MERGE SORT!

WHAT WE'VE COVERED?

Since the second week of this semester we've covered Linear Search, Binary Search, Bubble Sort, Selection Sort, and Insertion Sort!

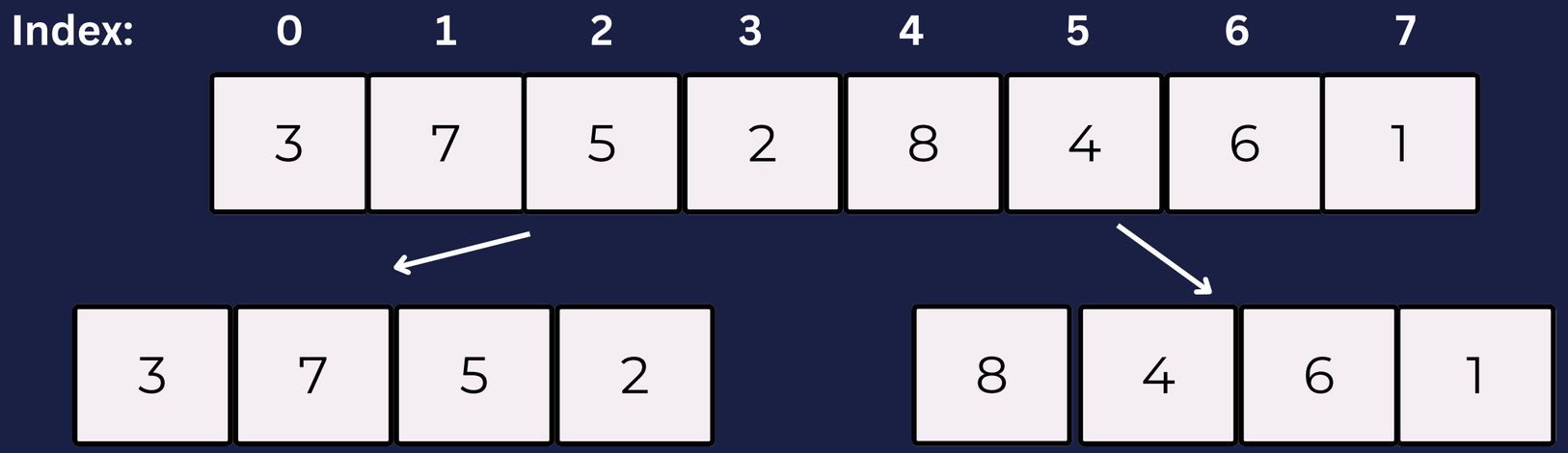
WHAT WE ARE LEARNING TODAY

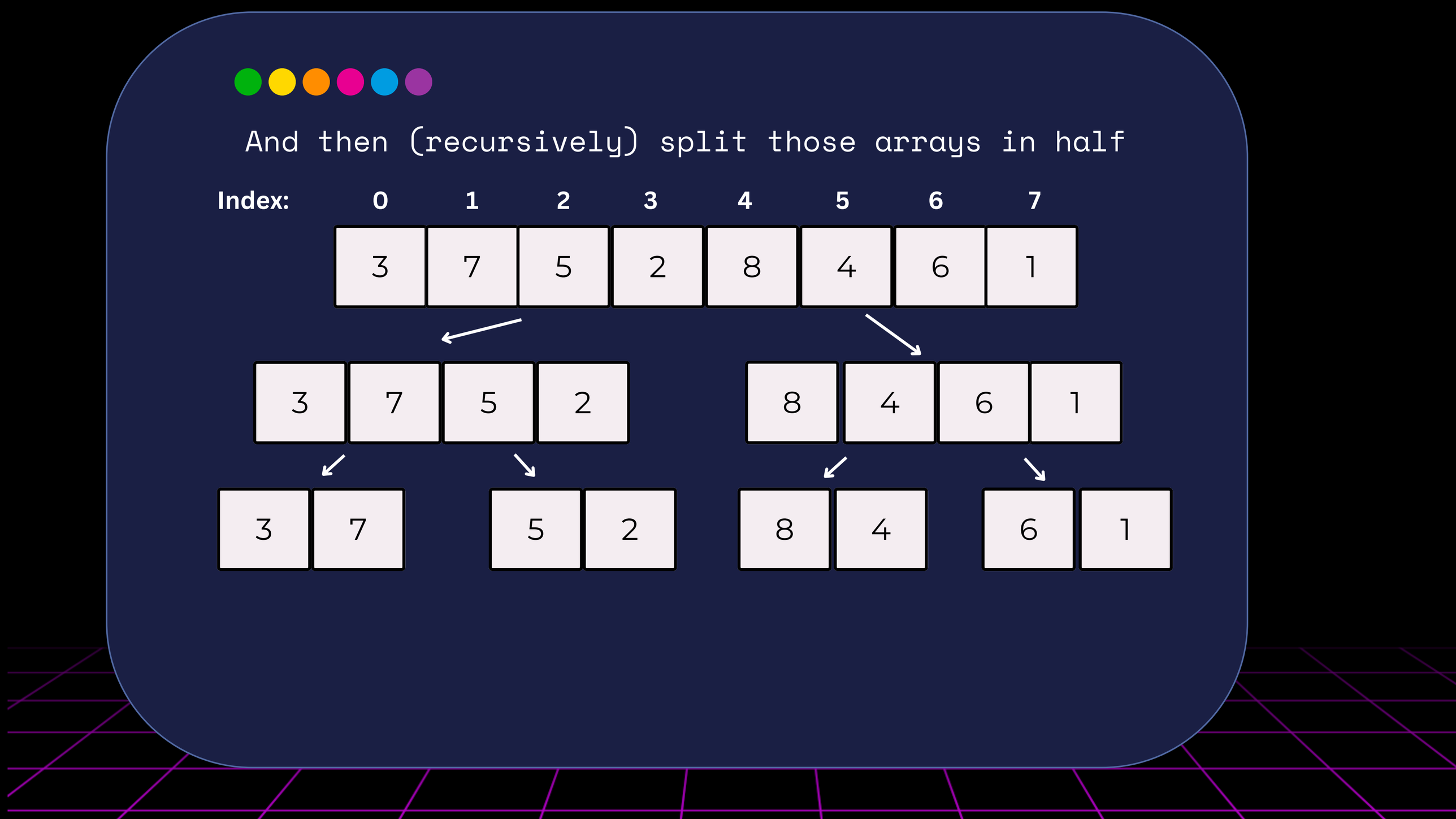
Today we are covering Merge Sort!
Merge Sort is an efficient, stable sorting algorithm that uses a divide-and-conquer approach to recursively sort sub-arrays and merge them back together.

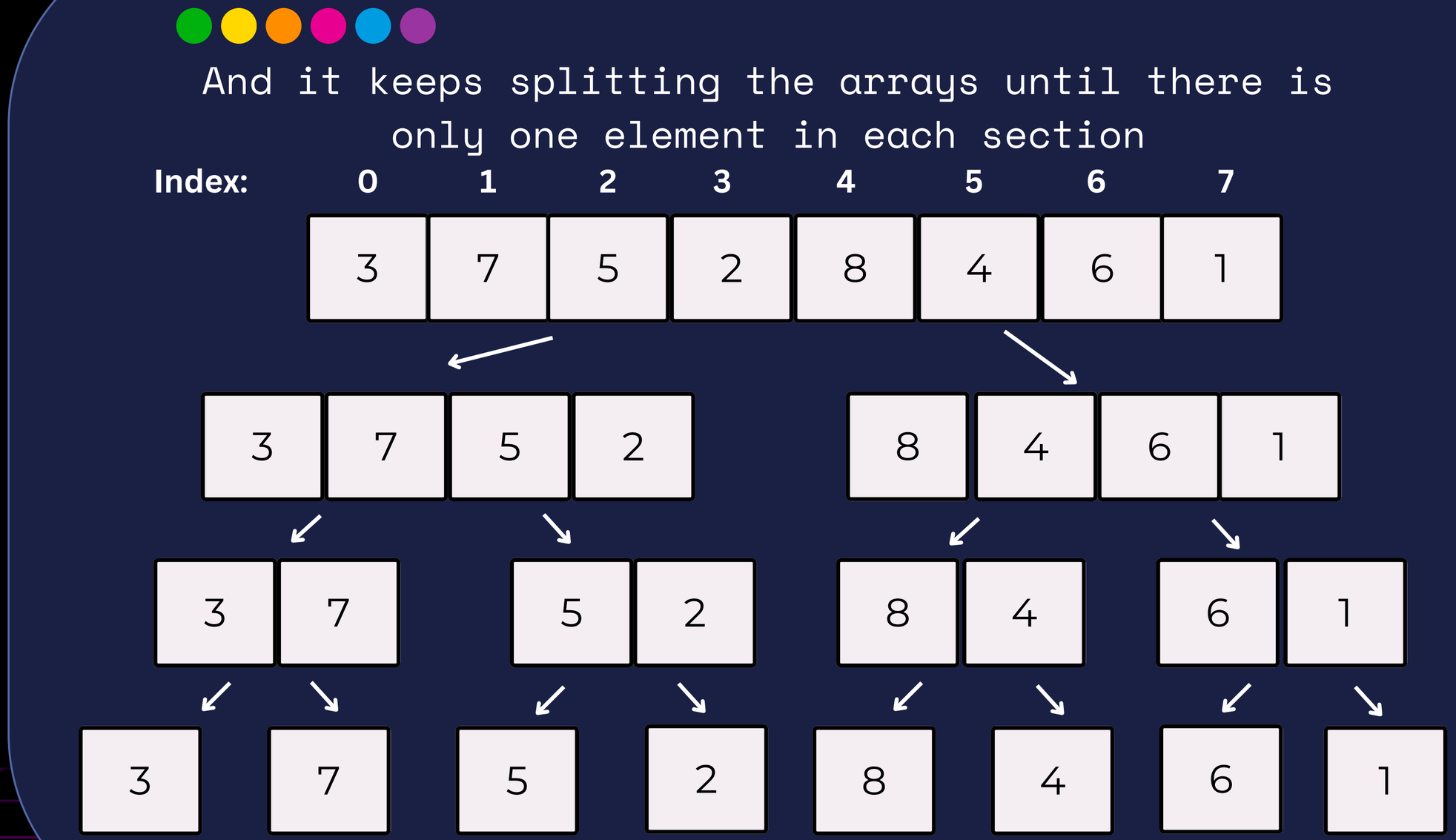
Lets consider an array of integer values:

Index:	0	1	2	3	4	5	6	7
	3	7	5	2	8	4	6	1

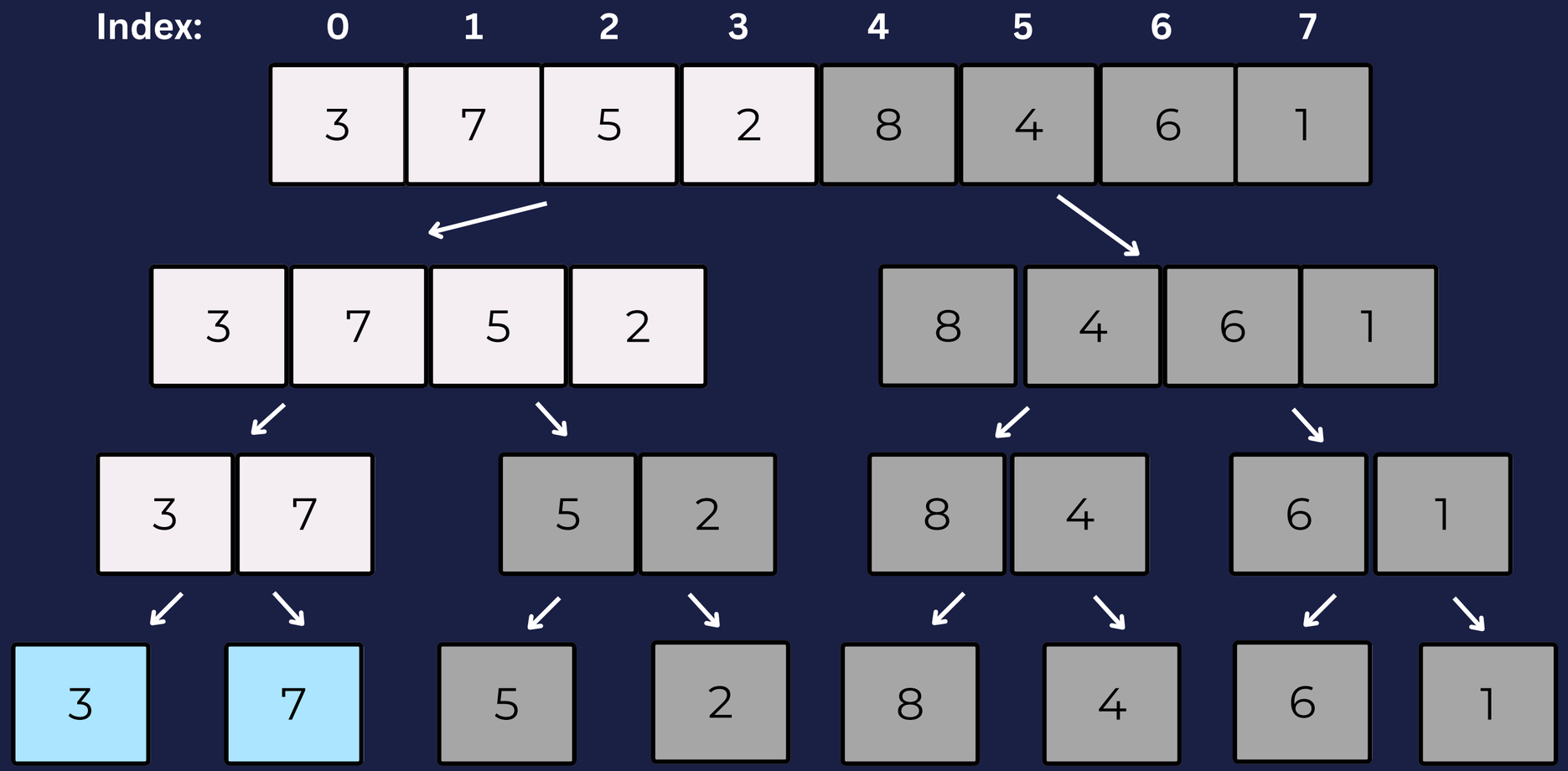
With Merge Sort, it would divide the array in half



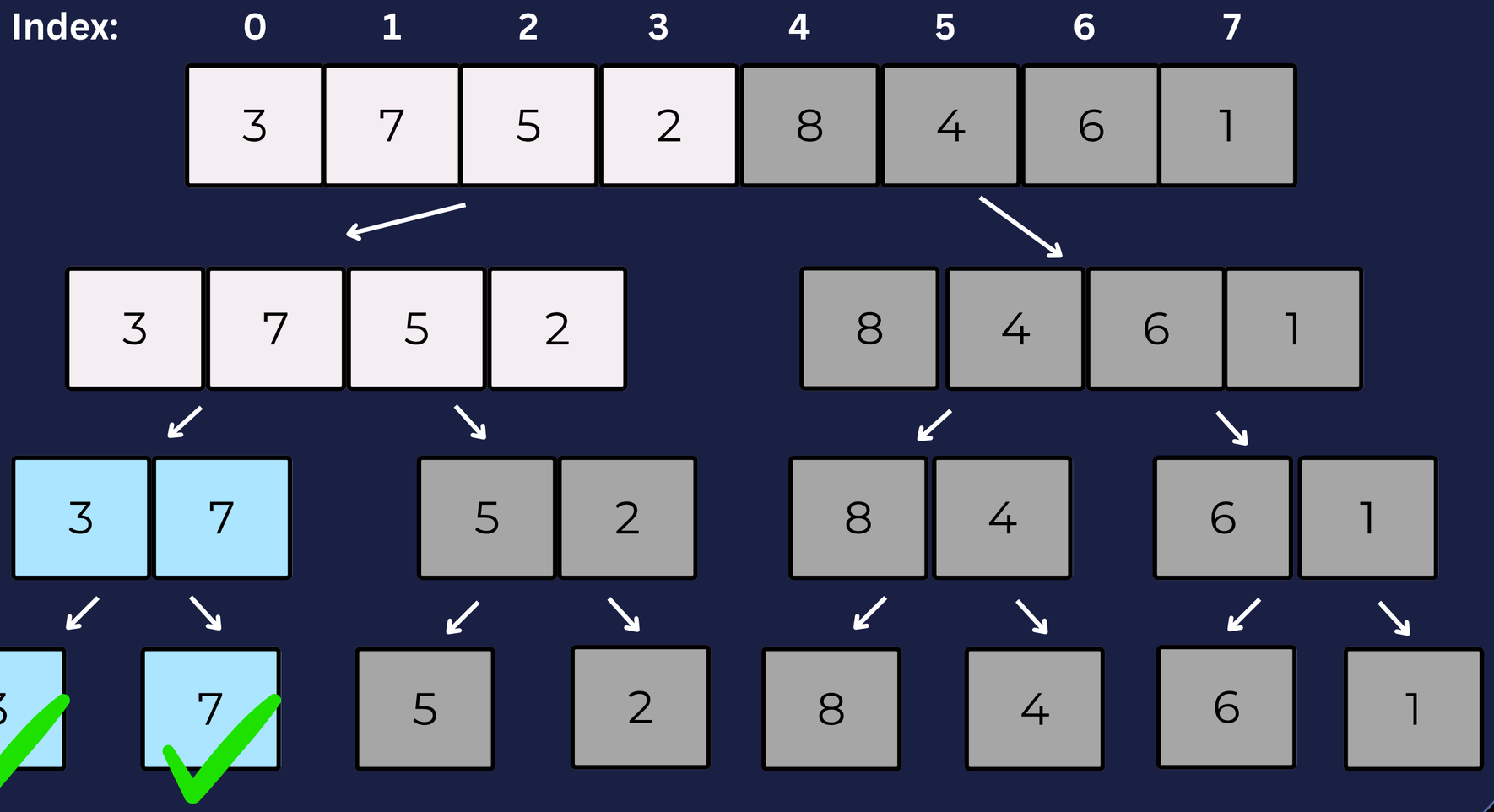

And then (recursively) split those arrays in half
Index:
0
1
2
3
4
5
6
7
3
7
5
2
8
4
6
1
3
7
5
2
8
4
6
1
3
7
5
2
8
4
6
1


