BUBBLE SORT!

LINEAR AND BINARY SEARCH

Over the last two weeks we covered Linear Search (searching a list one-by-one) and binary search (searching a list by continuously dividing the list in half).

SORTING ALGORITHMS

Now we are moving on to sorting algorithms, which rearranges a list into a desired order (for example: alphabetical or numerical)

BUBBLE SORT

A bubble sort algorithm repeatedly compares adjacent elements and swapping their positions if they are out of order. It repeatedly iterates through the list, comparing each pair of elements and swapping them if necessary, until no further swaps are needed, signifying a sorted list

Lets consider an array of integer values:

In a Bubble Sort Algorithm, it checks the first two numbers to see if they are in the right order

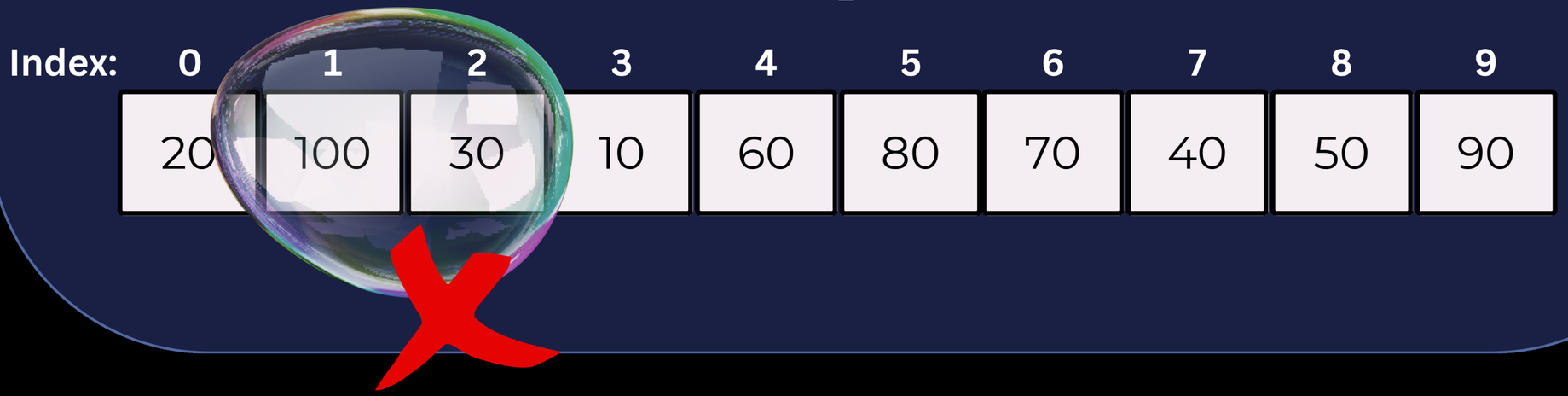
Index:	0	1	2	3	4	5	6	7	8	9
	20	100	30	10	60	80	70	40	50	90

In a Bubble Sort Algorithm, it checks the first two numbers to see if they are in the right order

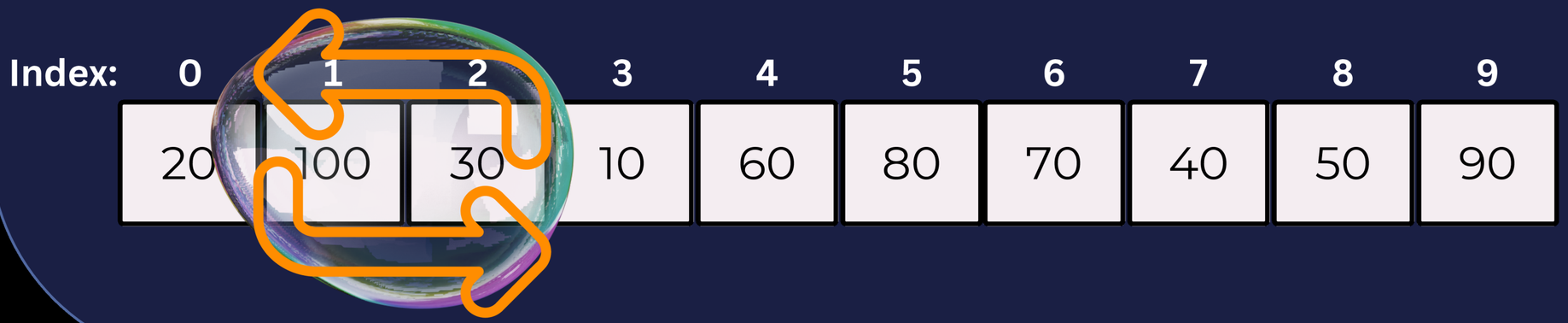
Index:	0	1	2	3	4	5	6	7	8	9
	20	100	30	10	60	80	70	40	50	90

