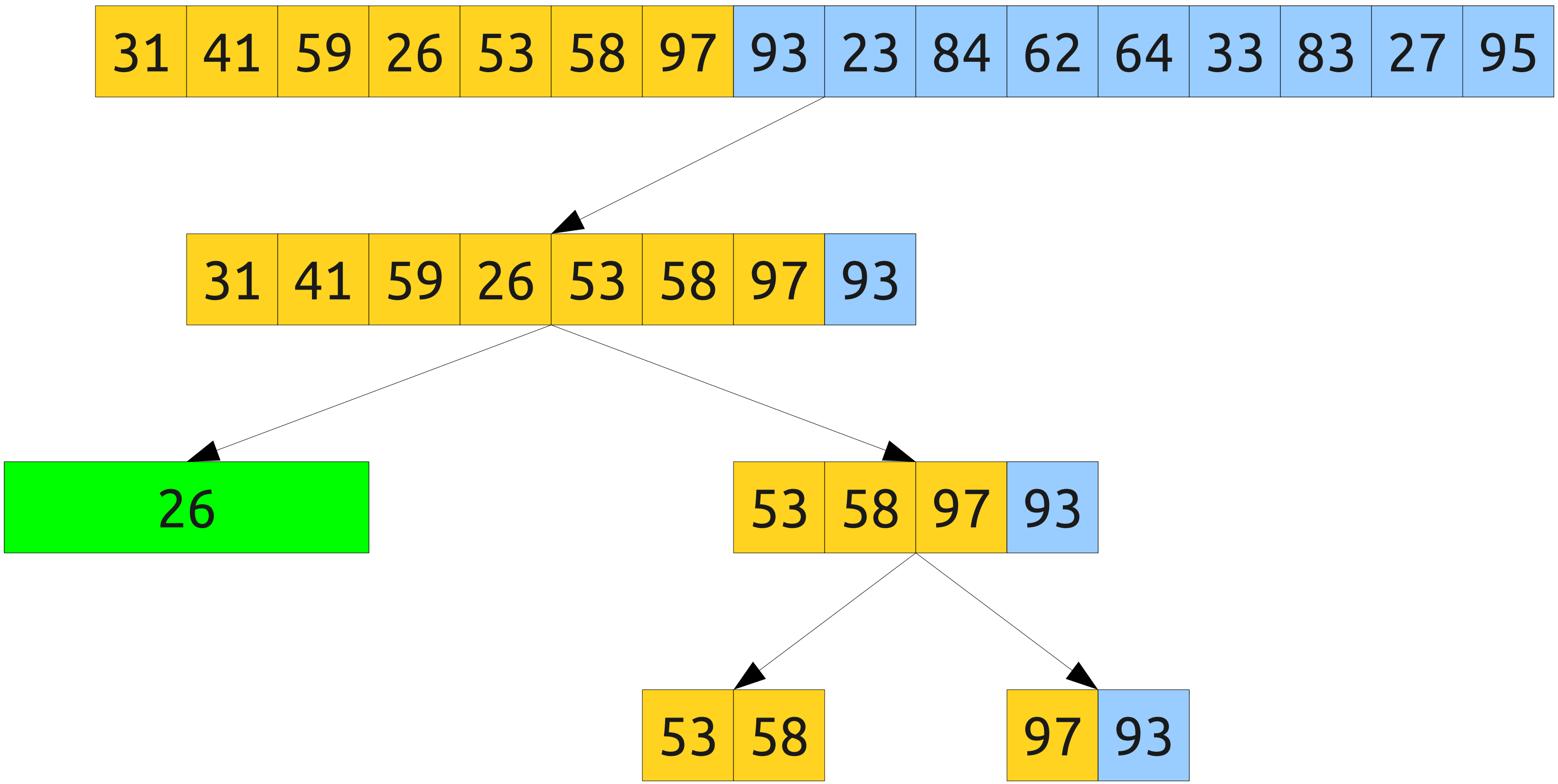
Segment Trees



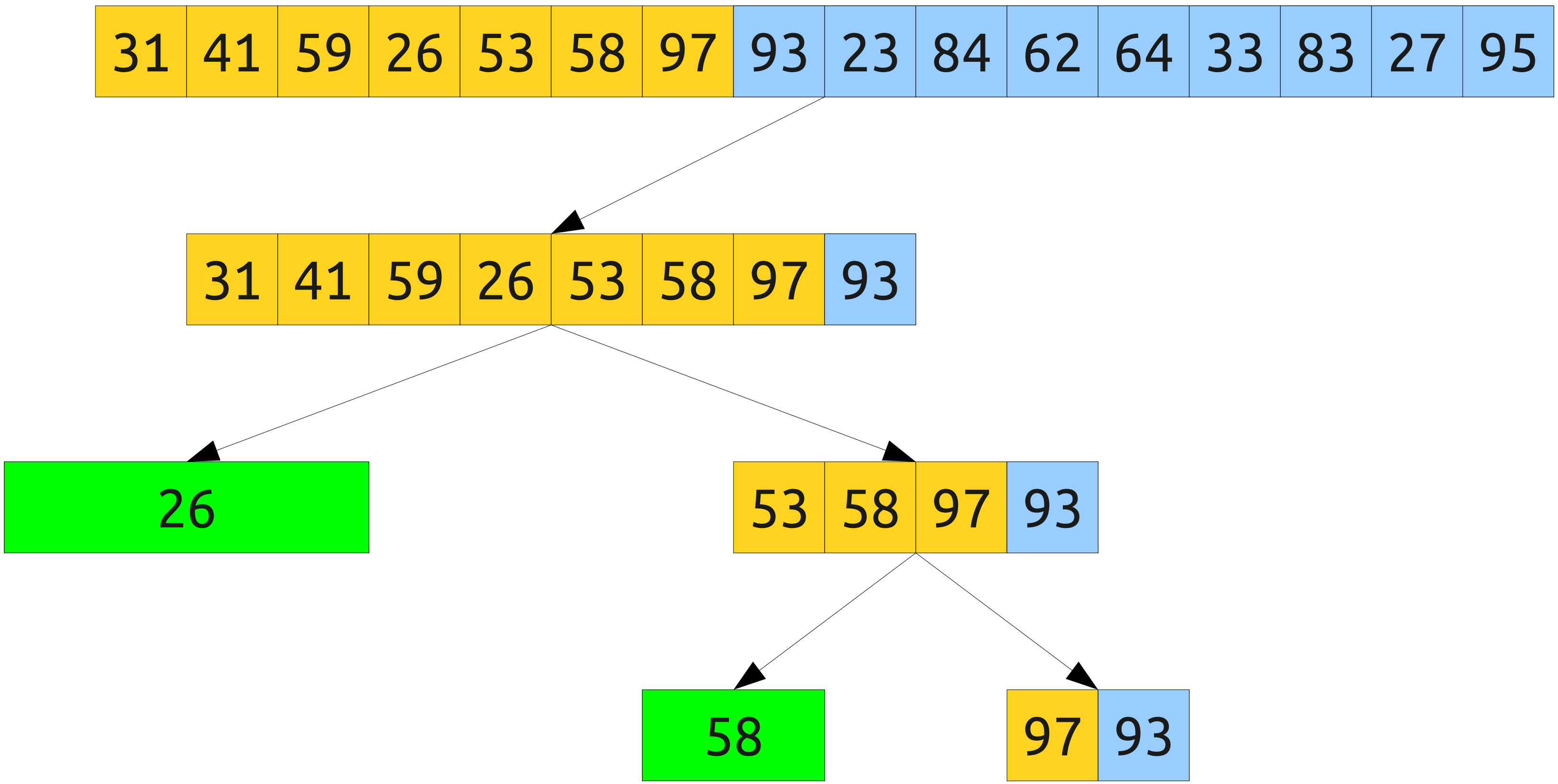
Segment Trees



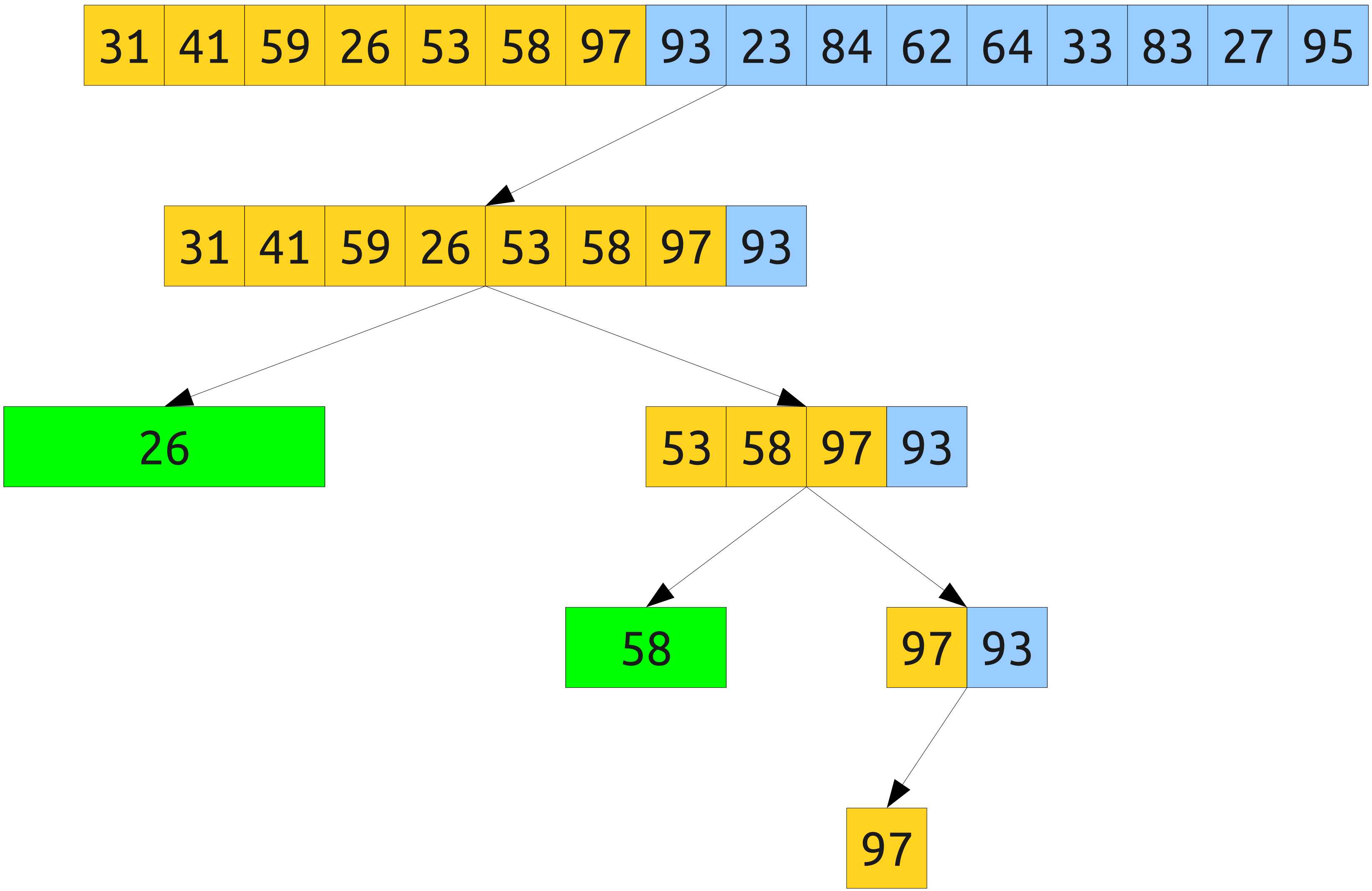
Segment Trees