And it keeps splitting the arrays until there is only one element in each section
Index:
0
1
2
3
4
5
6
7
3
7
5
2
8
4
6
1

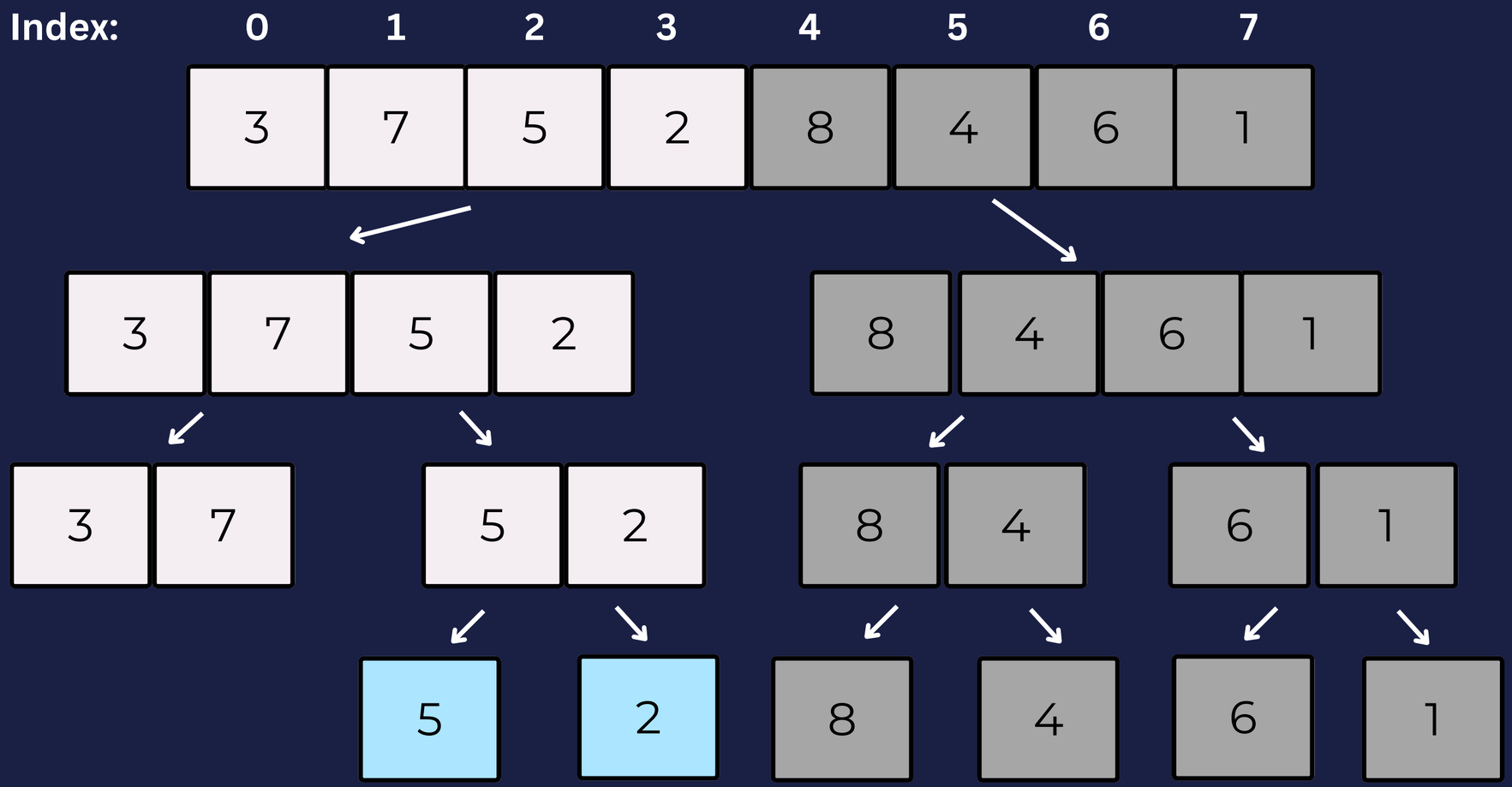
We sort backwards from the bottom and start with one branch at a time



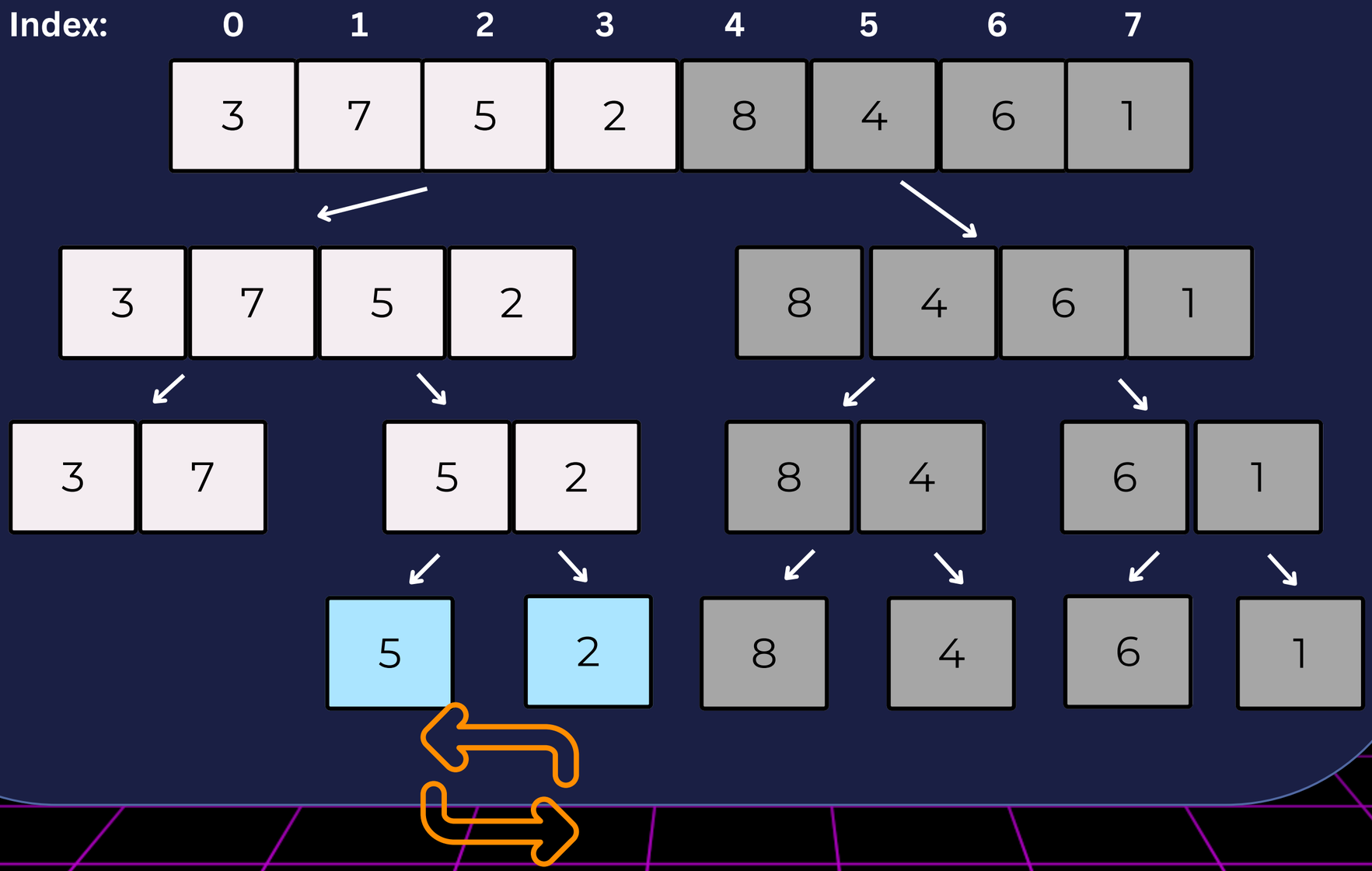
We check the first pair, 3 and 7 are in the correct order!



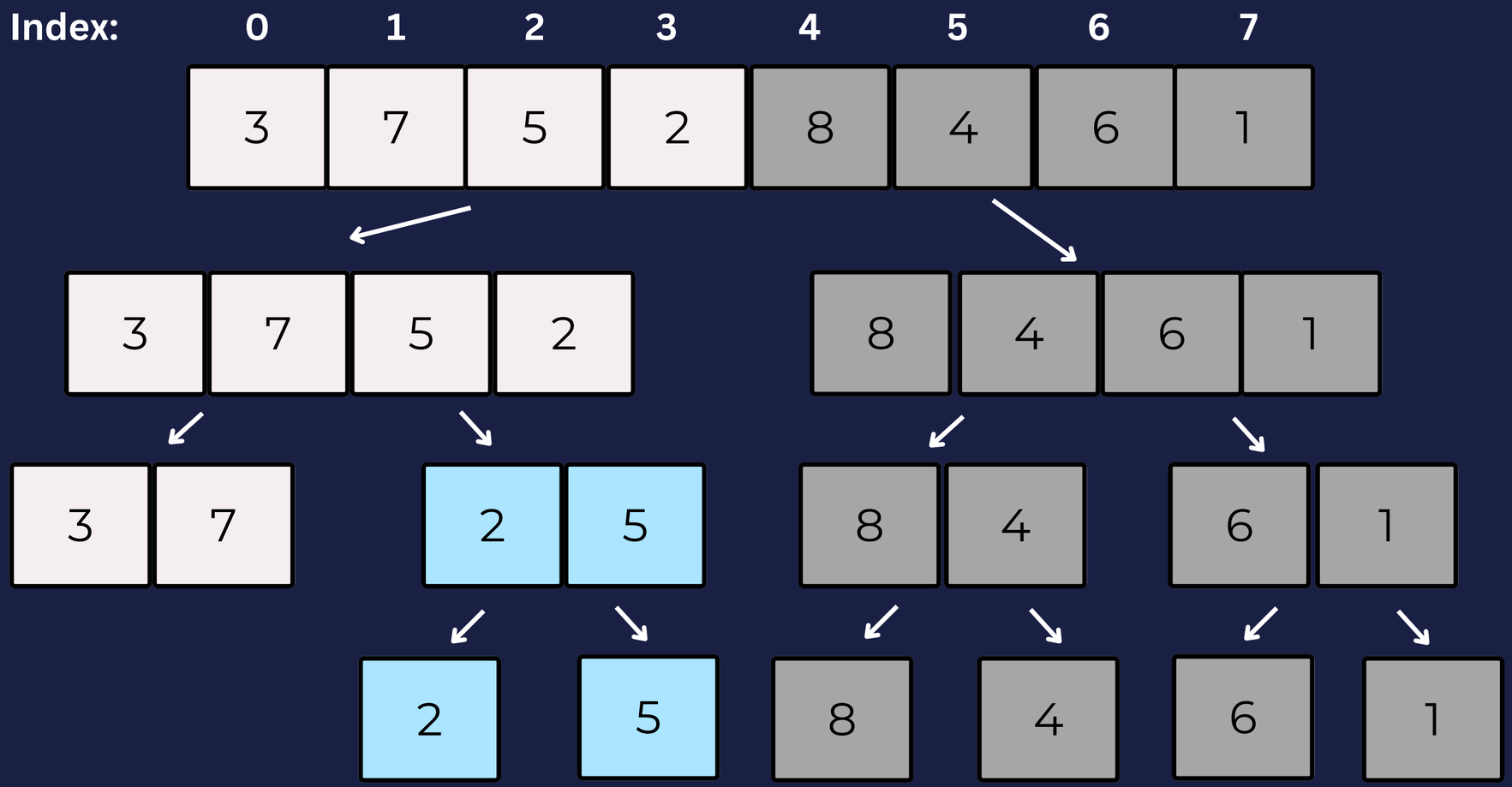
Now we need to check the next branch before we move up!



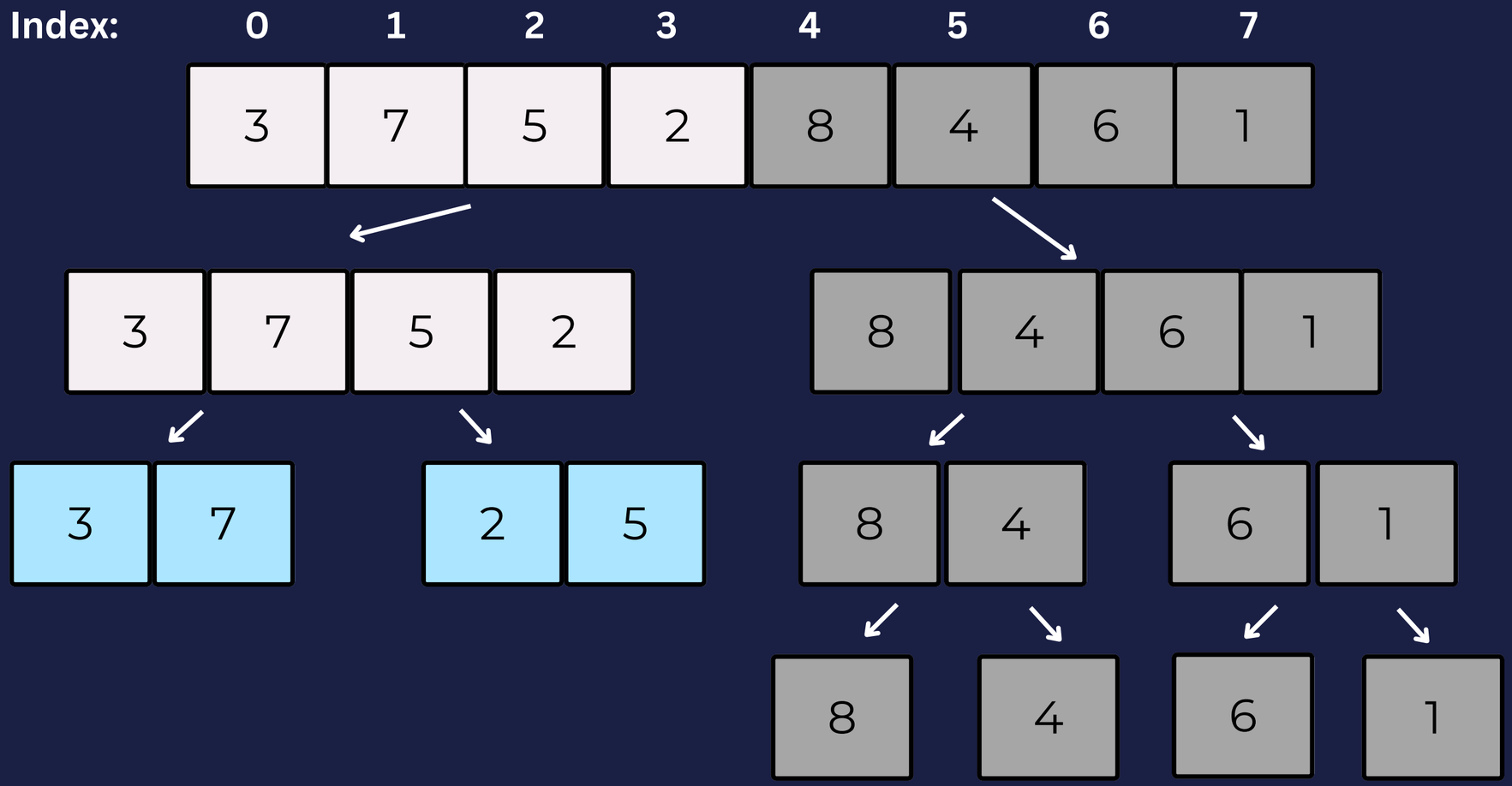
5 and 2 are not in the right order, so we need to swap them!