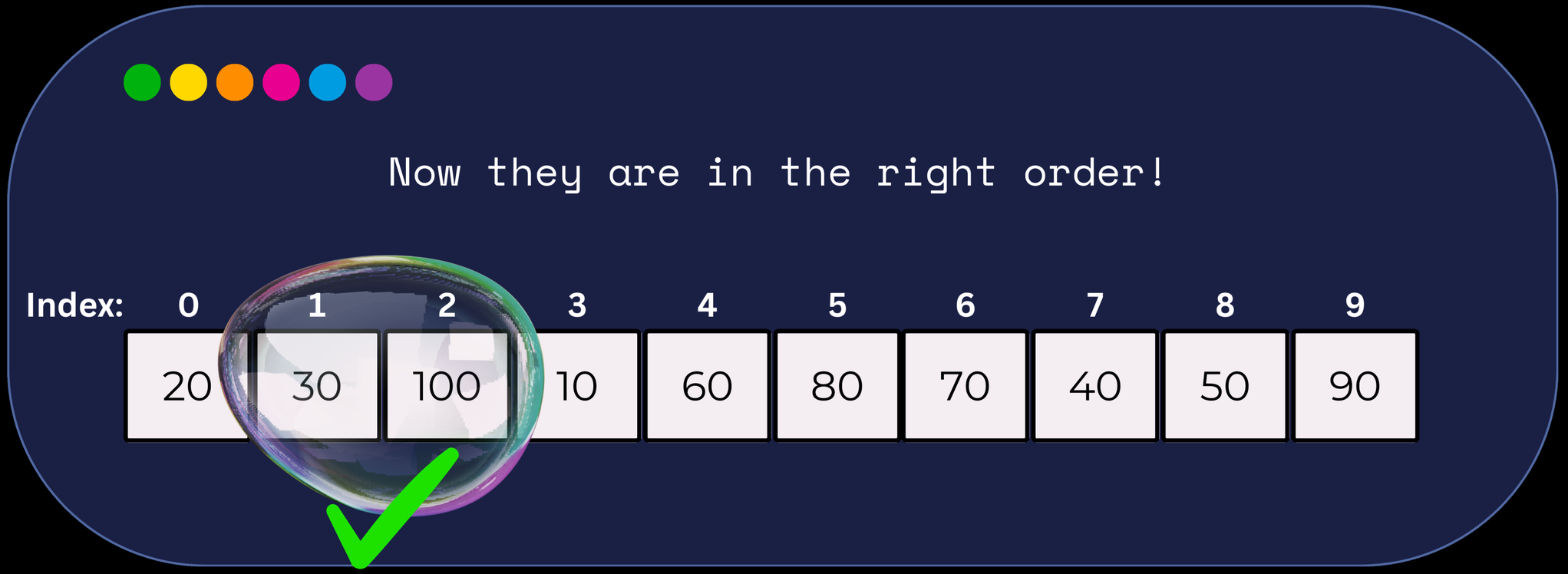
Then it checks the next two items to see if they are in the right order.

Then it checks the next two items to see if they are in the right order.

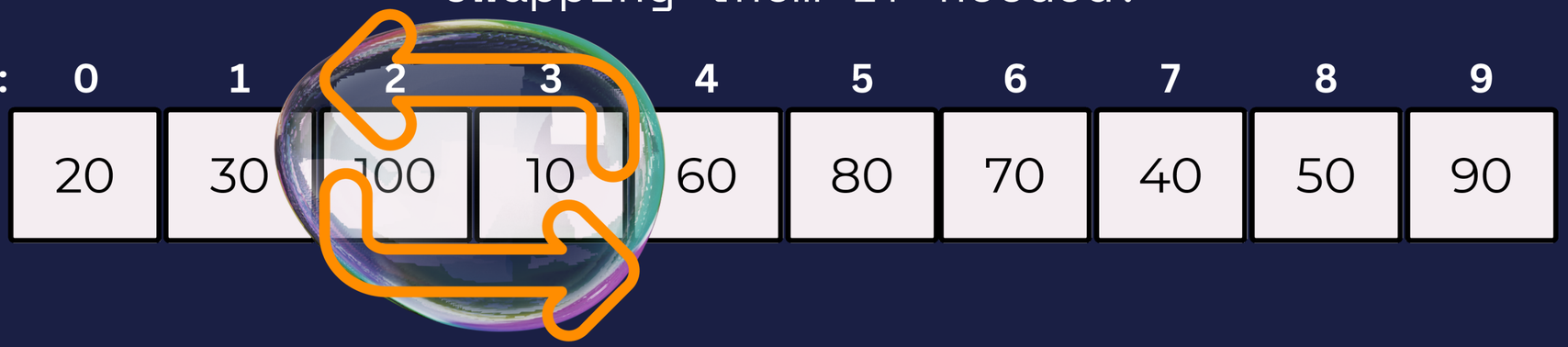


These are not in numeric order, so we swap them!