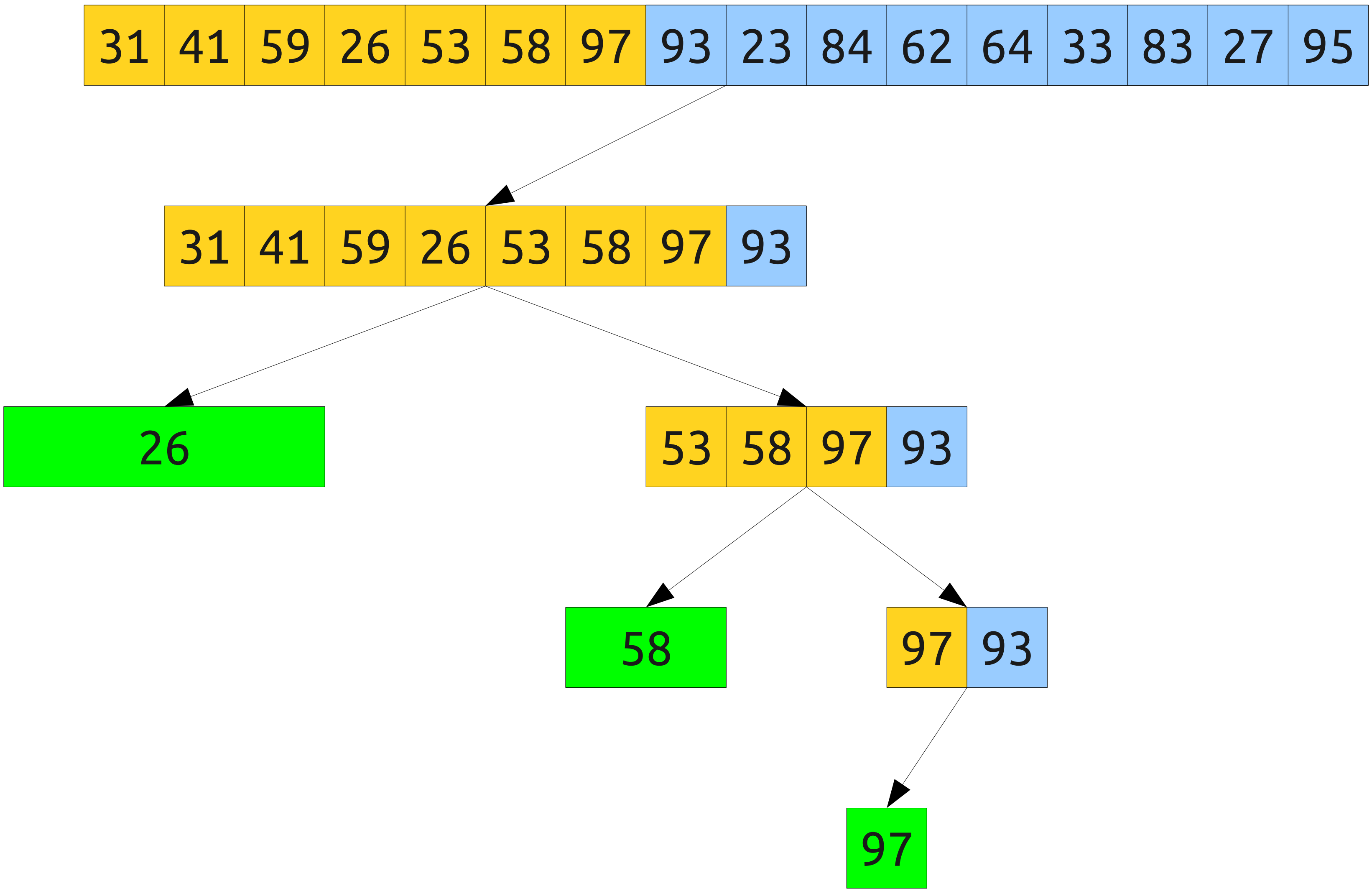
Segment Trees



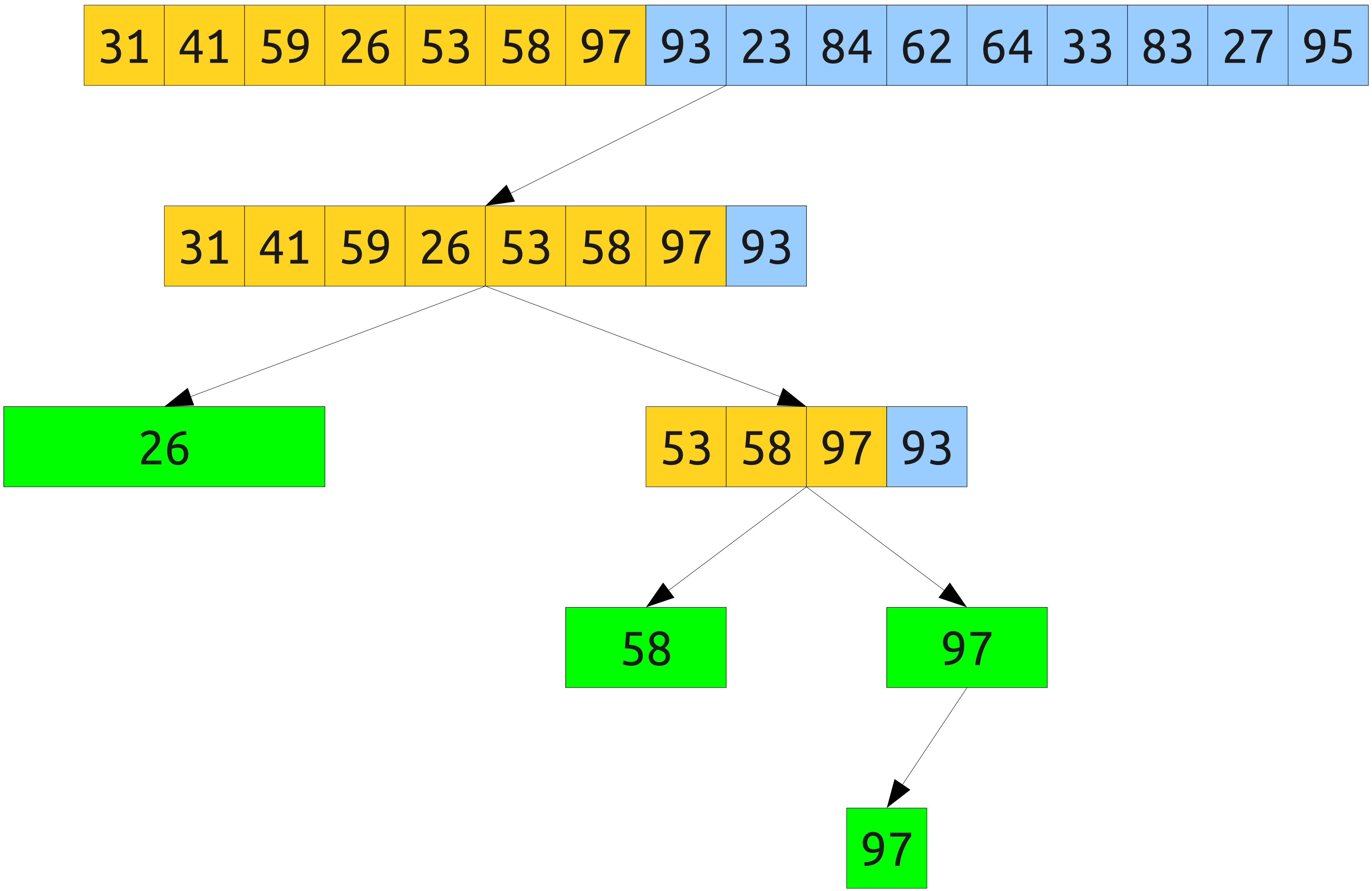
Segment Trees



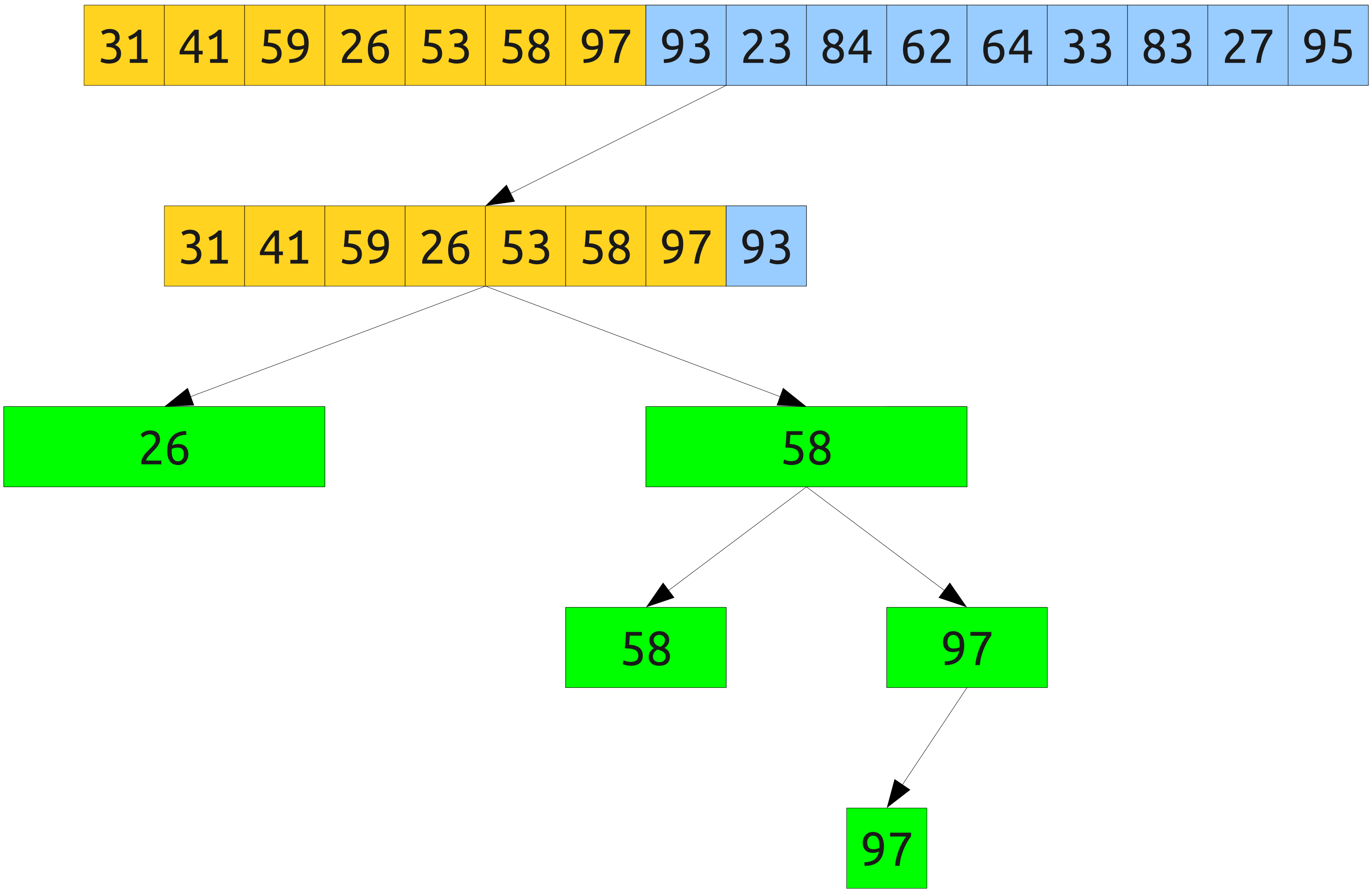
Segment Trees