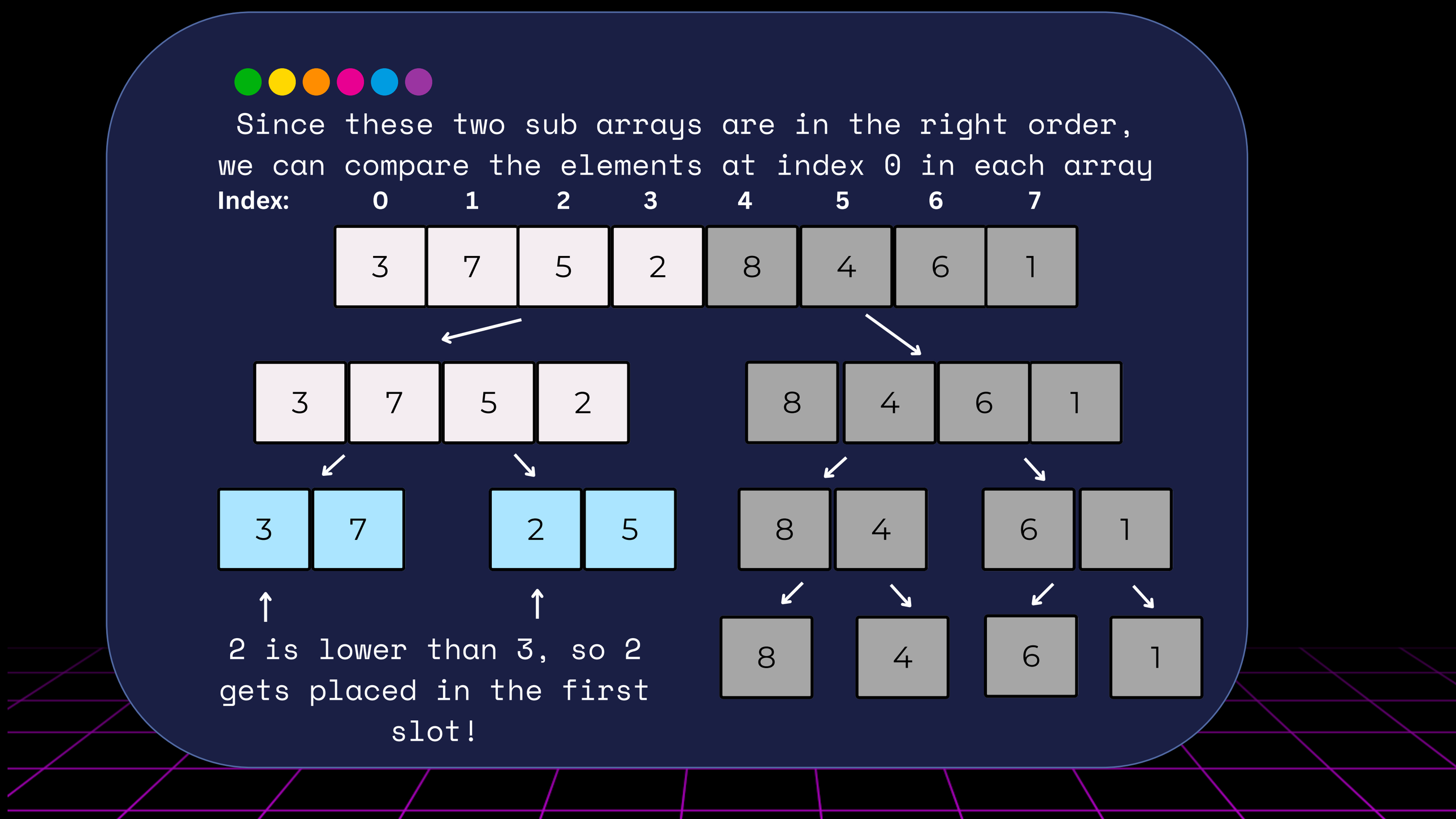


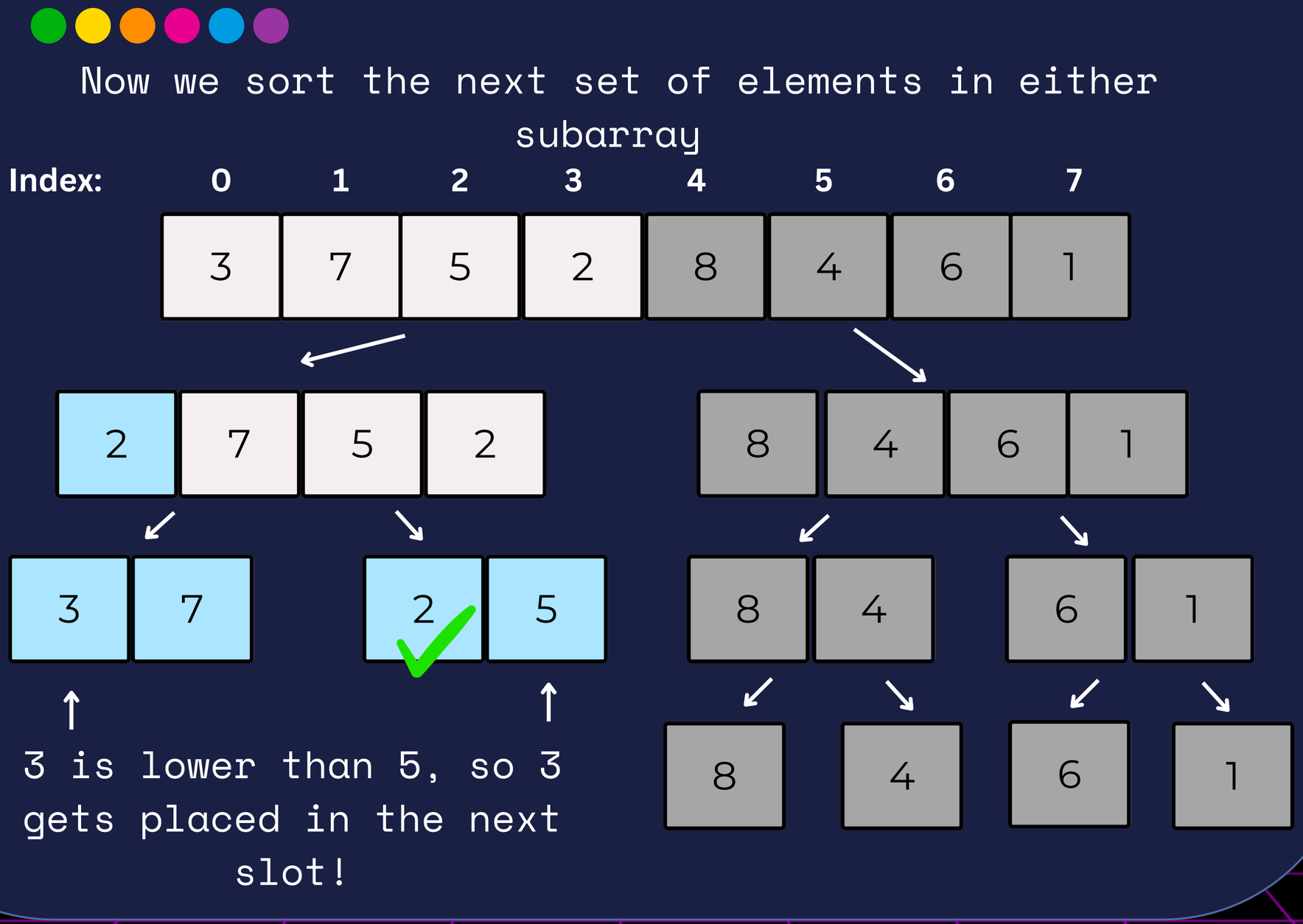
Now that they are in the right order we can move onto the higher level



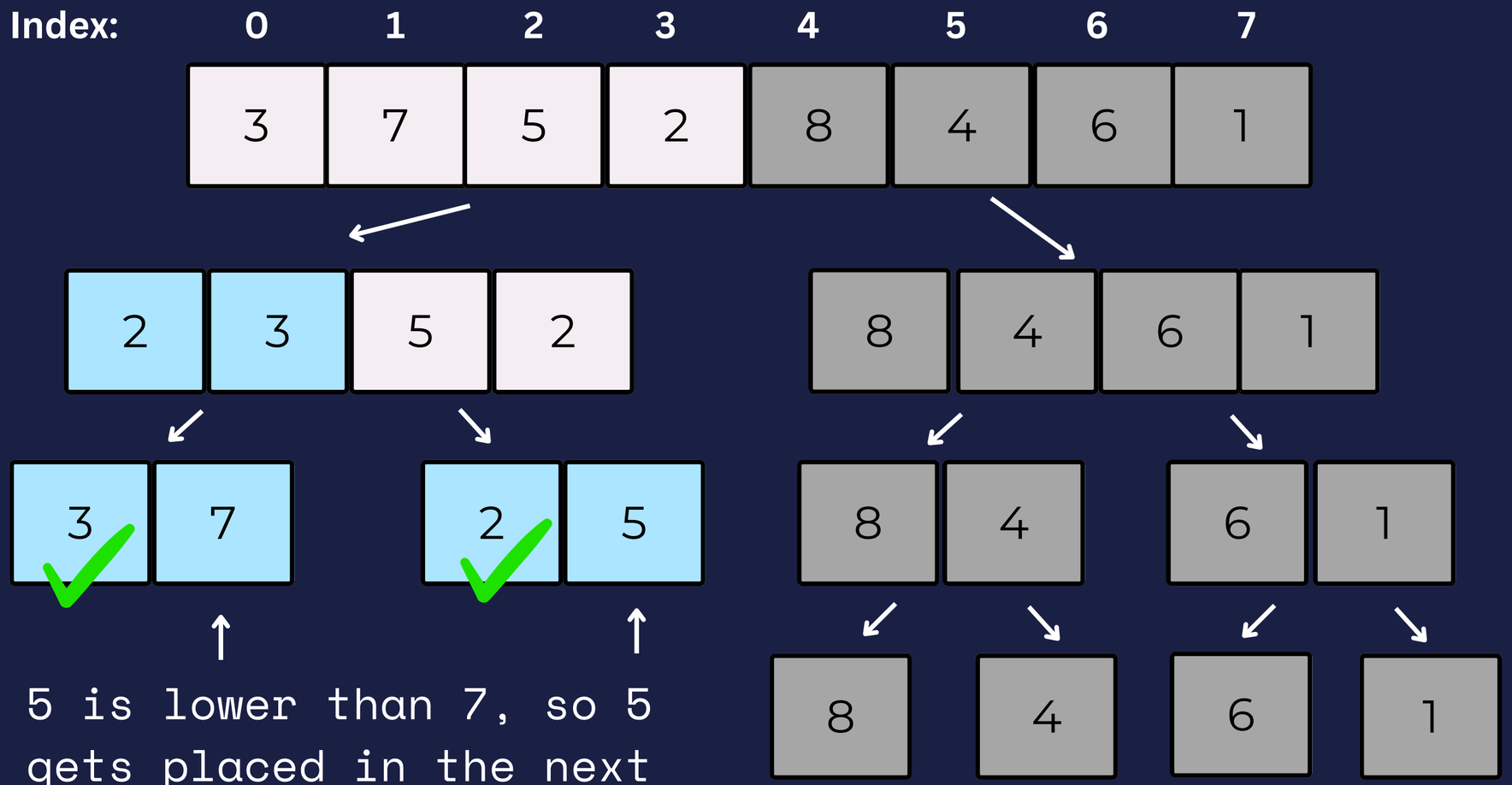
Since these two sub arrays are in the right order, we can compare the elements at index 0 in each array



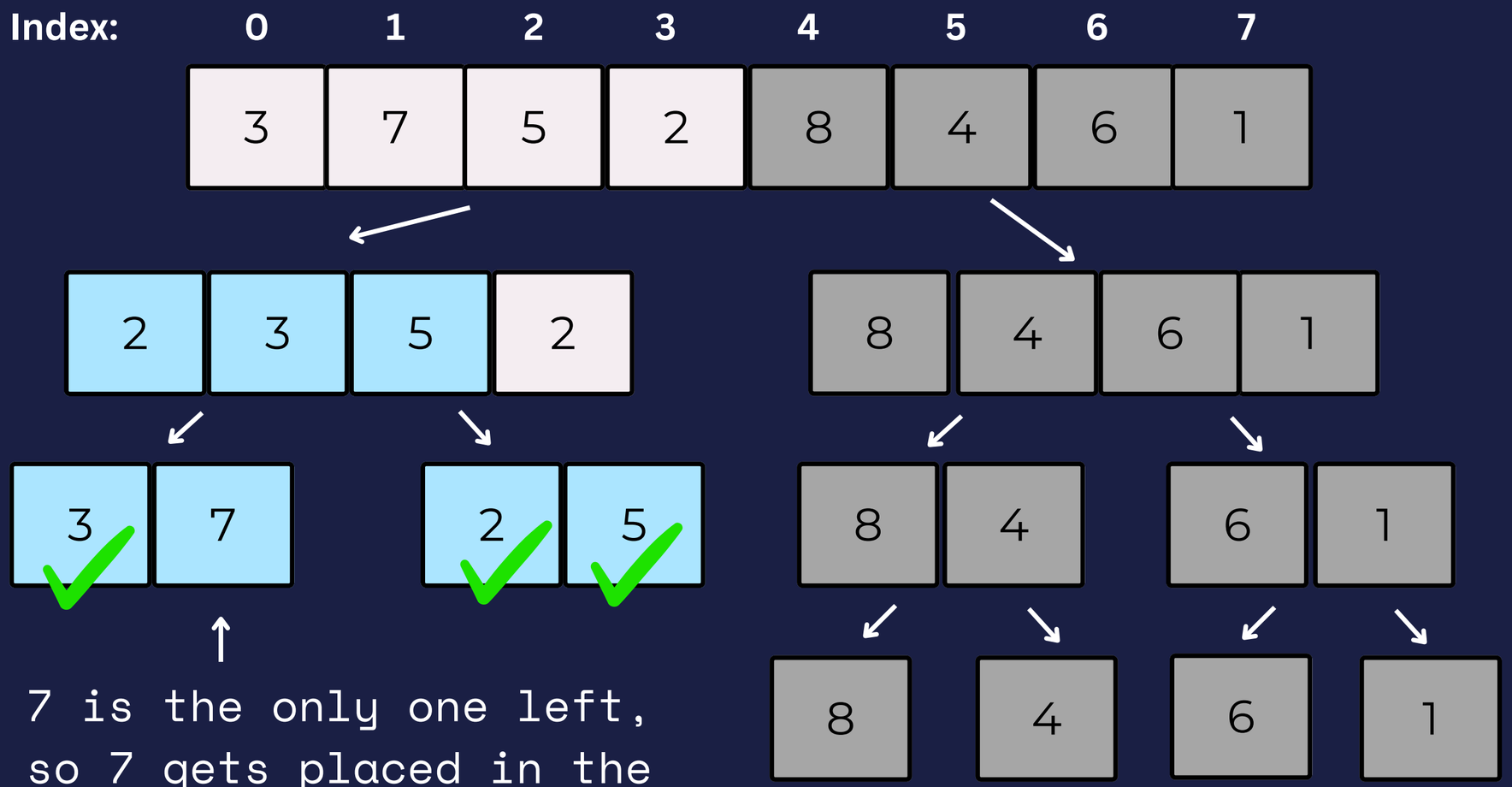

Since these two sub arrays are in the right order, we can compare the elements at index 0 in each array
Index: 0 1 2 3 4 5 6 7
3 7 5 2 8 4 6 1
3 7 5 2
8 4 6 1
3 7
2 5
8 4
6 1
8
4
6
1
2 is lower than 3, so 2 gets placed in the first slot!