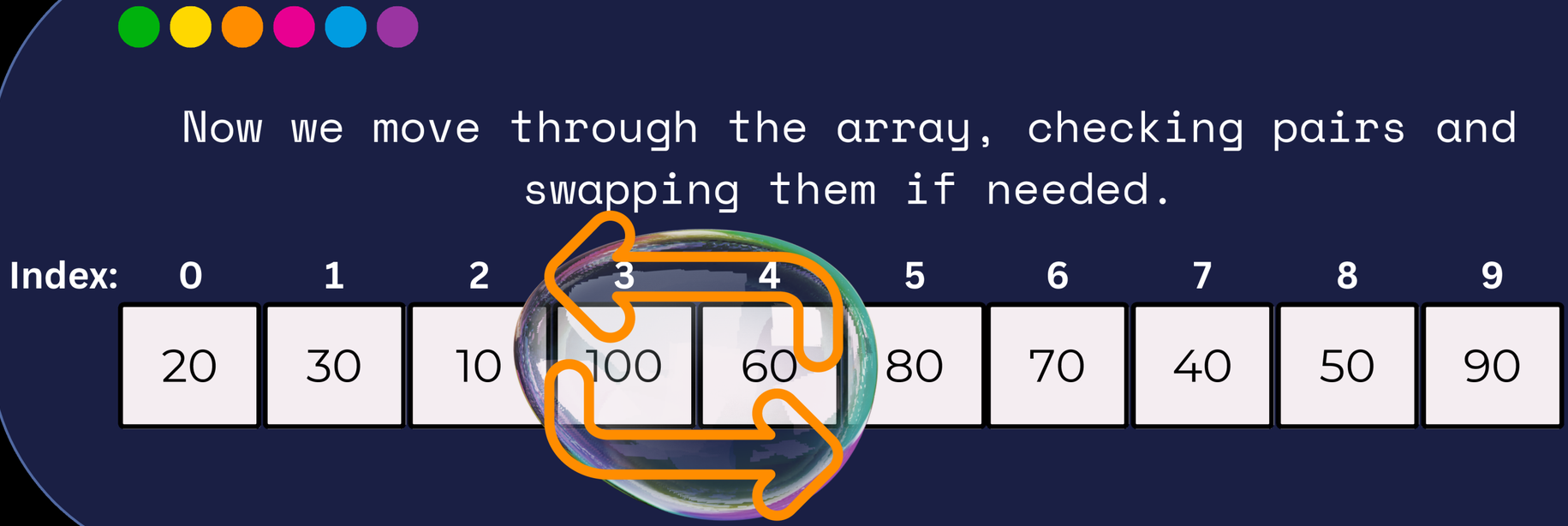



Now they are in the right order!
Index:
0
1
2
3
4
5
6
7
8
9
20
30
100
10
60
80
70
40
50
90

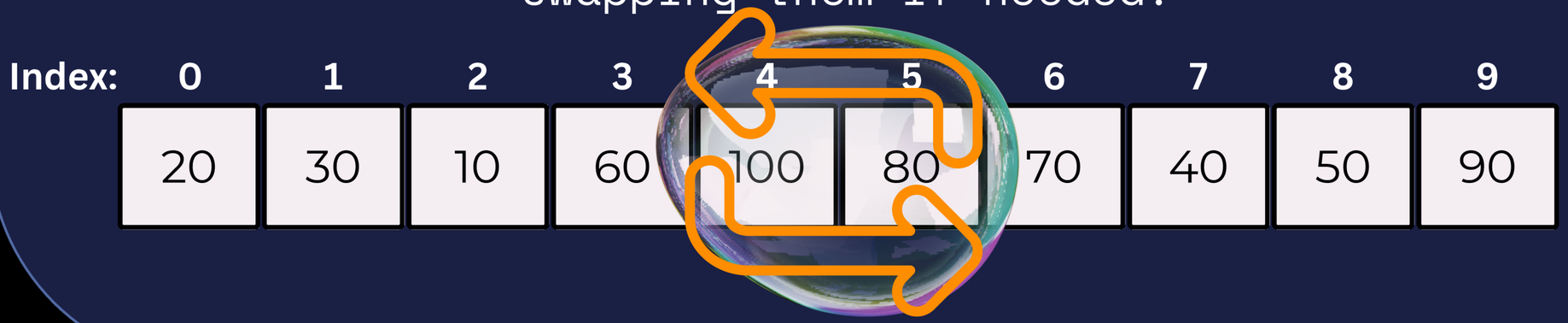
Now we move through the array, checking pairs and swapping them if needed.