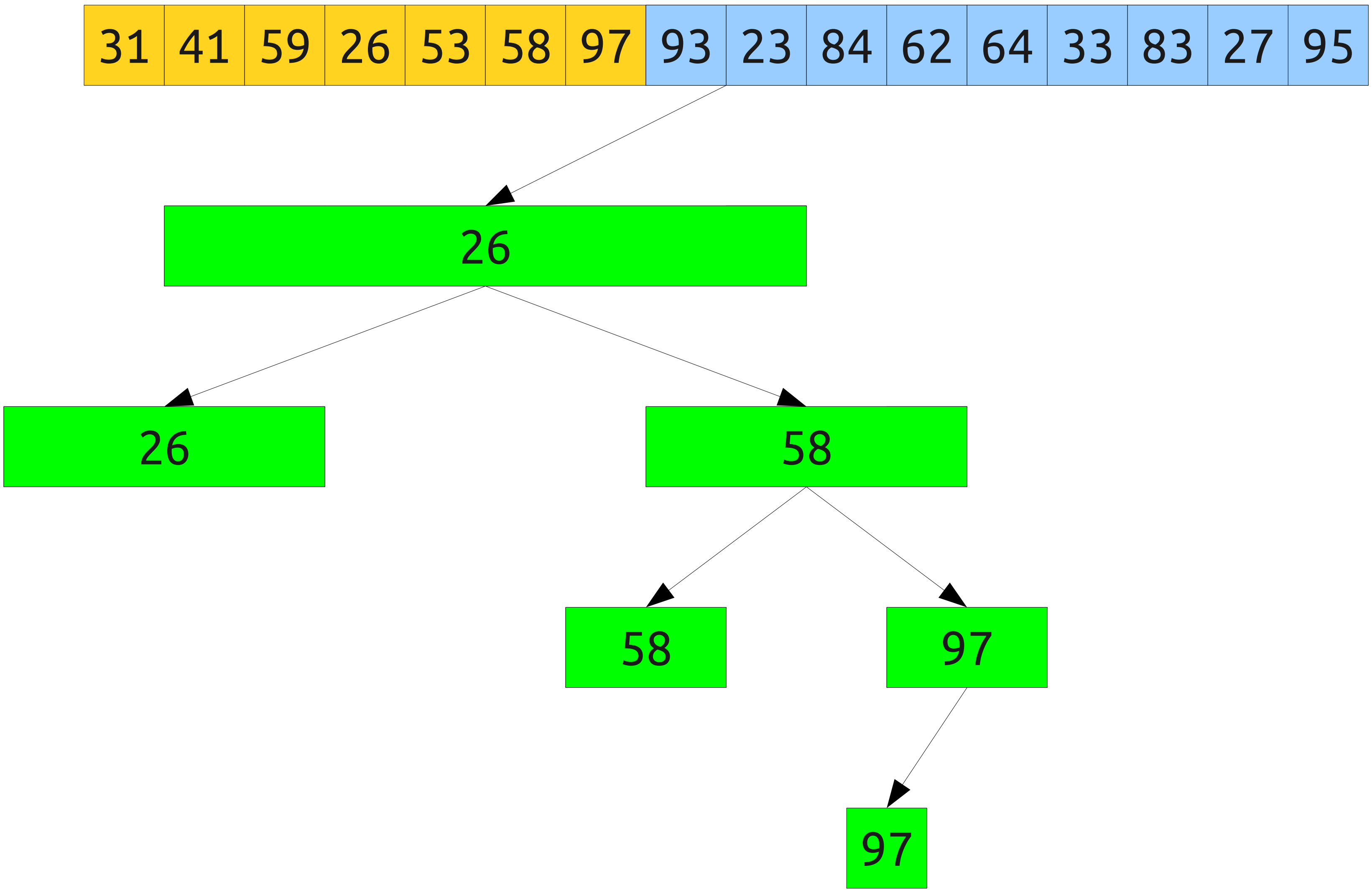
Segment Trees



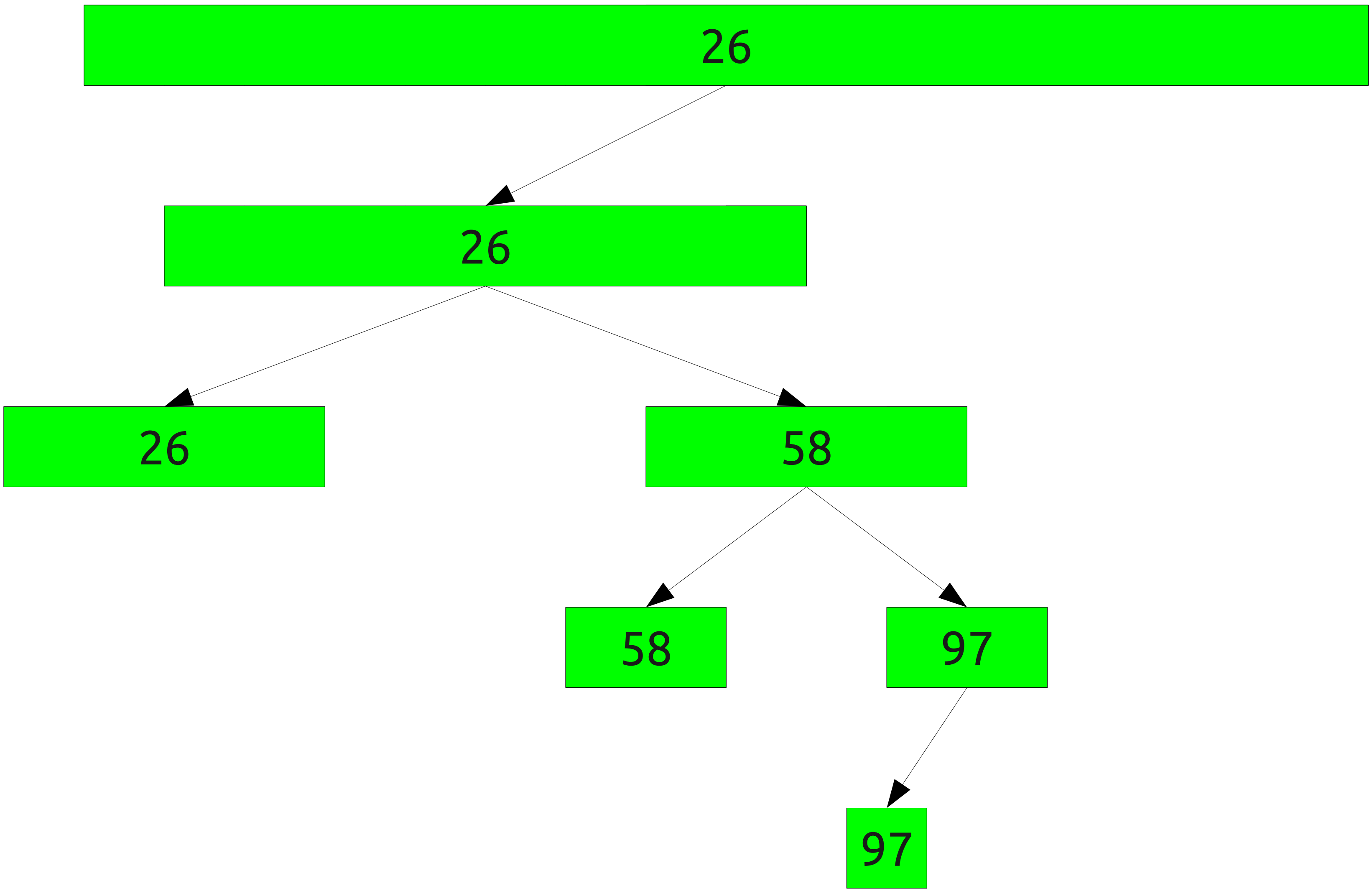
Segment Trees



Segment Trees



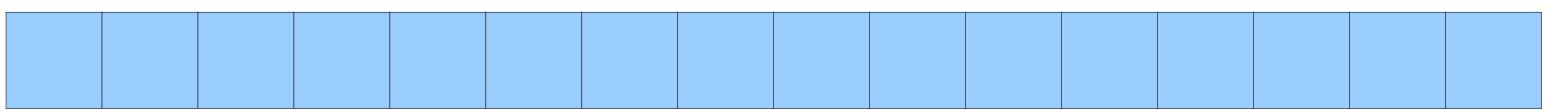
Segment Trees



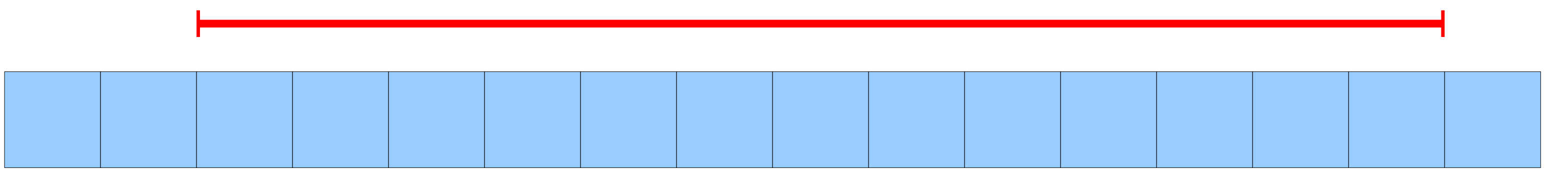
Is This Faster?

- The root cause of the inefficiency in the initial approach was the branching recursion, which is still present in this new solution.
- Is this new approach any faster than what we had before?
- **Claim:** Yes! In fact, queries only take time O(log n).