Now we sort the next set of elements in either subarray
Index:
0
1
2
3
4
5
6
7
3
7
5
2
8
4
6
1
2
7
5
2
8
4
6
1
3
7
2
5
8
4
6
1
8
4
6
1
3 is lower than 5, so 3 gets placed in the next slot!

Now we sort the next set of elements in either subarray

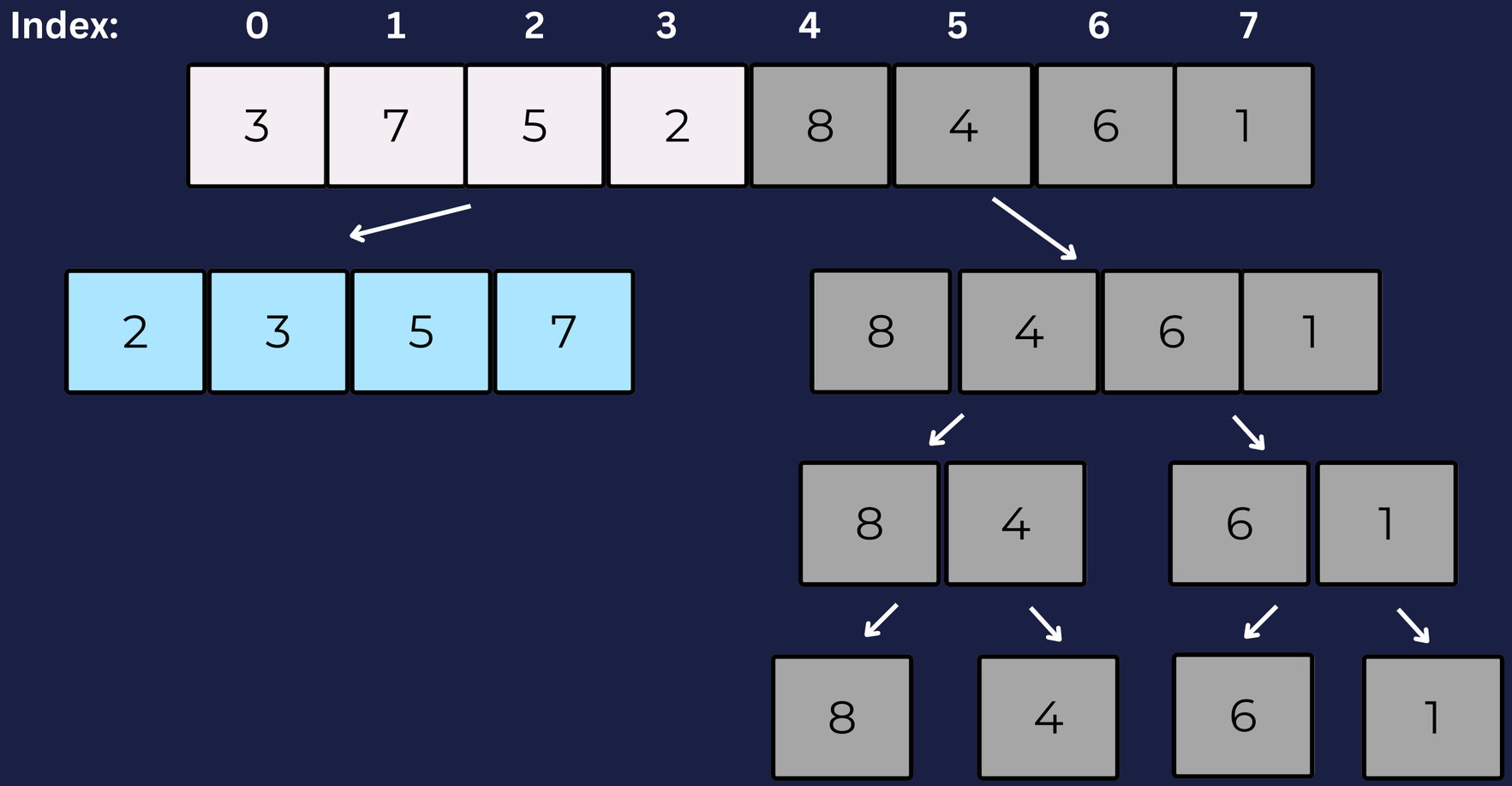


Now we sort the next set of elements in either subarray

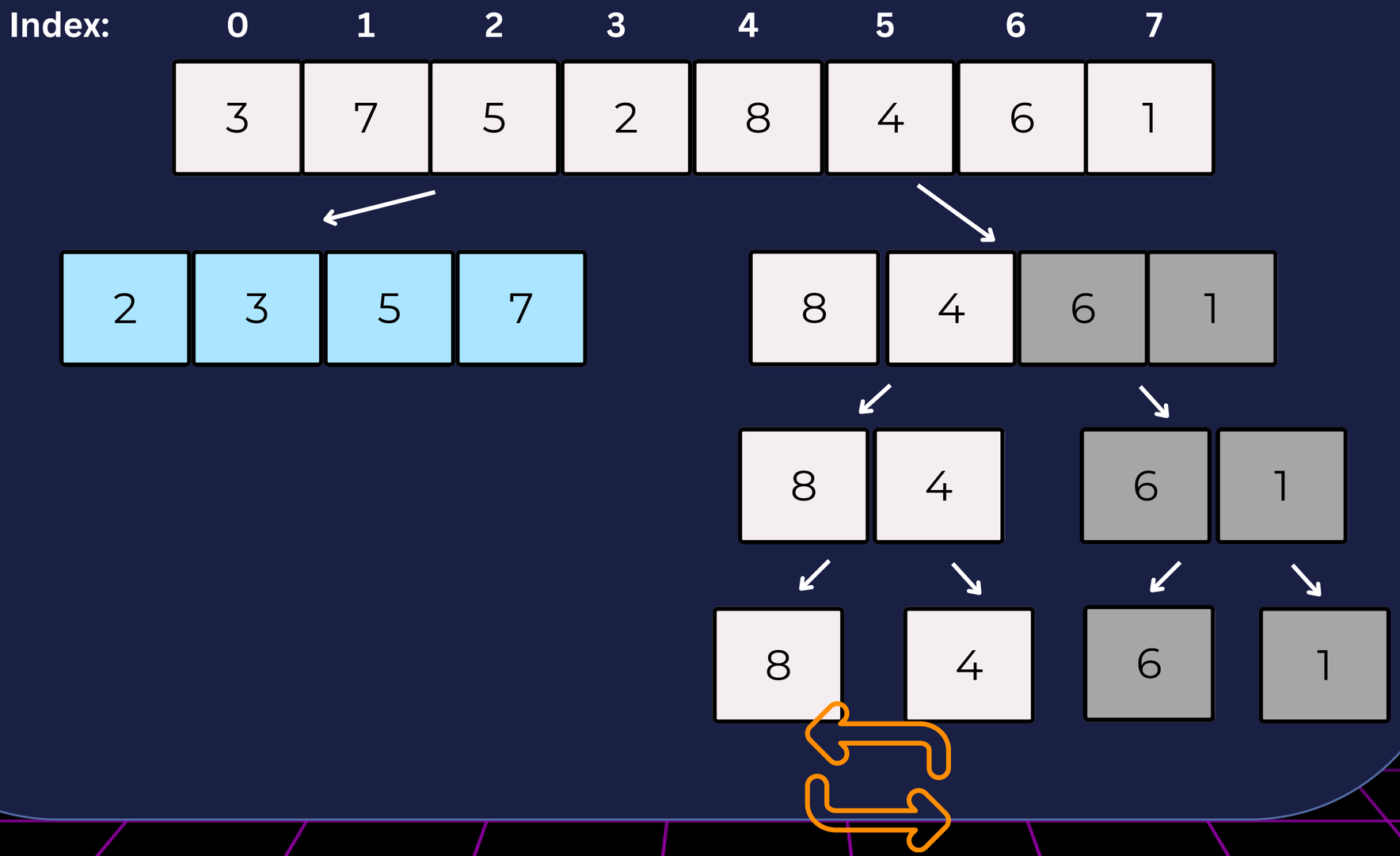


7 is the only one left, so 7 gets placed in the last spot

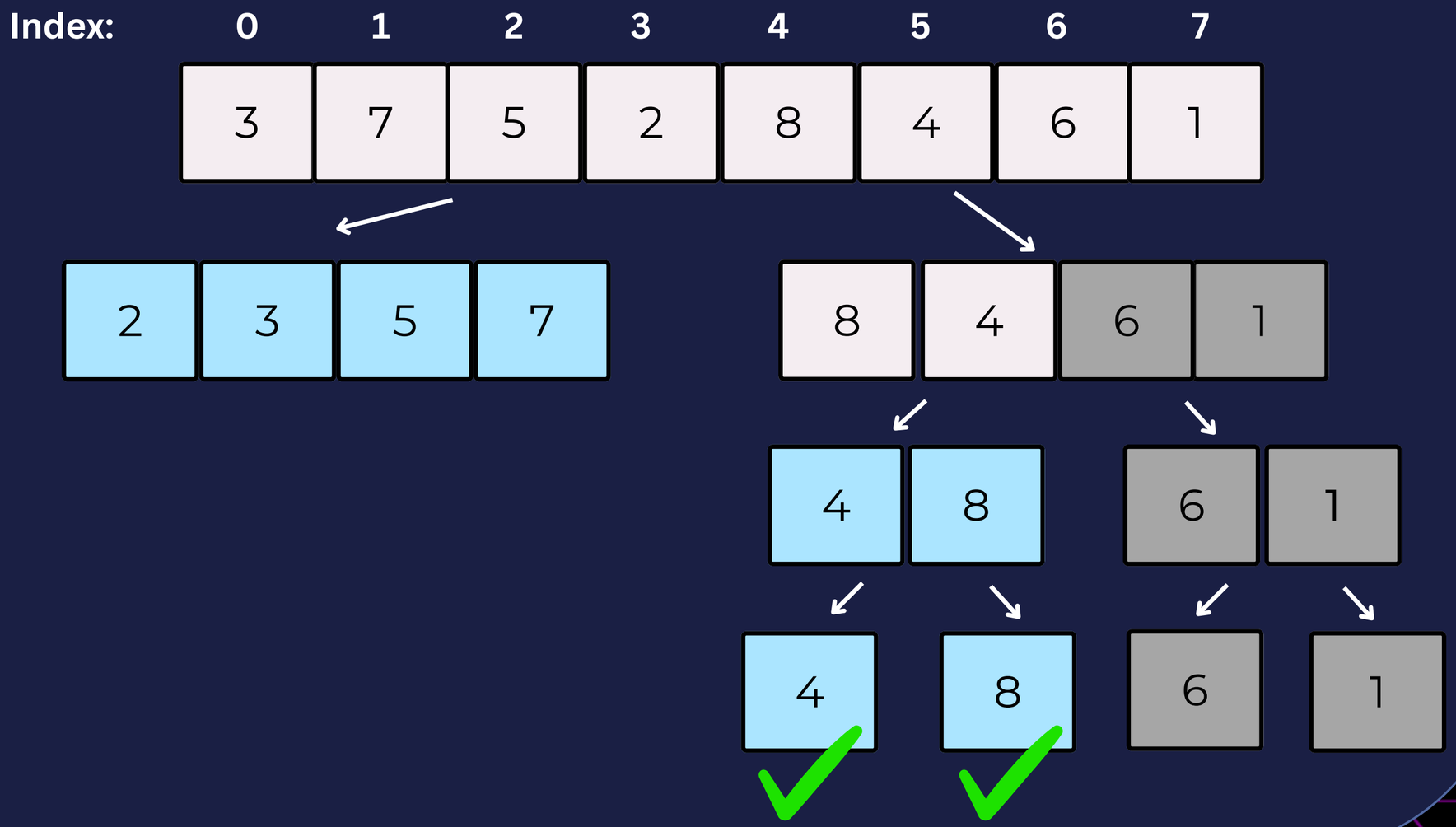
Since we can't move up without the other subarray,
we will work on the next branch



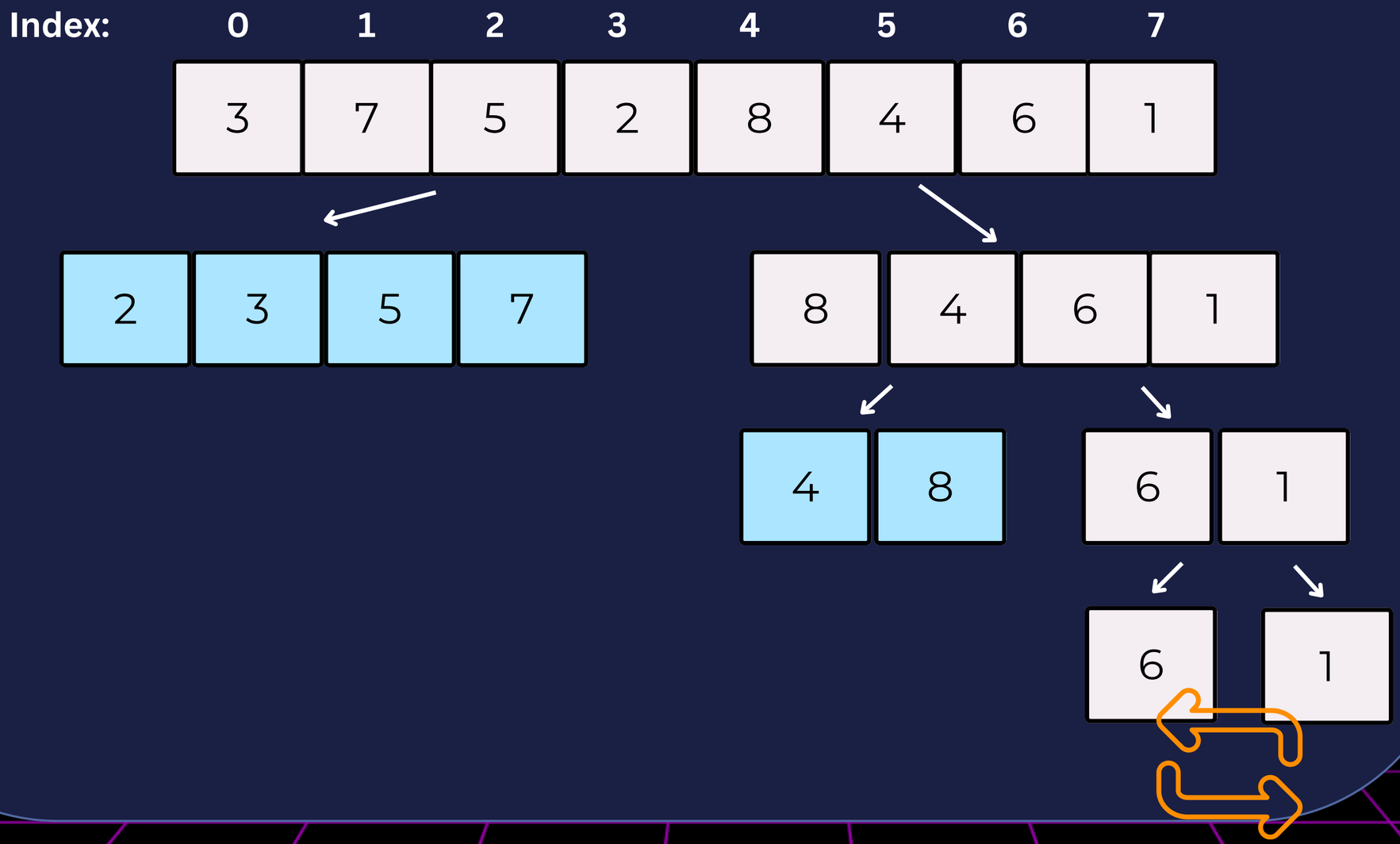
Lets go through this side quickly...