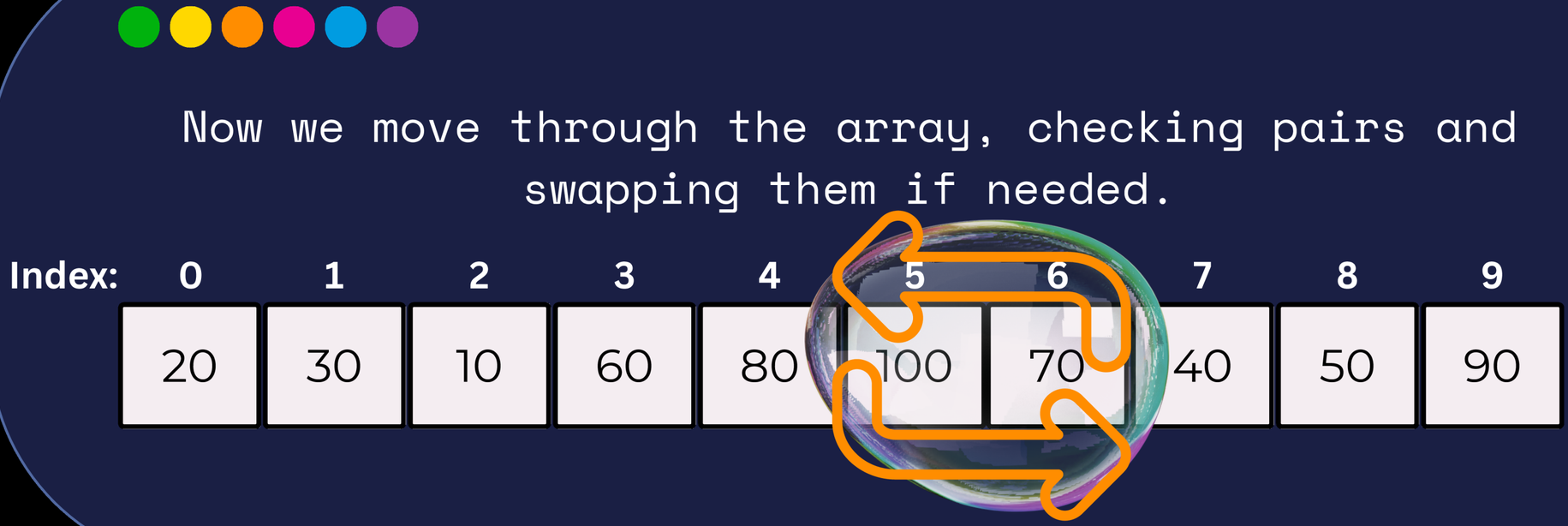
Now we move through the array, checking pairs and swapping them if needed.



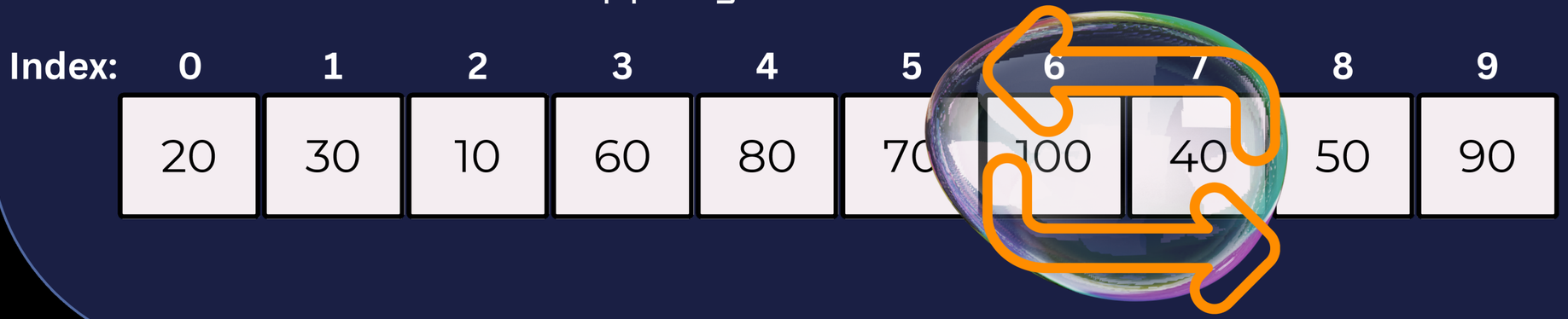
Now we move through the array, checking pairs and swapping them if needed.