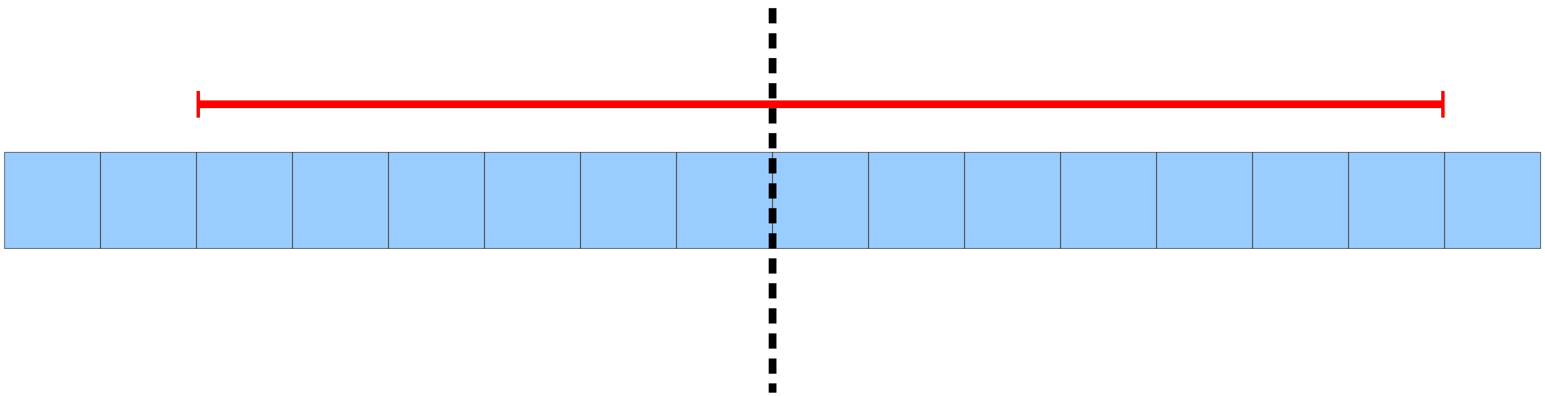
An Observation



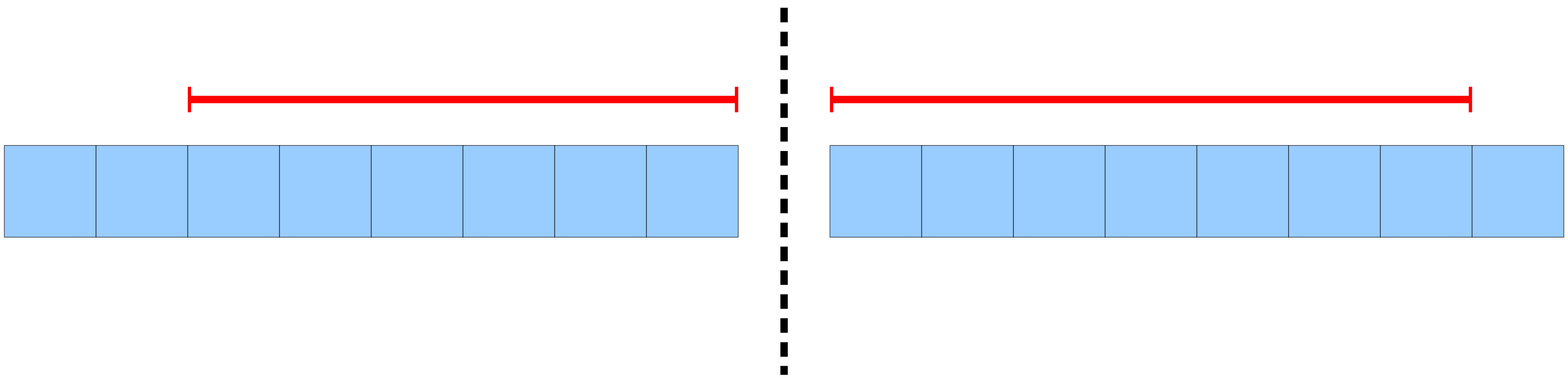
An Observation



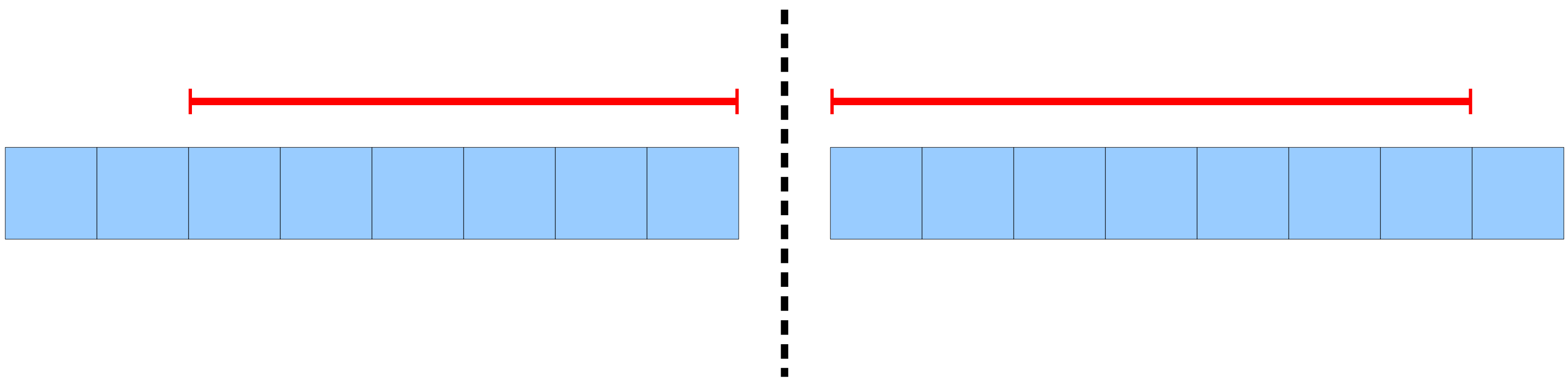
An Observation



An Observation



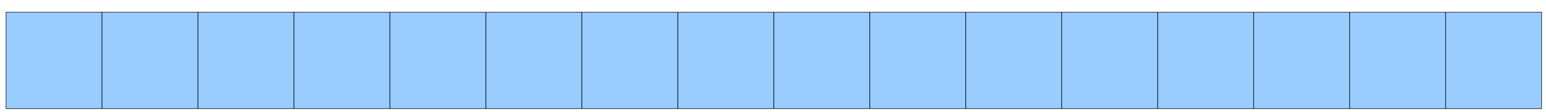
An Observation



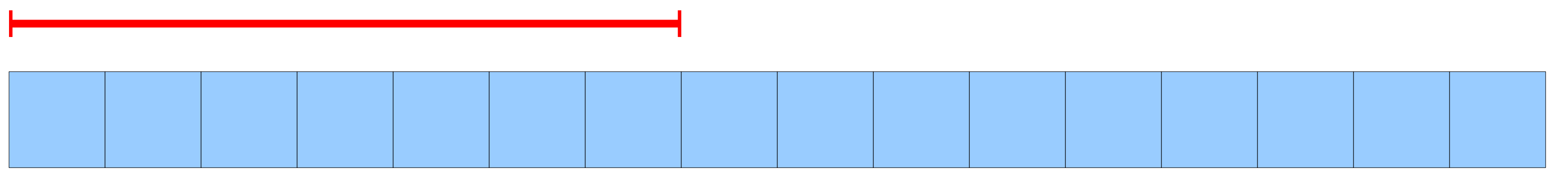
Claim 1: The first time the recursion splits, it leaves behind two ranges that are each flush against one side of the subarray.