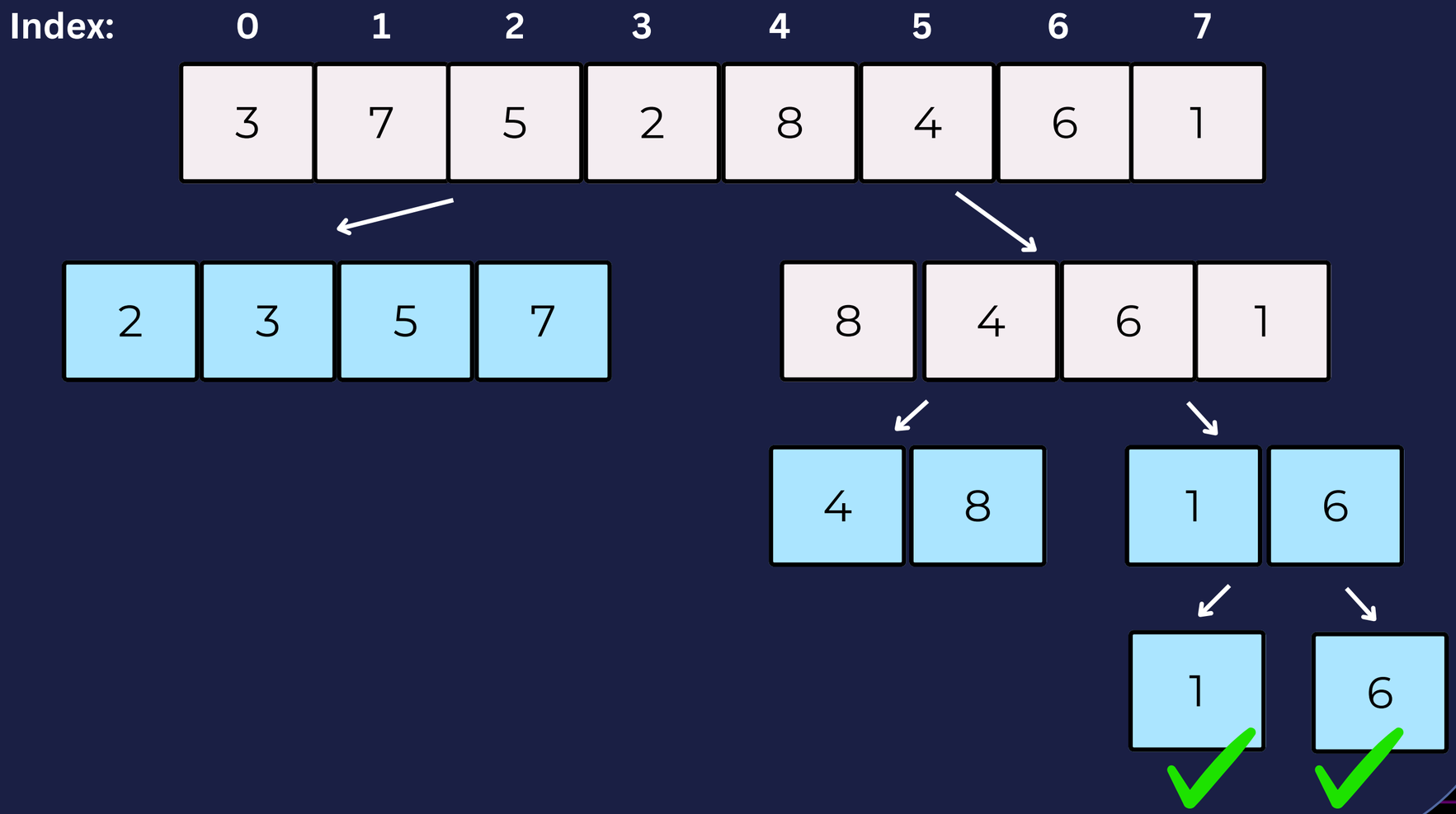
Lets go through this side quickly...



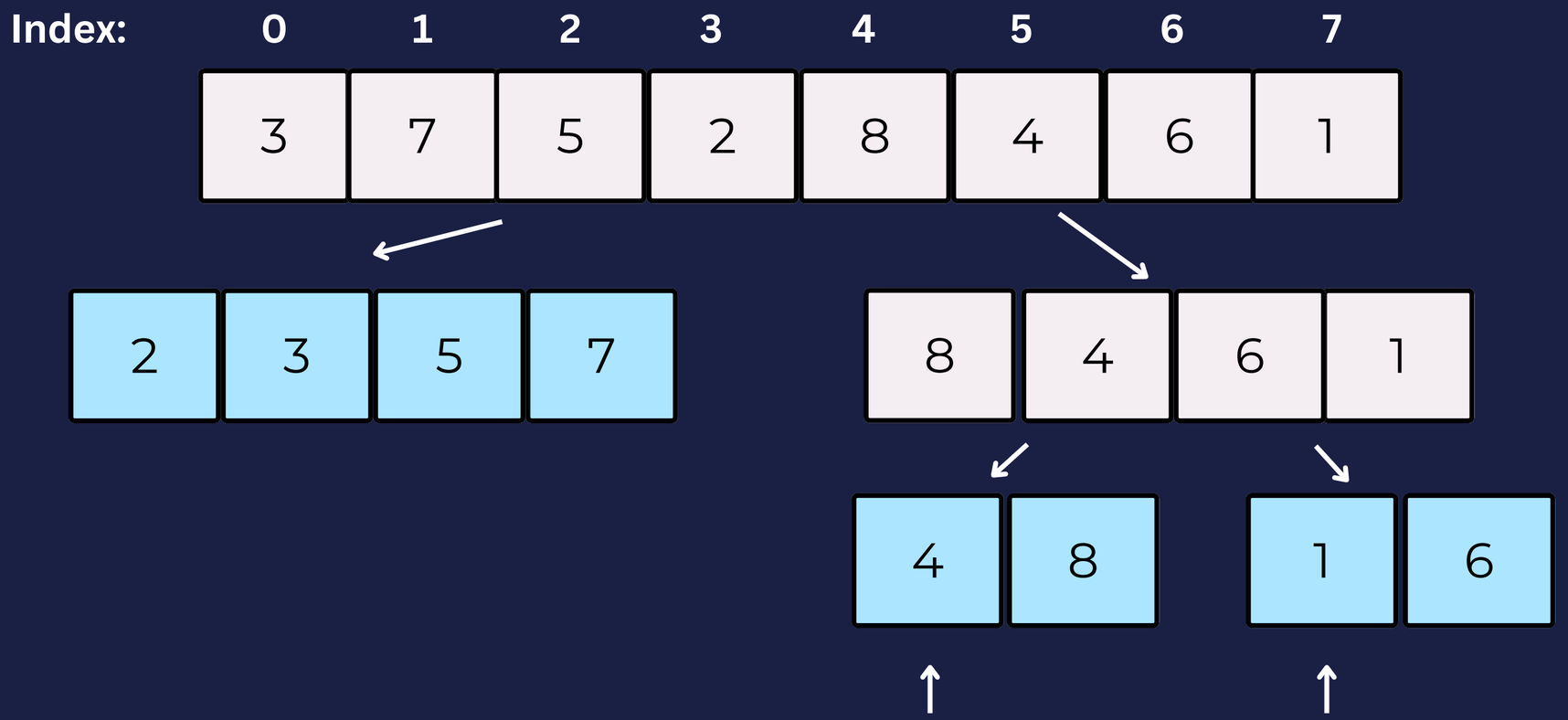
Lets go through this side quickly...



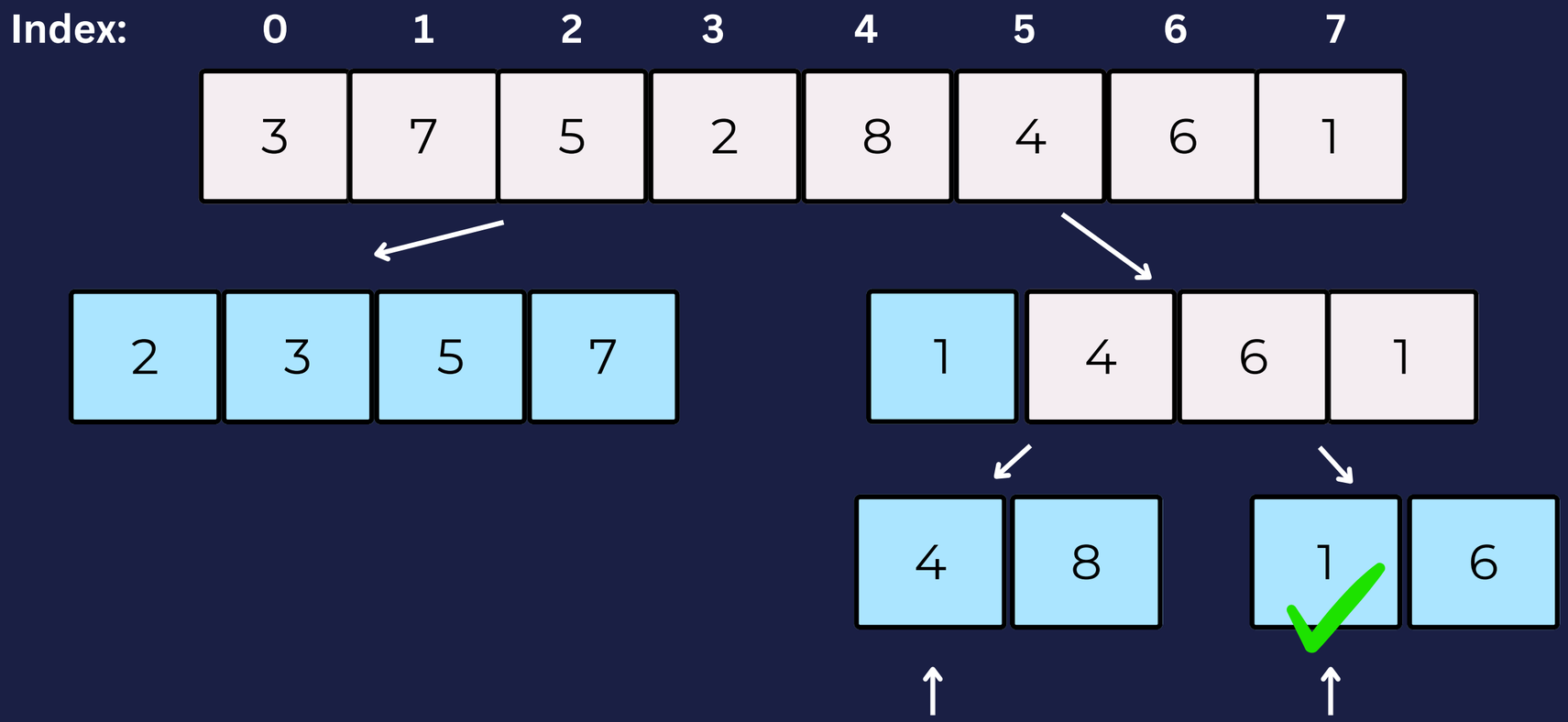
Lets go through this side quickly...



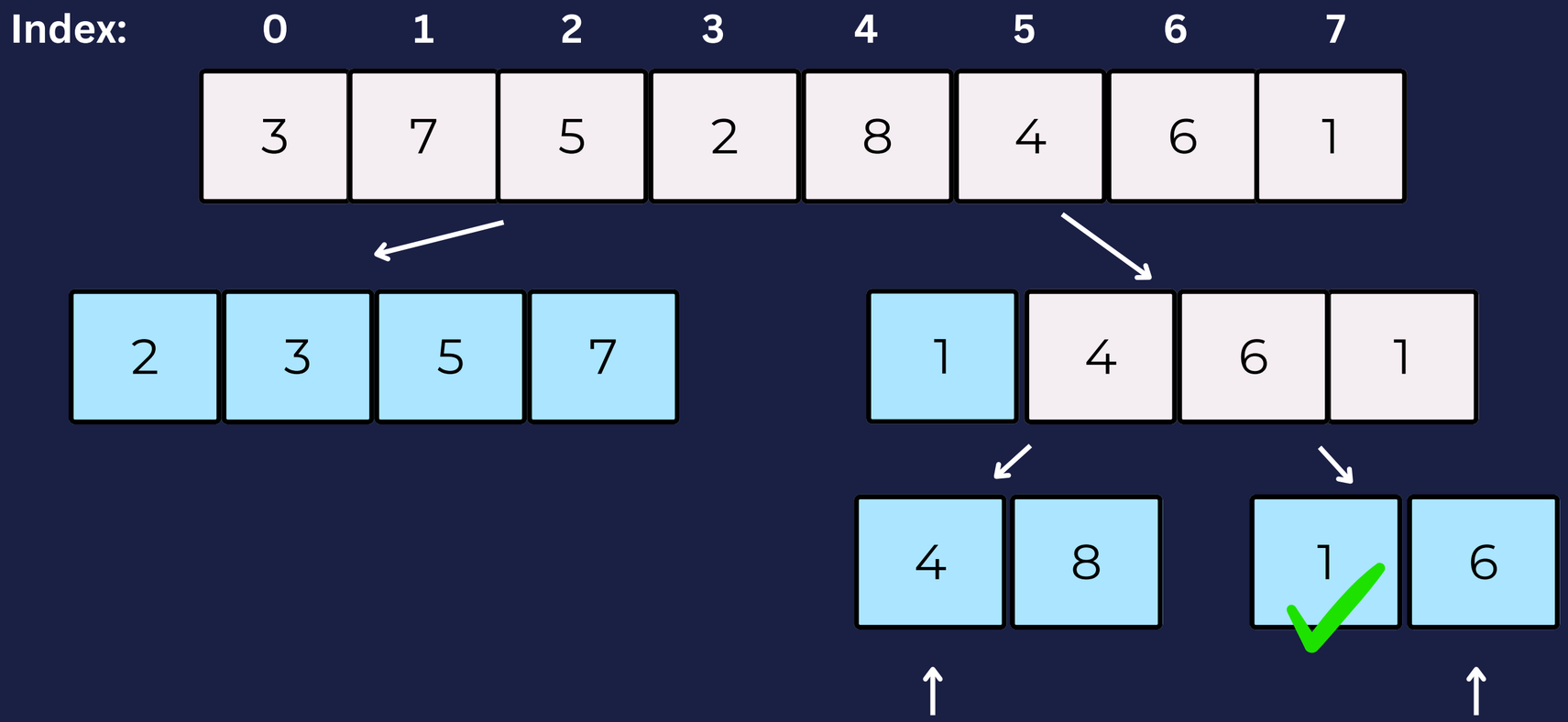
Lets go through this side quickly...



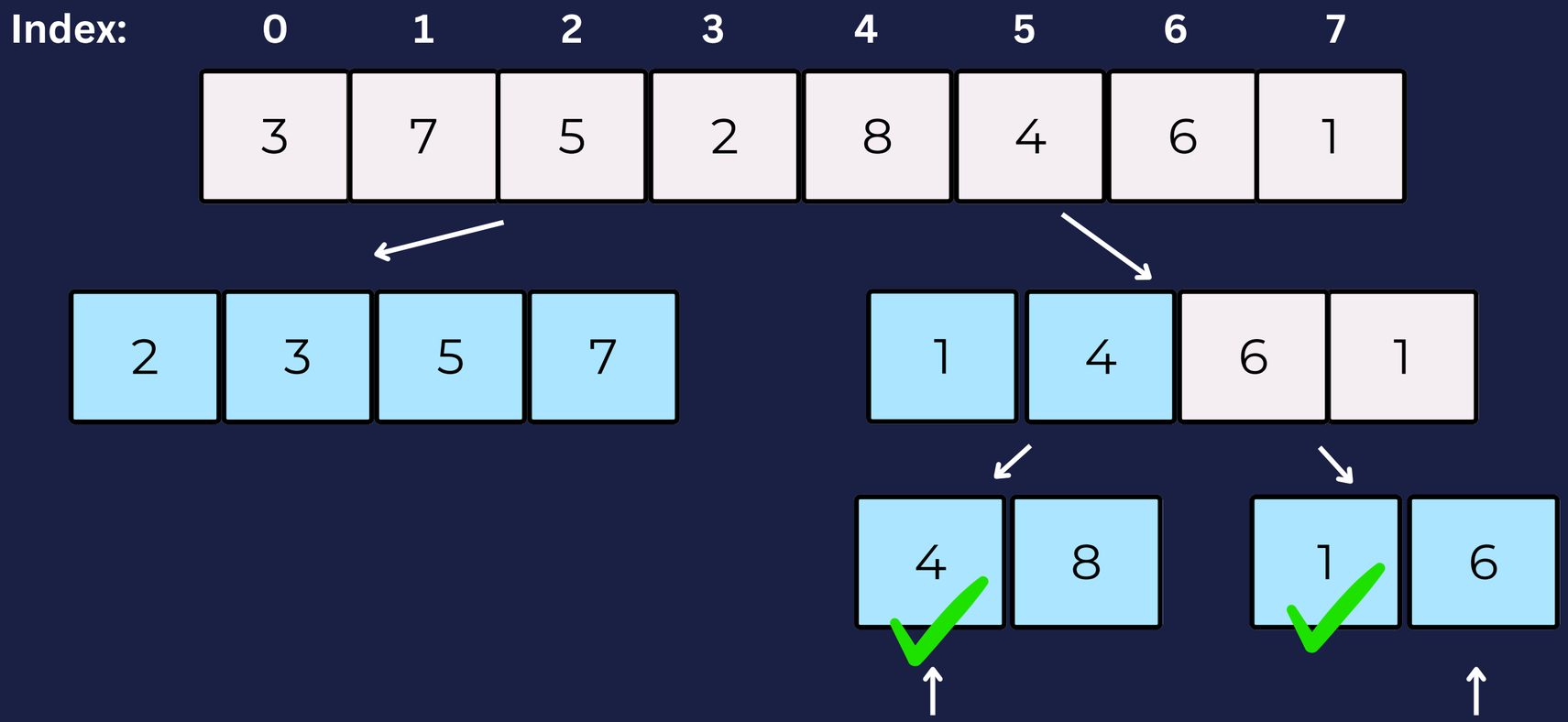
Lets go through this side quickly...