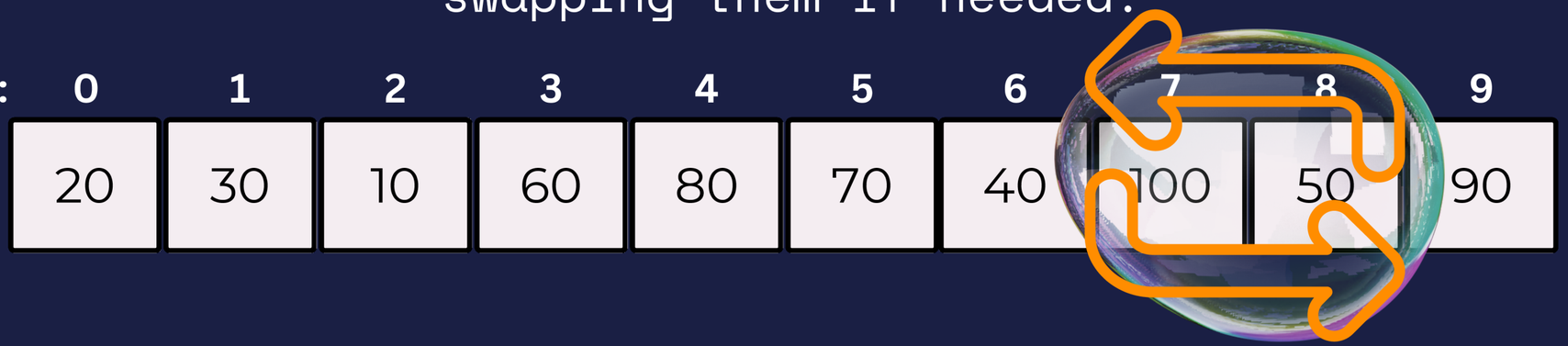
Now we move through the array, checking pairs and swapping them if needed.



Now we move through the array, checking pairs and swapping them if needed.



Now we move through the array, checking pairs and swapping them if needed.

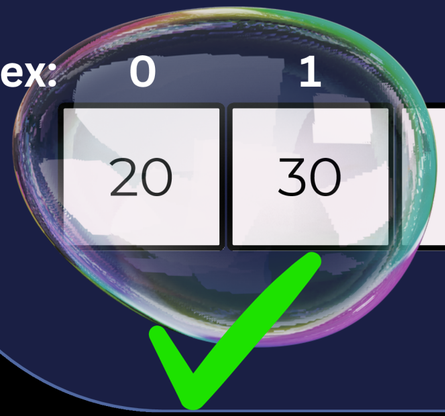


Now we move through the array, checking pairs and swapping them if needed.

Index:	0	1	2	3	4	5	6	7	8	9
	20	30	10	60	80	70	40	50	100	90

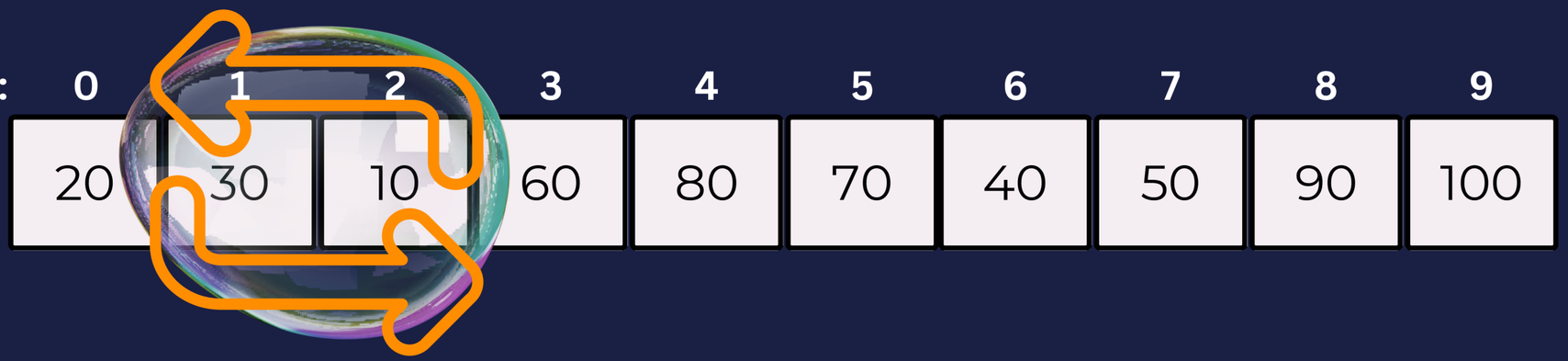
We've now finished one pass through the array, but we will need start again from the beginning to continue storting!