Another Observation

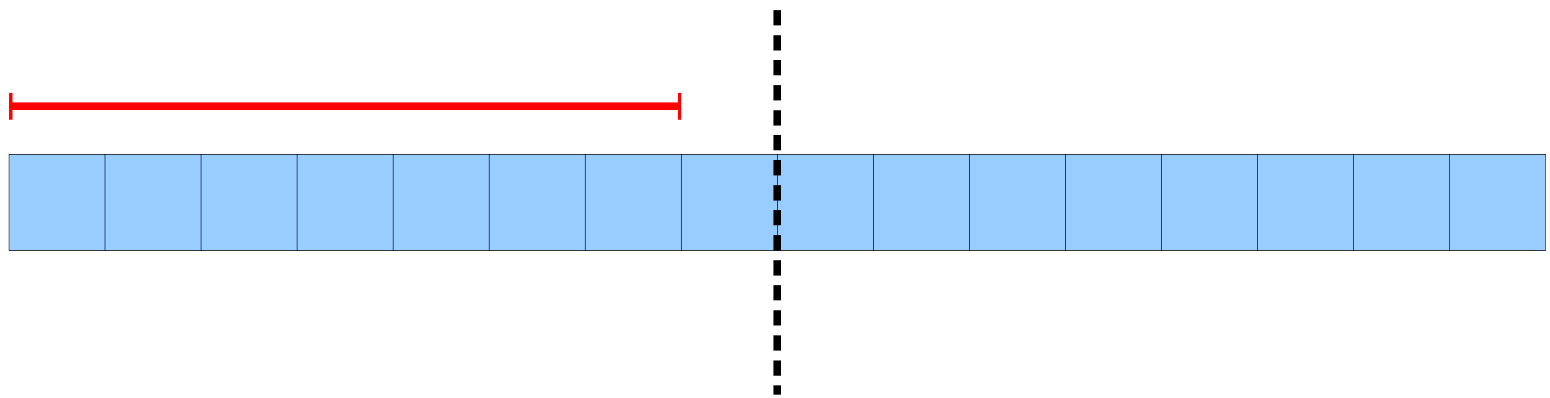
Another Observation



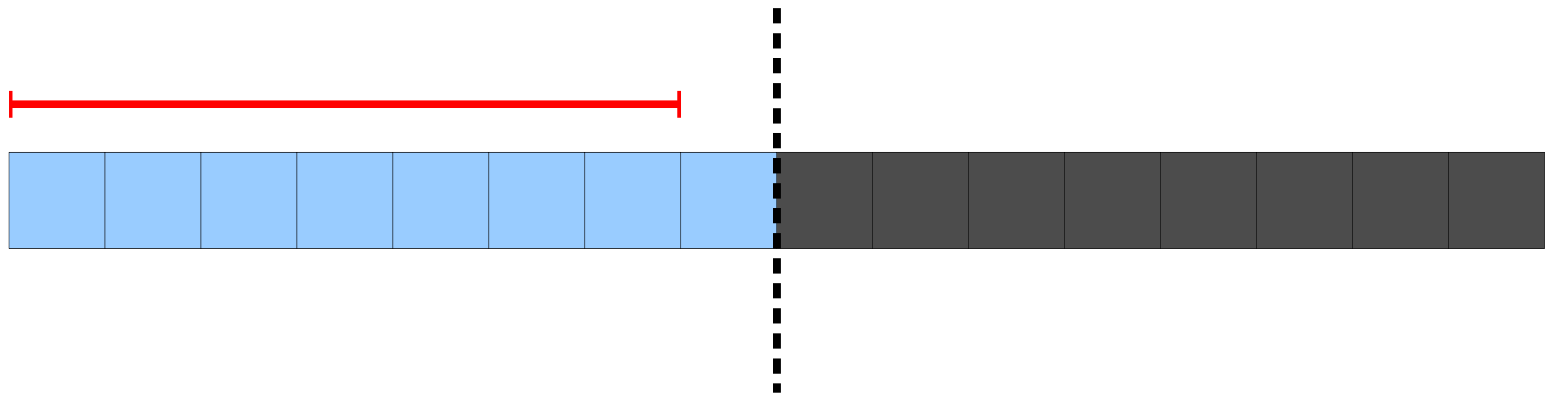
Another Observation



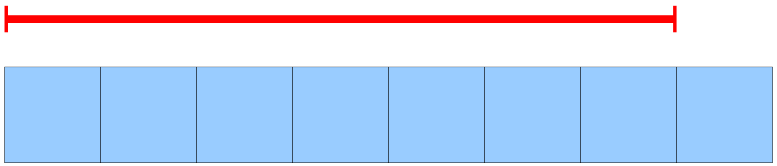
Another Observation



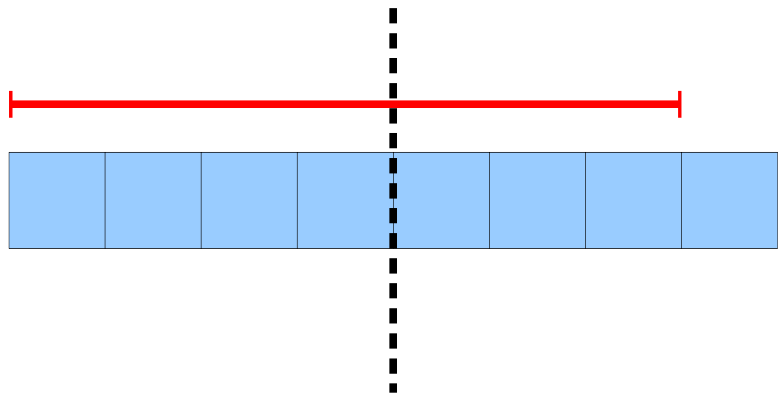
Another Observation



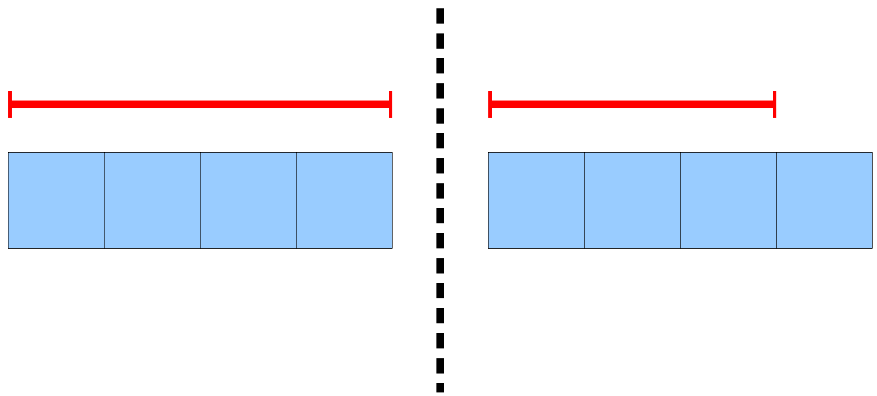
Another Observation



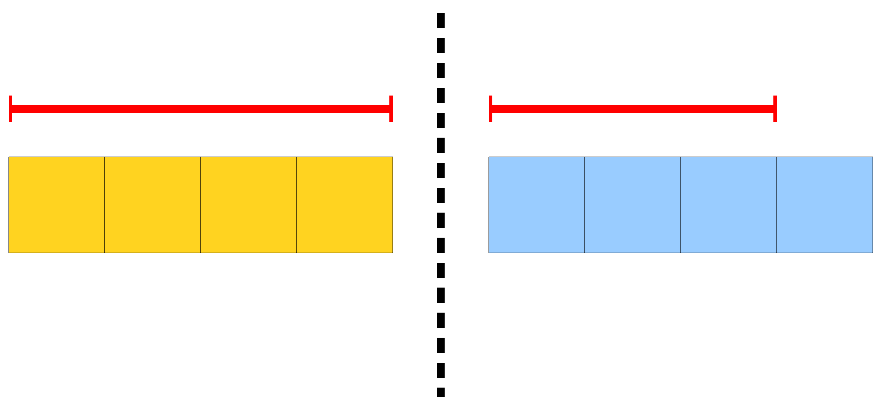
Another Observation



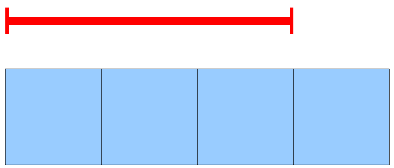
Another Observation



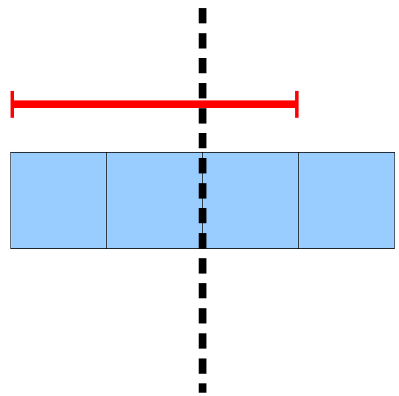
Another Observation



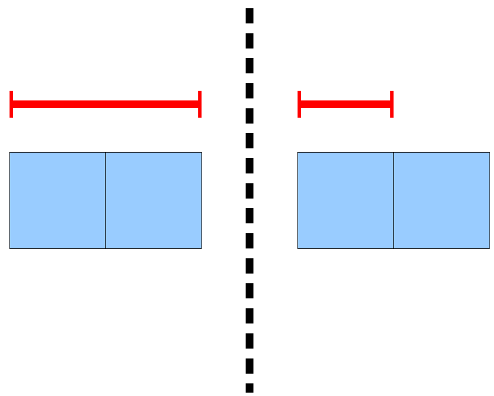
Another Observation



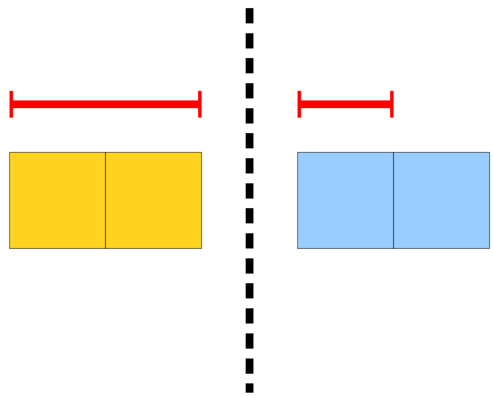
Another Observation



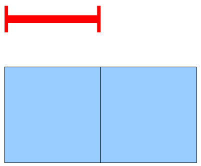
Another Observation



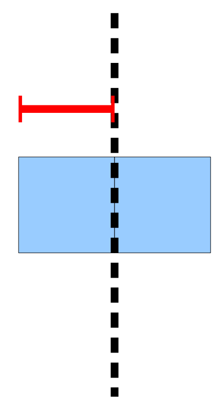
Another Observation



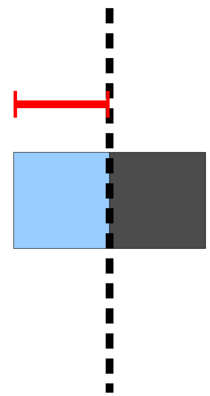
Another Observation



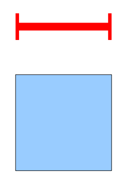
Another Observation



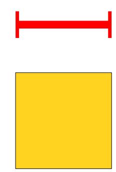
Another Observation



Another Observation



Another Observation



Another Observation

- Suppose the RMQ is over a range that is flush against one edge of the subarray.
- The recursion will then either
  - immediately terminate, or
  - recurse purely in one half of the subarray, or
  - recurse in both halves, but one of the recursive calls will immediately terminate.
- In this case, there is at most one "real" recursive call.

The Final Analysis

- If the recursion never splits into two pieces, the runtime is O(log n).
- If the recursion does split into two pieces:
  - Up until the split, we only can do O(log n) work because there is one recursive call per level.
  - After the recursion splits, each of the two pieces will have the “flush against the wall” structure and will take time only O(log n).
  - Total work done: O(log n).
- This is exponentially faster than before!

Segment Trees

- The segment tree approach requires $O(n)$ preprocessing and $O(\log n)$ time per query.
- We now have an $\langle O(n), O(\log n)\rangle$ solution to RMQ!
- To put that in perspective:
  - If we make $o(n^2 / \log n)$ queries, this is asymptotically faster than precomputing everything.
  - If we make $\omega(1)$ “large” queries, this is asymptotically faster than doing no precomputation.