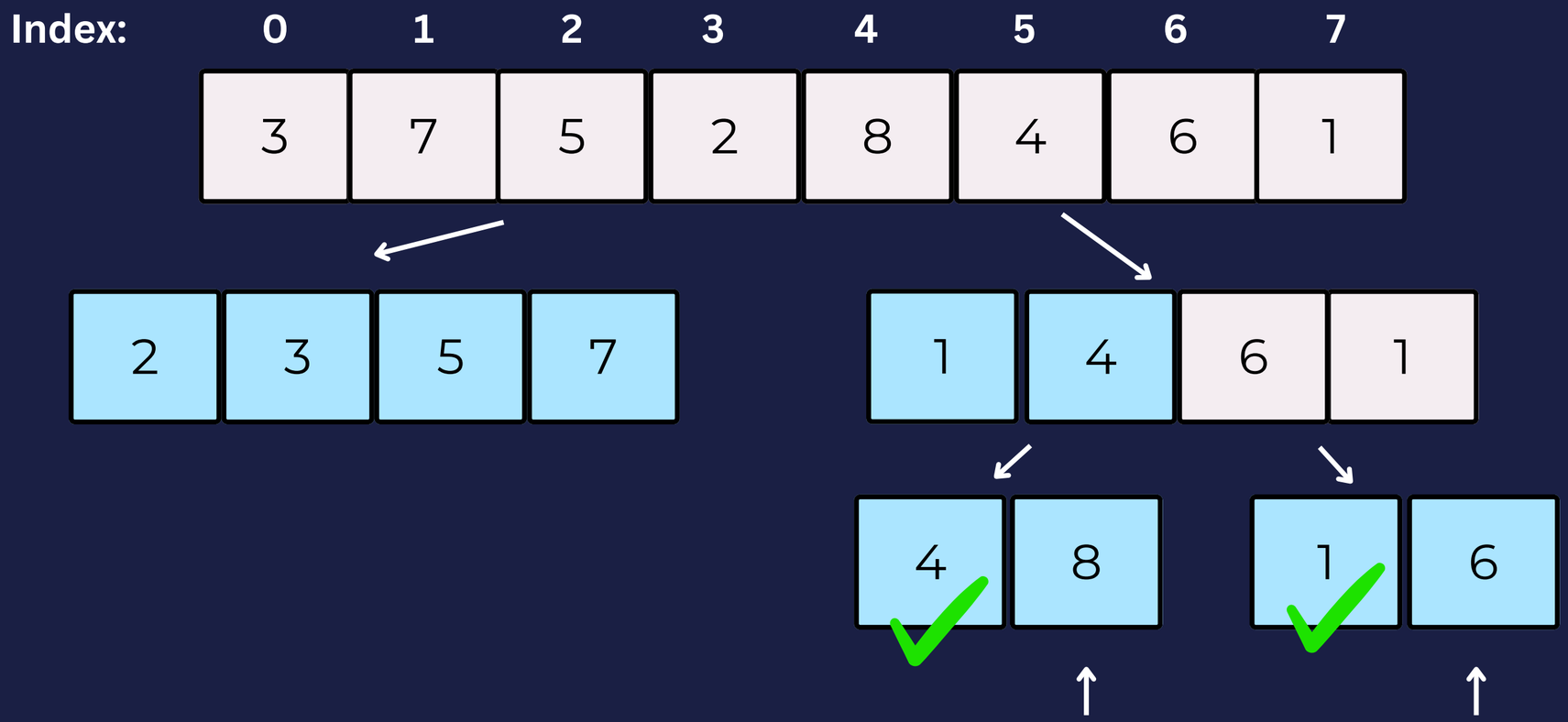
Lets go through this side quickly...



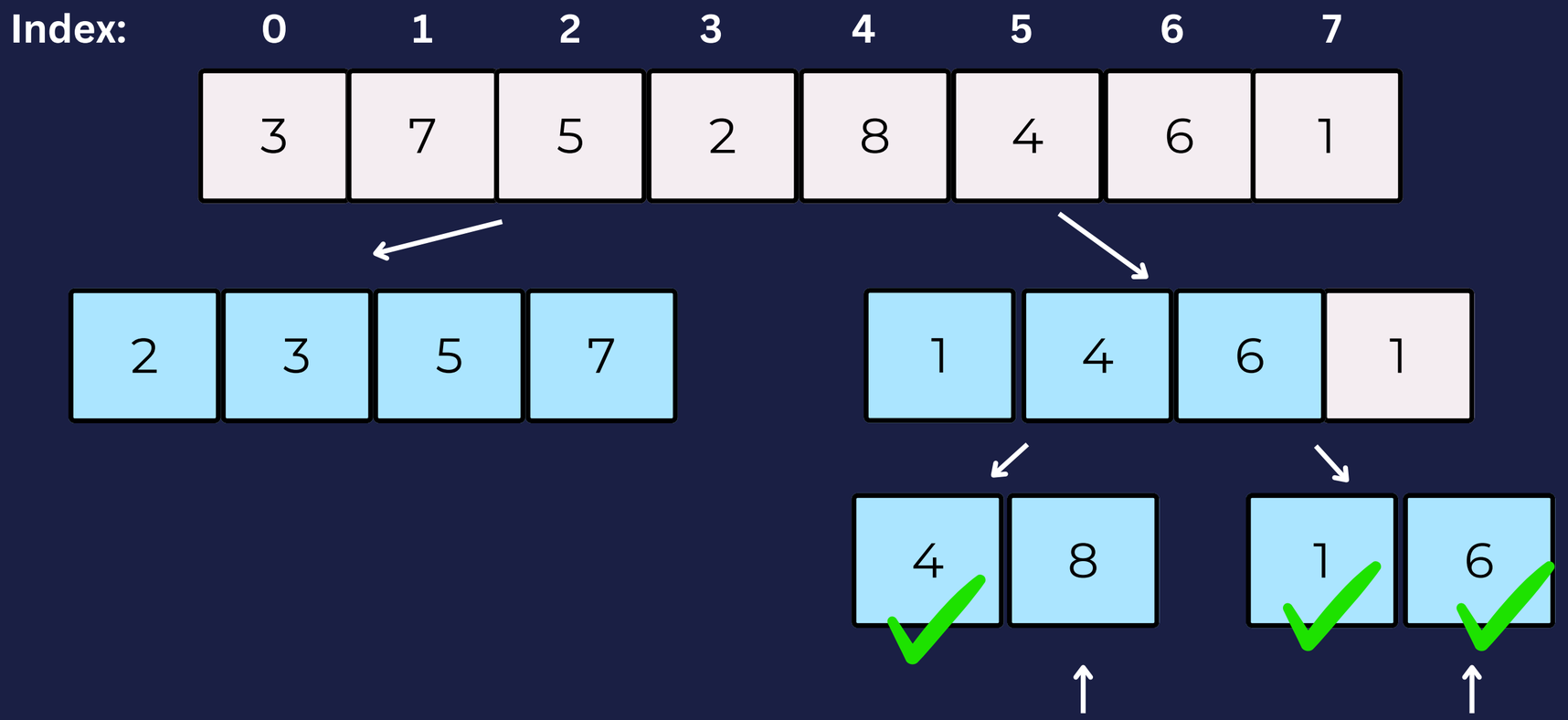
Lets go through this side quickly...



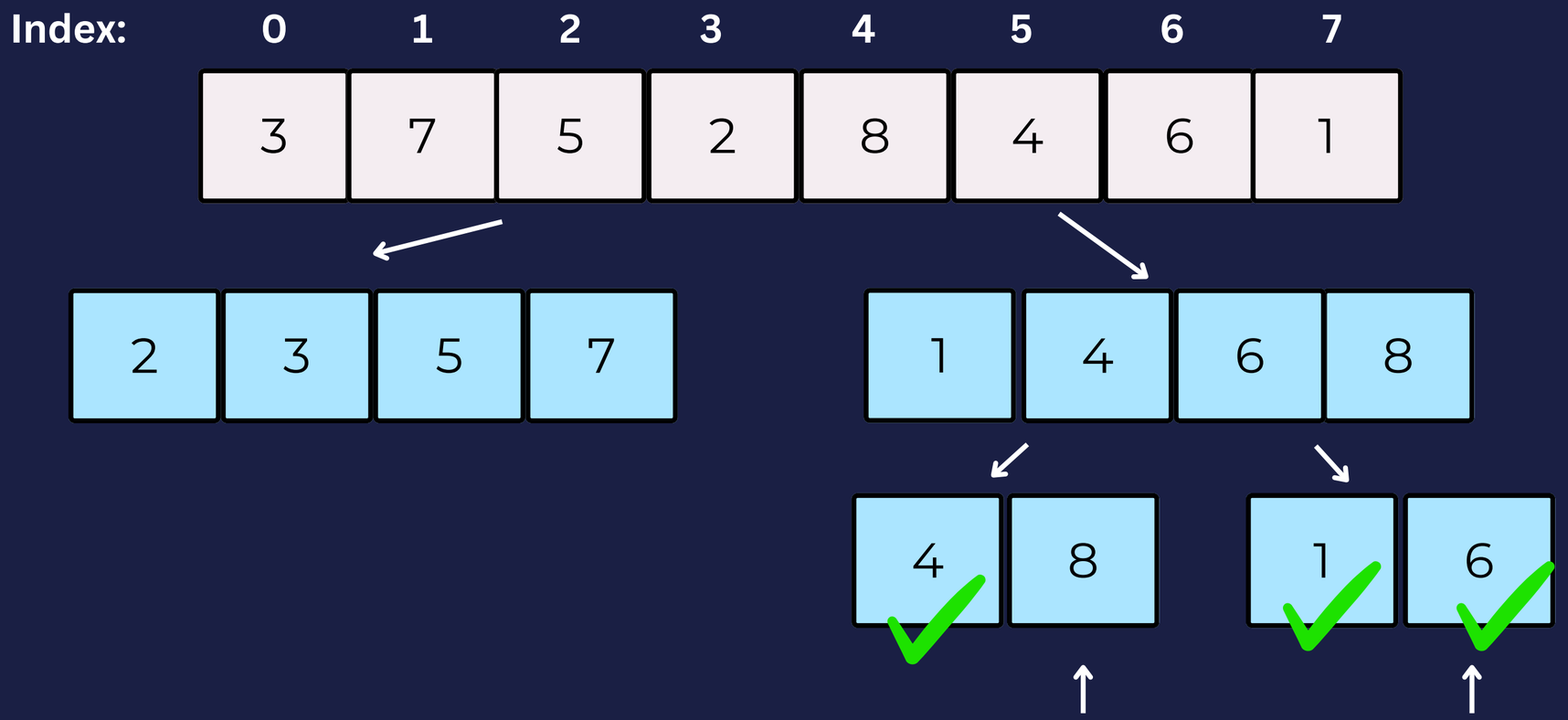
Lets go through this side quickly...



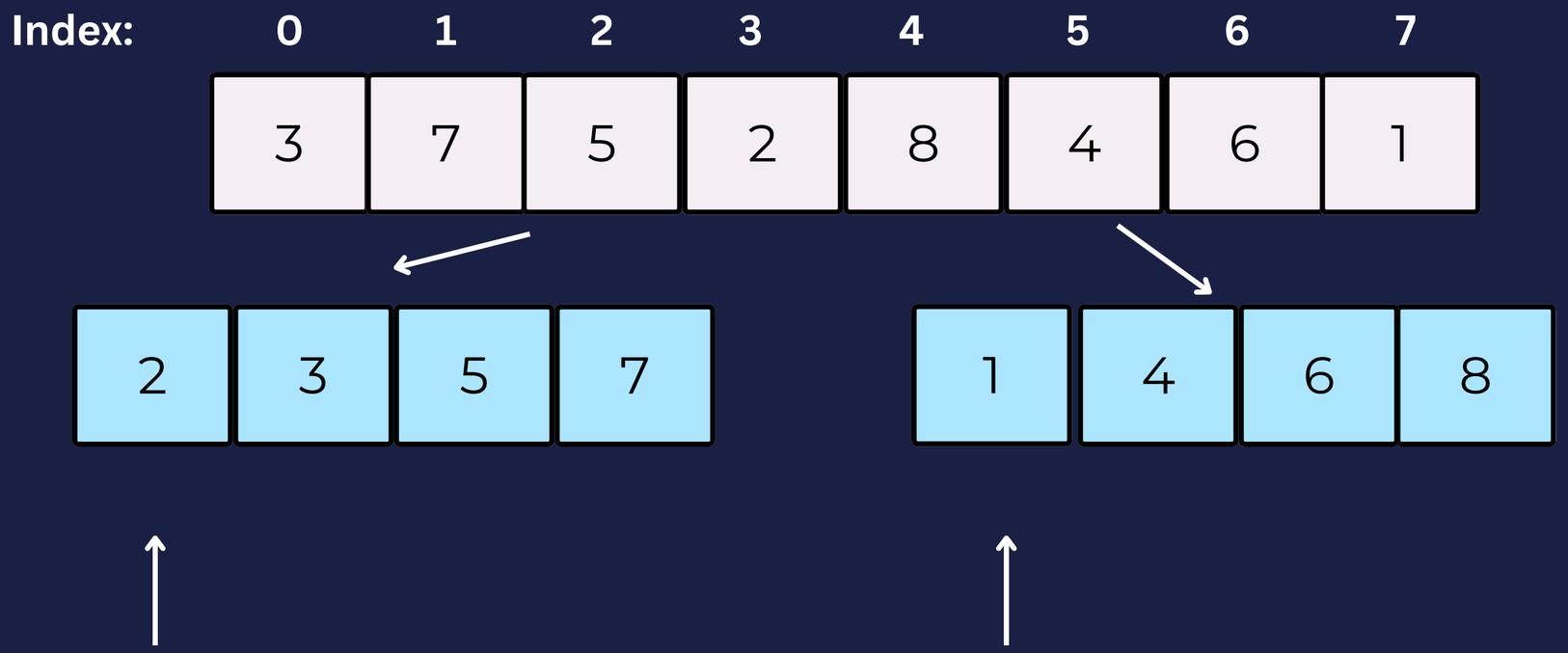
Lets go through this side quickly...