Round 2!

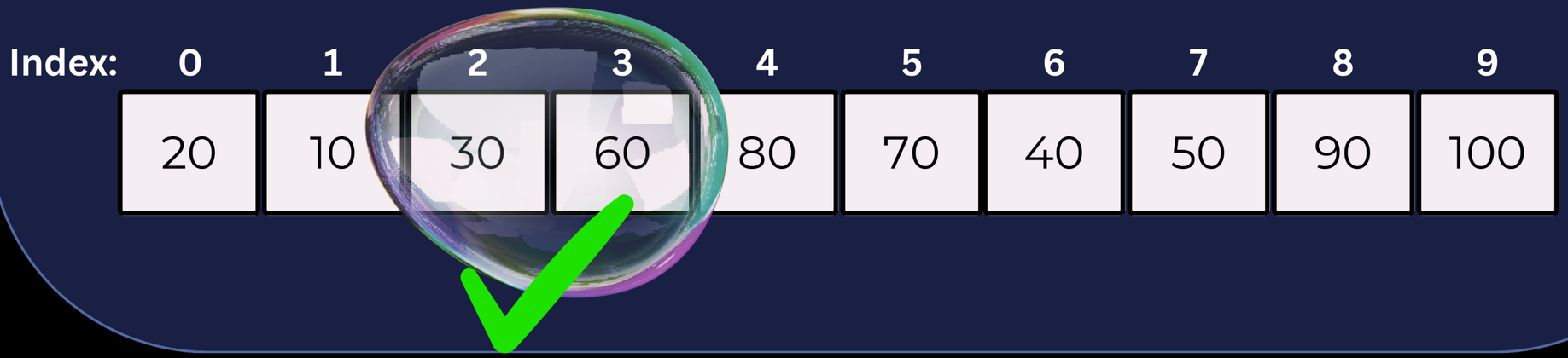


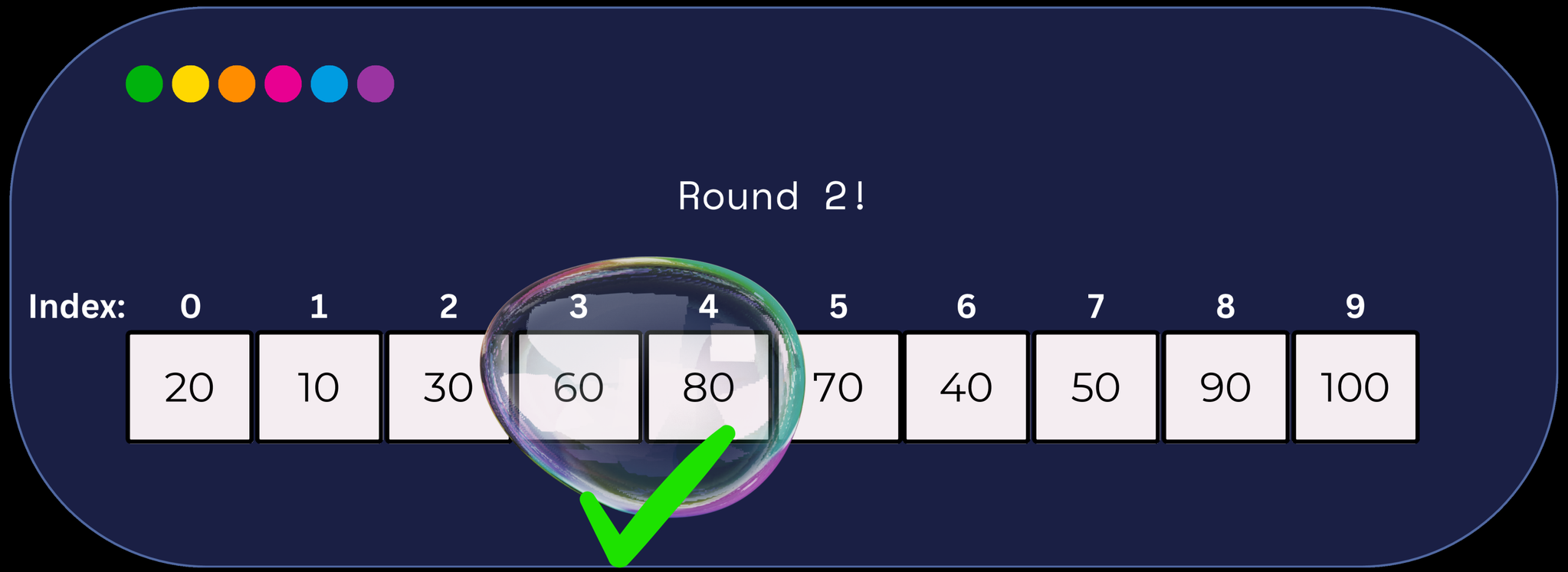
Index:	0	1	2	3	4	5	6	7	8	9
	20	30	10	60	80	70	40	50	90	100

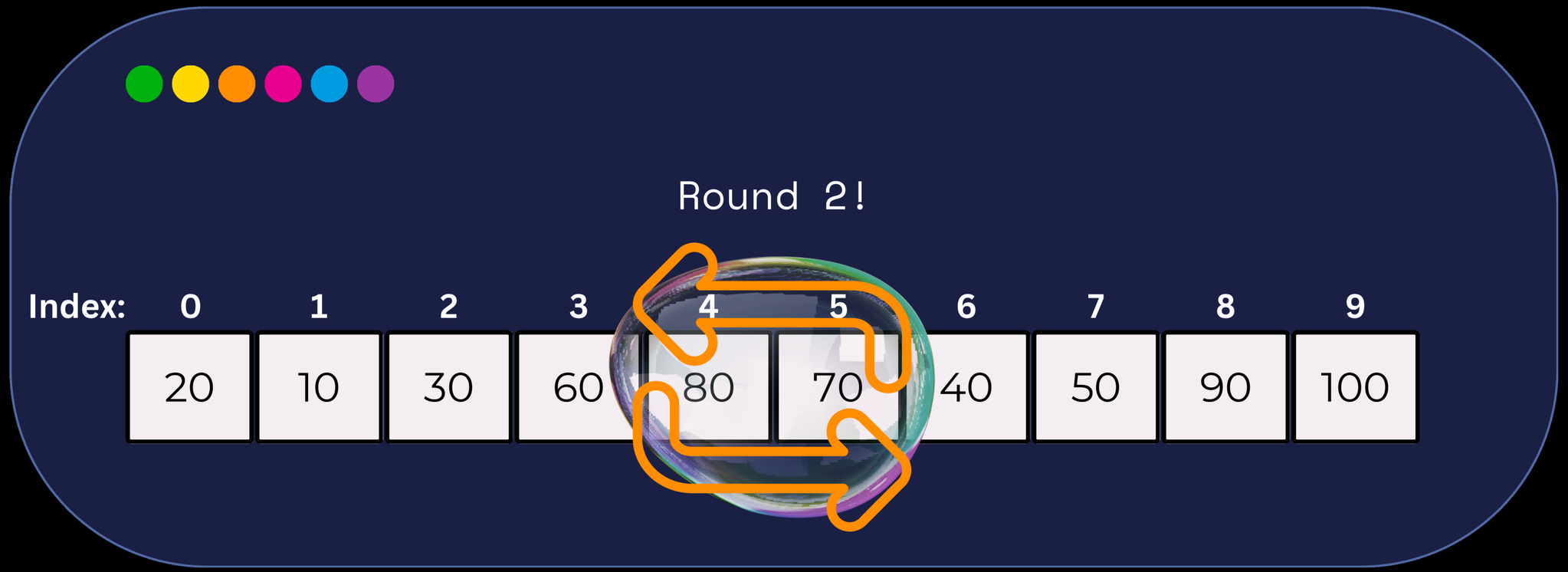
Round 2!



Round 2!