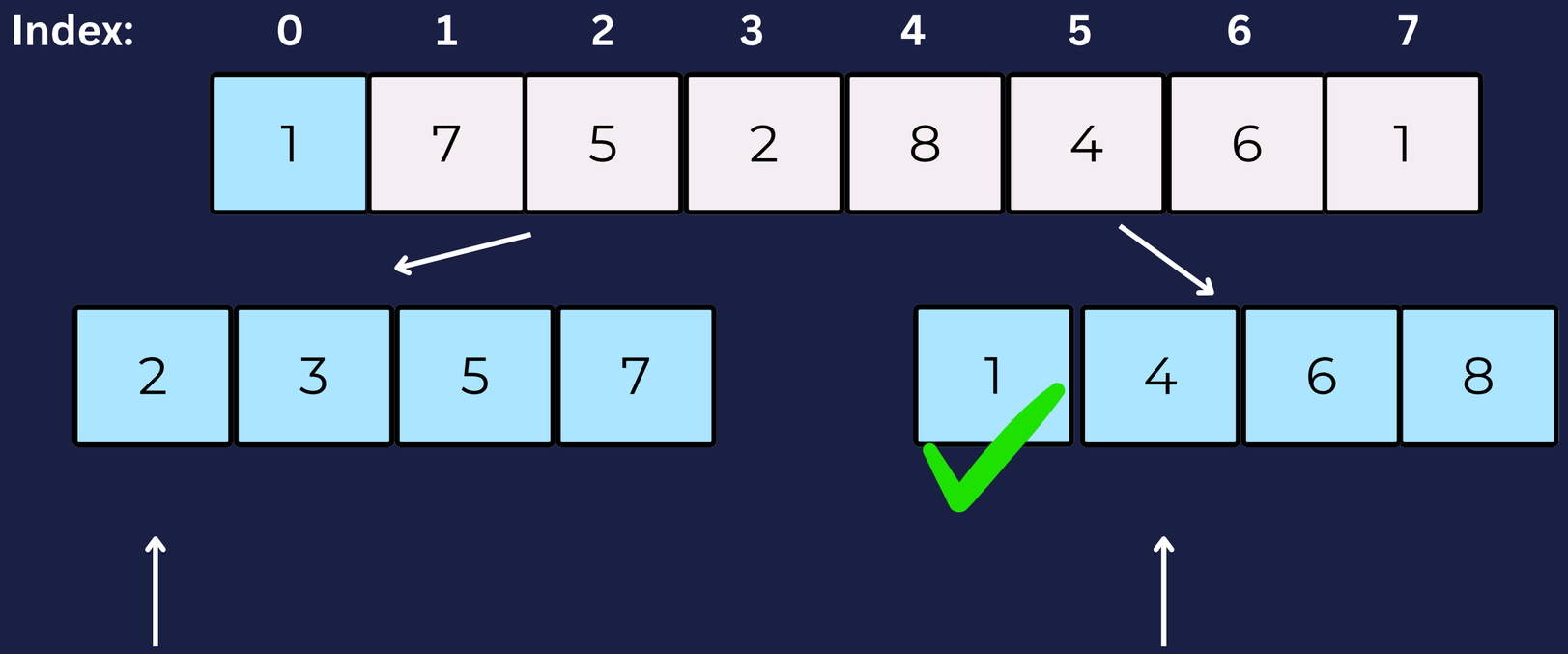
Lets go through this side quickly...



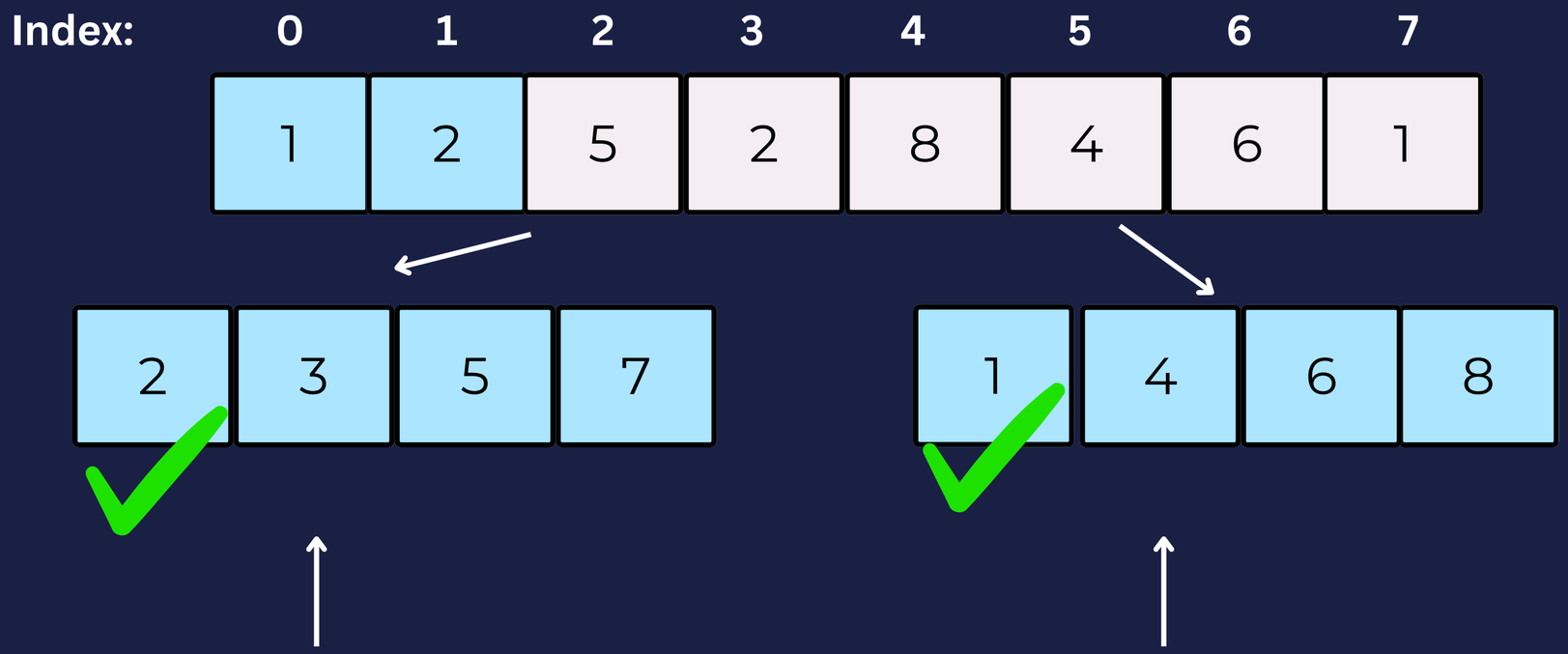
Lets go through this side quickly...



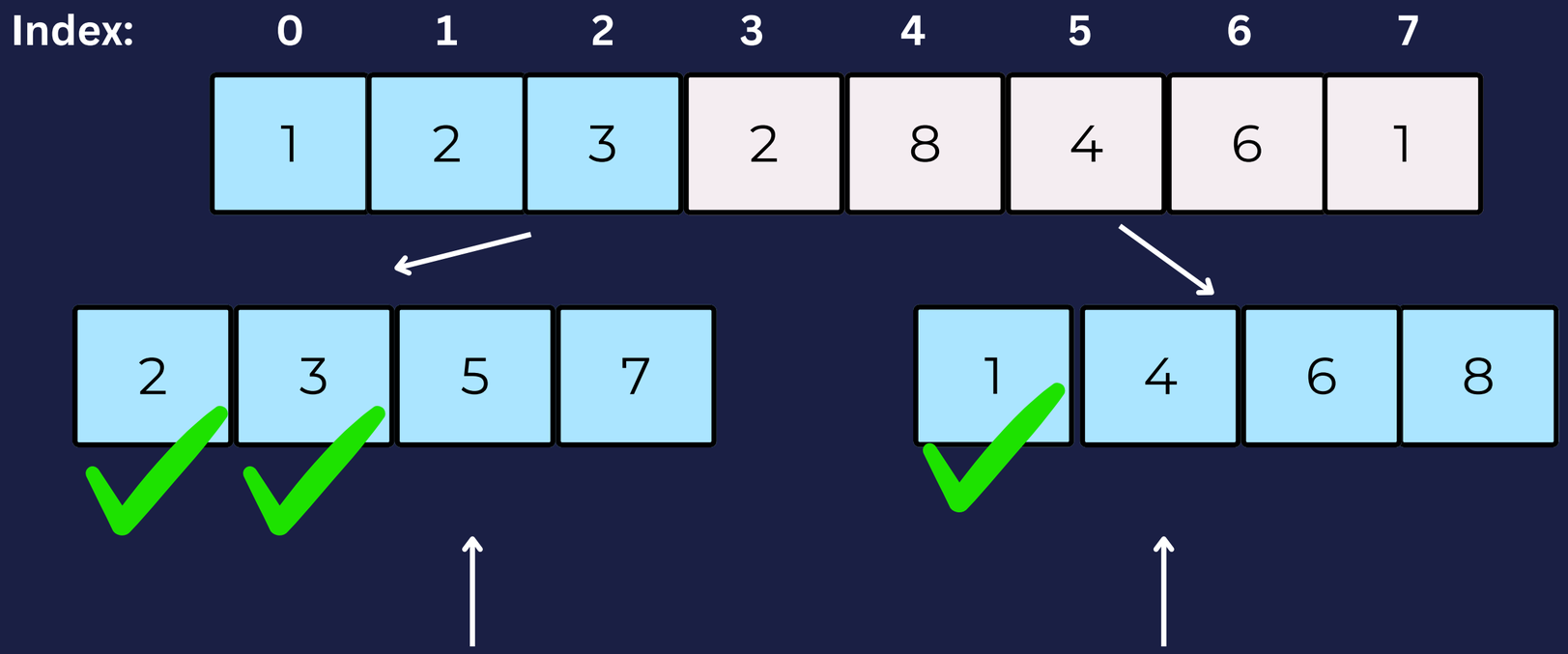
Lets go through this side quickly...

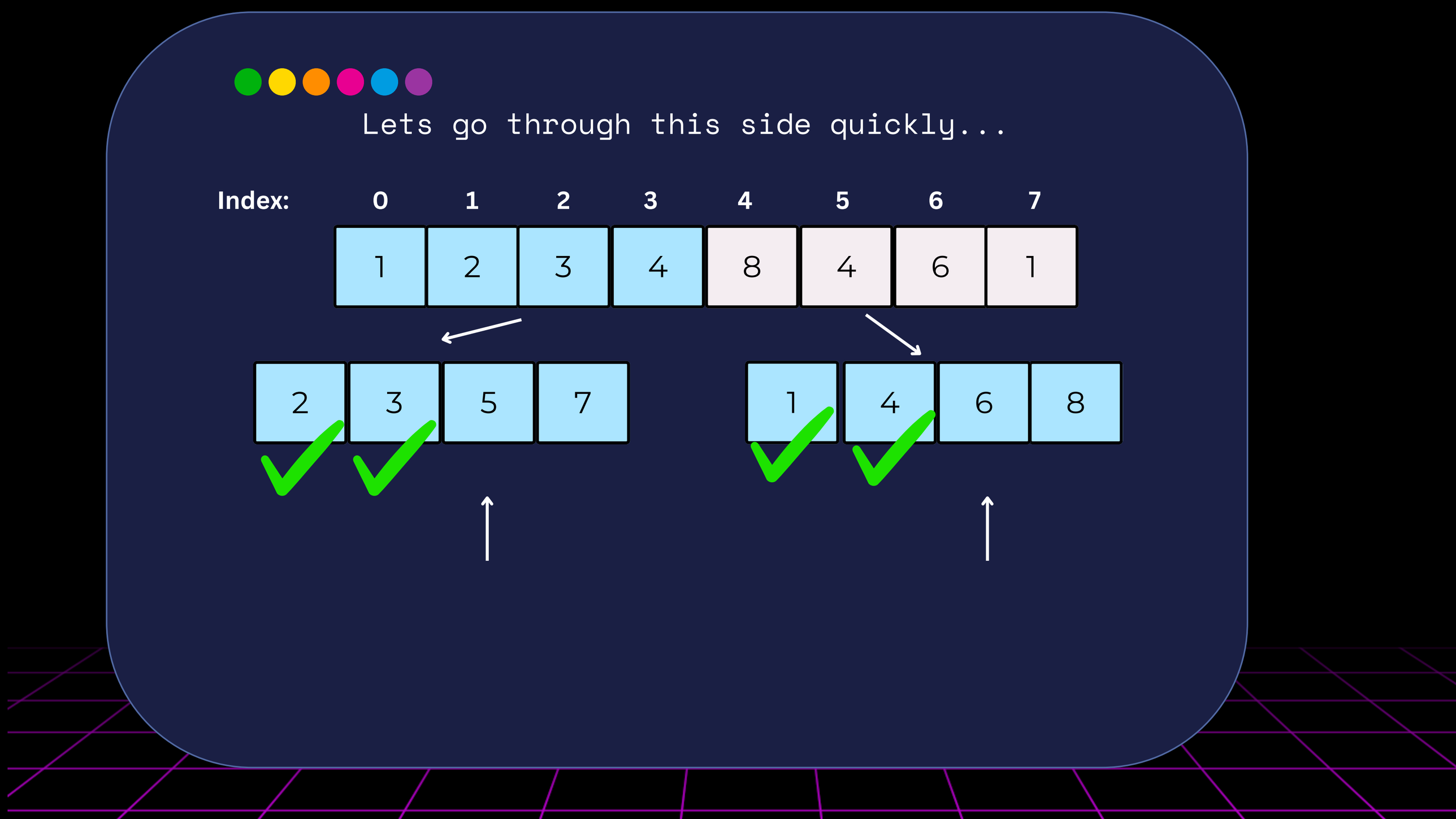


Lets go through this side quickly...

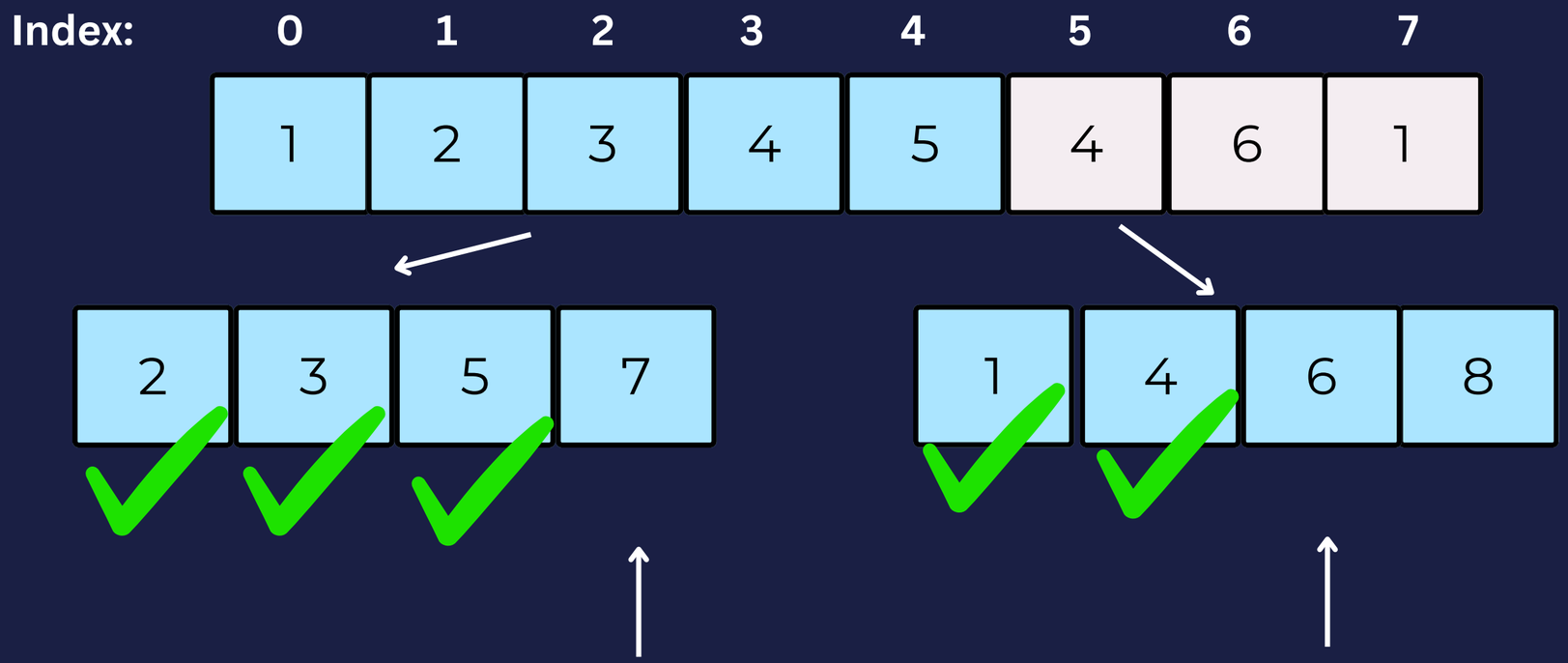


Lets go through this side quickly...

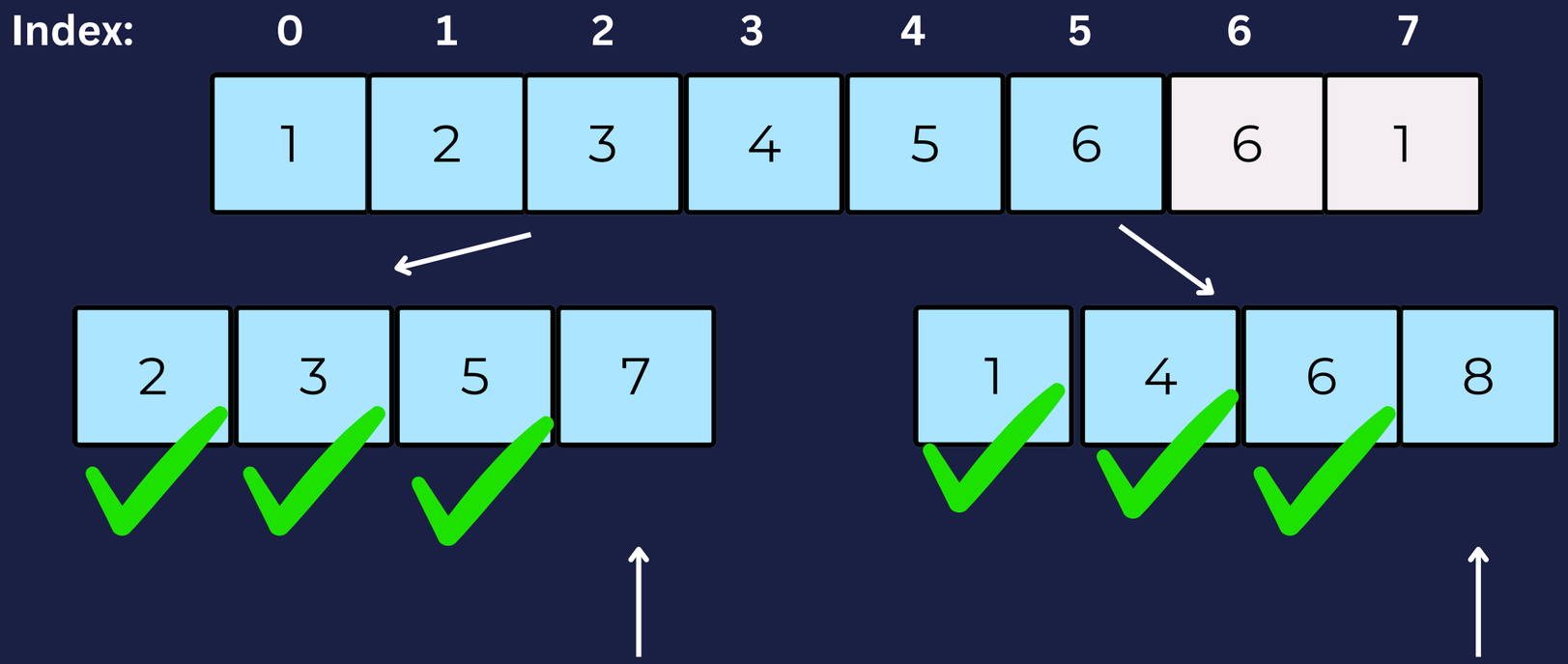



Lets go through this side quickly...
Index:
0
1
2
3
4
5
6
7
1
2
3
4
8
4
6
1
2
3
5
7
1
4
6
8

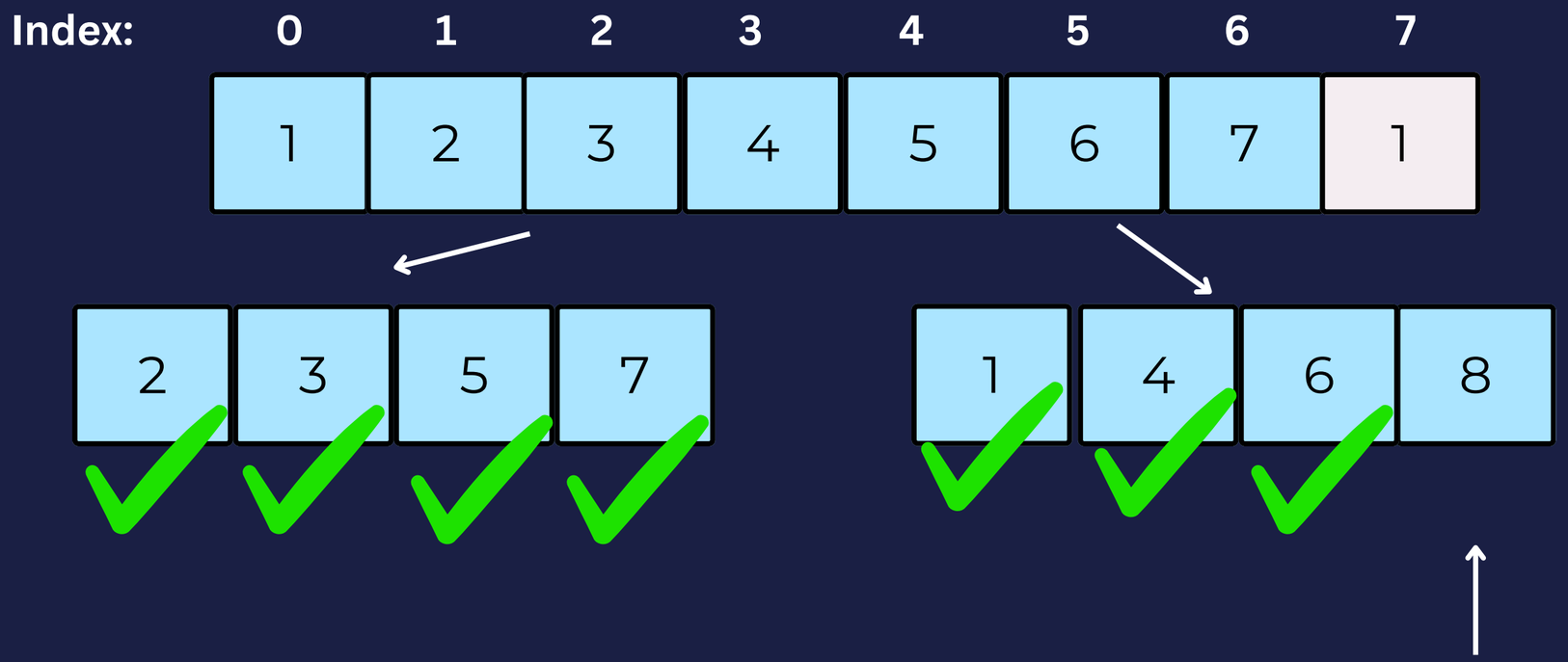
Lets go through this side quickly...



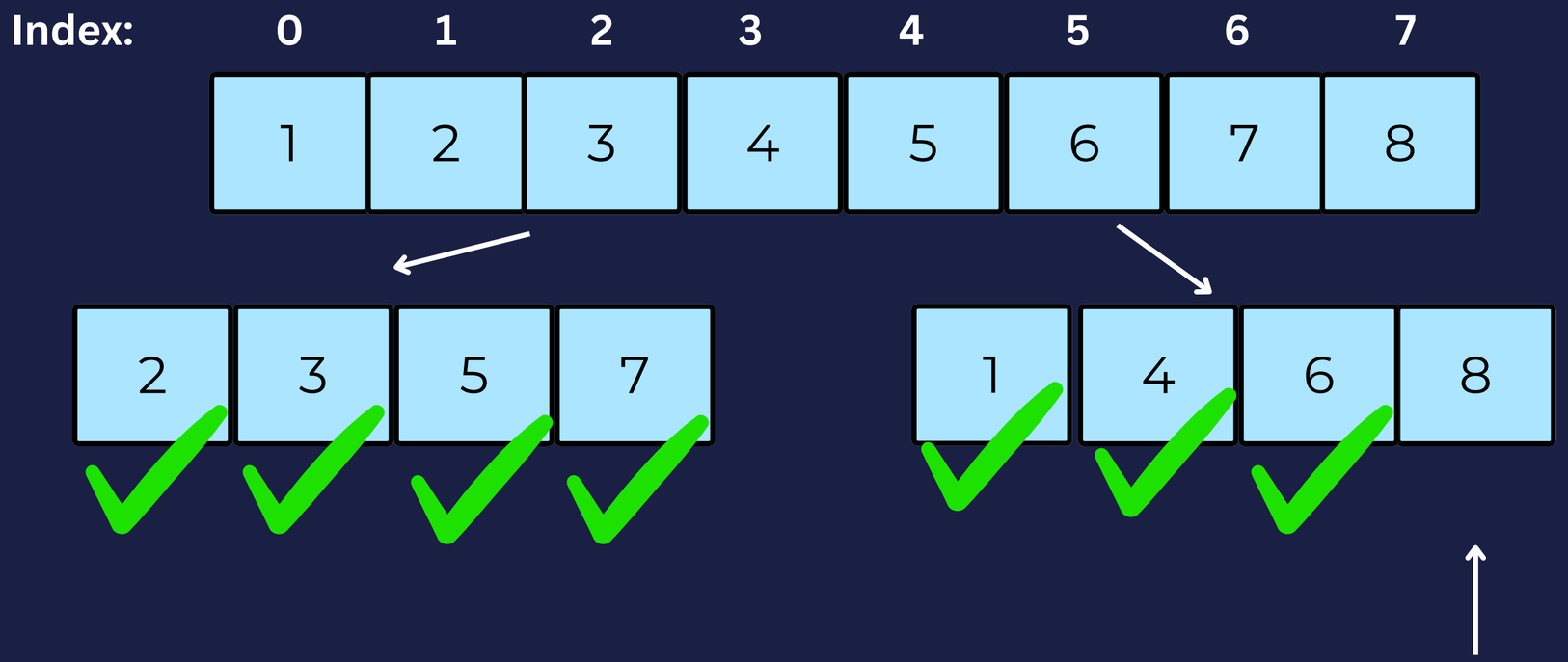
Lets go through this side quickly...



Lets go through this side quickly...



Lets go through this side quickly...



Now we have a sorted array!

Index:	0	1	2	3	4	5	6	7
	1	2	3	4	5	6	7	8

THE PSEUDOCODE

```
function MERGE_SORT(array):
    IF length(array) > 1:
        mid = length(array)/2
        left = array[0:mid]
        right = array[mid:length(array)]

        // Recursively sort both halves
        MERGE_SORT(left)
        MERGE_SORT(right)

        // Indexes for left(i), right(j) and merged array (k)
        WHILE i < length(left) AND j < length(right):
            IF left[i] < right[j]:
                array[k] = left[i]
                i++
            ELSE:
                array[k] = right[j]
                j++
```

THE PSEUDOCODE

```
        k++

        // Copy any remaining elements from left/right (if any)
        WHILE i < length(left):
            array[k] = left[i]
            i++
            k++

        WHILE j < length(right):
            array[k] = right[j]
            j++
            k++

// Call the function
MERGE_SORT(array)
```

EXAMPLES REPLIT! PLEASE GO TO:

HTTPS://REPLIT.COM/@RIKKIEHRHART/GRABABYTE

O NOTATION!

What is O Notation?

- aka “Big O Notation” is a way to describe how efficient an algorithm is as the size of the input grows.
- It tells us how *long* an algorithm might take or how much *work* it might need to do
- Essentially, how many steps or iterations at the *worst*

Common O Notations:

- Linear Time | O(n) - covered with Linear Search
- Logarithmic Time | O(log n) - covered with Binary Search
- Quadratic Time | O(n^2) - covered with Bubble Sort
- Log-Linear Time | O(n log n) - covering today!
- Constant Time | O(1) - cover with Hashing

LOG-LINEAR TIME - O(N LOG N)

Commonly used in recursive sorting algorithms.

With Log-Linear time, we take O(n), which is going through every item in a list and O(log n) which is when we keep cutting it in half and combine them into O(n log n).

So we are ***BOTH*** processing every item in a list {O(n)} and dividing the problem {O(log n)} repeatedly.

LOG-LINEAR TIME - O(N LOG N)

Think of it like this. You have 1 group of 8 boxes.

LOG-LINEAR TIME - O(N LOG N)

You split the group in half, which is our first iteration.

GROUP 1

GROUP 2

Iterations: 1

LOG-LINEAR TIME - O(N LOG N)

You split those groups in half, which makes iteration number 2.

GROUP 1 GROUP 2 GROUP 3 GROUP 4

Iterations: 2

LOG-LINEAR TIME - O(N LOG N)

Then you split them up a final time (iteration 3)

GROUP 1

GROUP 2

GROUP 3

GROUP 4

GROUP 5

GROUP 6

GROUP 7

GROUP 8

Iterations: 3

LOG-LINEAR TIME - O(N LOG N)

That makes the O(log n) = 3. Now lets find the O(n).

GROUP 1

GROUP 2

GROUP 3

GROUP 4

GROUP 5

GROUP 6

GROUP 7

GROUP 8

Iterations: 3

LOG-LINEAR TIME - O(N LOG N)

We check each box in the group!

GROUP 1

GROUP 2

GROUP 3

GROUP 4

GROUP 5

GROUP 6

GROUP 7

GROUP 8

Iterations: 0

LOG-LINEAR TIME - O(N LOG N)

We check each box in the group!

GROUP 1

GROUP 2

GROUP 3

GROUP 4

GROUP 5

GROUP 6

GROUP 7

GROUP 8

Iterations: 1

LOG-LINEAR TIME - O(N LOG N)

We check each box in the group!

GROUP 1

GROUP 2

GROUP 3

GROUP 4

GROUP 5

GROUP 6

GROUP 7

GROUP 8

Iterations: 2

LOG-LINEAR TIME - O(N LOG N)

We check each box in the group!

GROUP 1

GROUP 2

GROUP 3

GROUP 4

GROUP 5

GROUP 6

GROUP 7

GROUP 8

Iterations: 3

LOG-LINEAR TIME - O(N LOG N)

We check each box in the group!

GROUP 1

GROUP 2

GROUP 3

GROUP 4

GROUP 5

GROUP 6

GROUP 7

GROUP 8

Iterations: 4

LOG-LINEAR TIME - O(N LOG N)

Checking each box in this grouping is 8 iterations

GROUP 1

GROUP 2

GROUP 3

GROUP 4

GROUP 5

GROUP 6

GROUP 7

GROUP 8

Iterations: 8

LOG-LINEAR TIME - O(N LOG N)

And checking each box in the 4 boxes grouping is 8 iterations

GROUP 1

GROUP 2

GROUP 3

GROUP 4

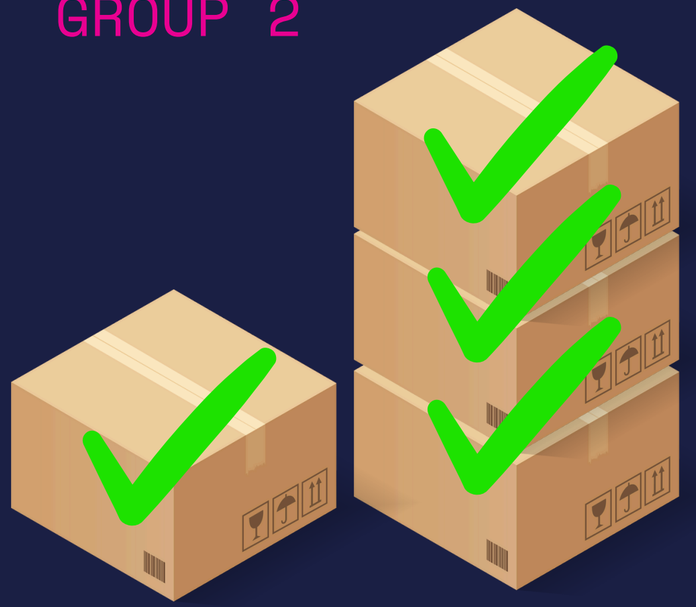
Iterations: 8

LOG-LINEAR TIME - O(N LOG N)

And checking each box in the 2 boxes grouping is 8 iterations

GROUP 1

GROUP 2



Iterations: 8

LOG-LINEAR TIME - O(N LOG N)

And checking each box in the whole boxes grouping is 8 iterations

Iterations: 8

LOG-LINEAR TIME - O(N LOG N)

Therefore the O(n) = 8.

So if the O(log n) = 3 and the O(n) = 8

that means that O(n log n) = 8 * 3

Total

Iterations: 24

UP NEXT

Mar 12 - Quick Sort

SPRING BREAK!

Mar 26 - Breadth-First Search (BFS)

Apr 2 - Depth-First Search (DFS)

Apr 9 Hashing

Apr 16 - Dijkstra's Algorithm

Apr 23 - Dynamic Programming (Knapsack Problem)

Apr 30 - Union-Find

May 7 - Kruskal's Algorithm

May 14 - Prim's Algorithm

Questions? - rikki.ehrhart@g.ausitncc.edu

If you'd like the opportunity to run a Grab a Byte algorithm workshop, please let me know!