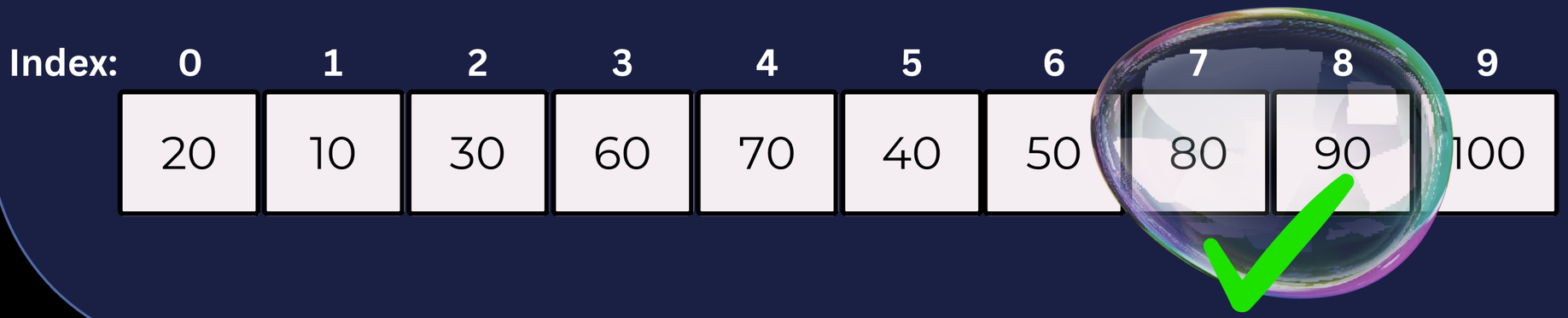

Round 2!
Index:
0
1
2
3
4
5
6
7
8
9
20
10
30
60
80
70
40
50
90
100


Round 2!
Index:
0
1
2
3
4
5
6
7
8
9
20
10
30
60
80
70
40
50
90
100

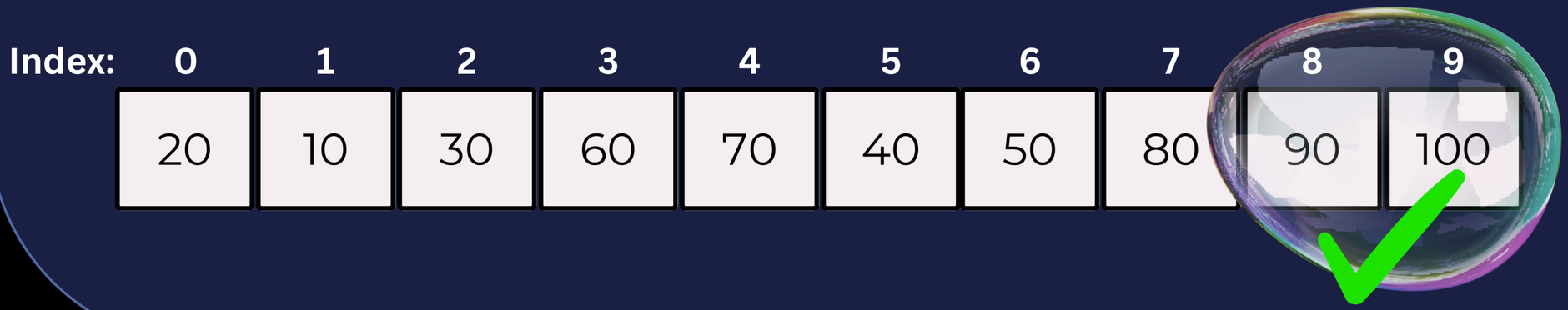
Round 2!

Round 2!

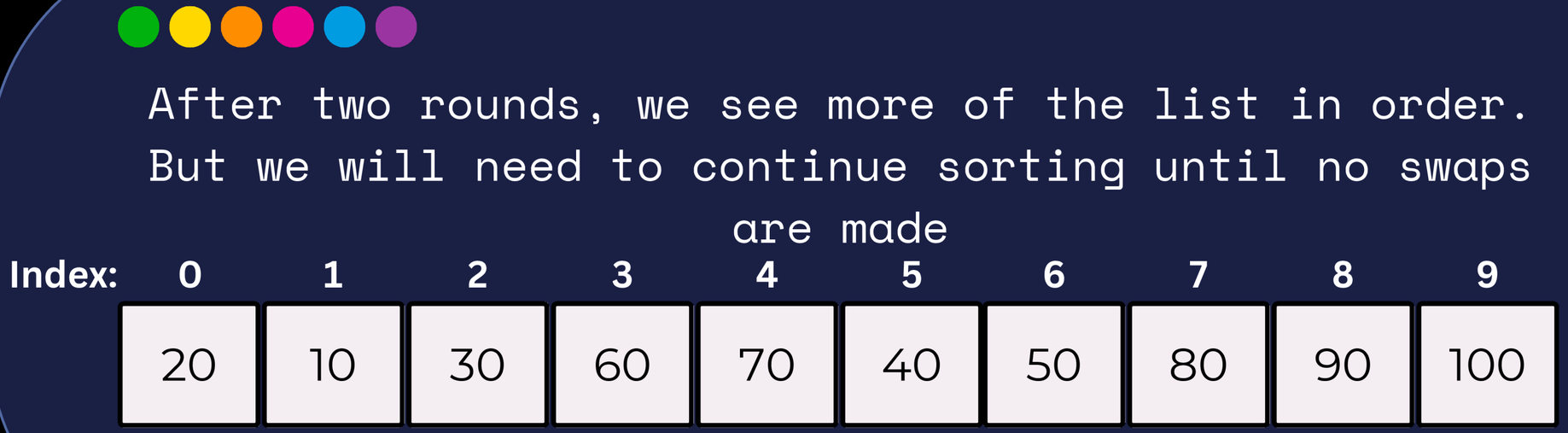
Round 2!



Round 2!



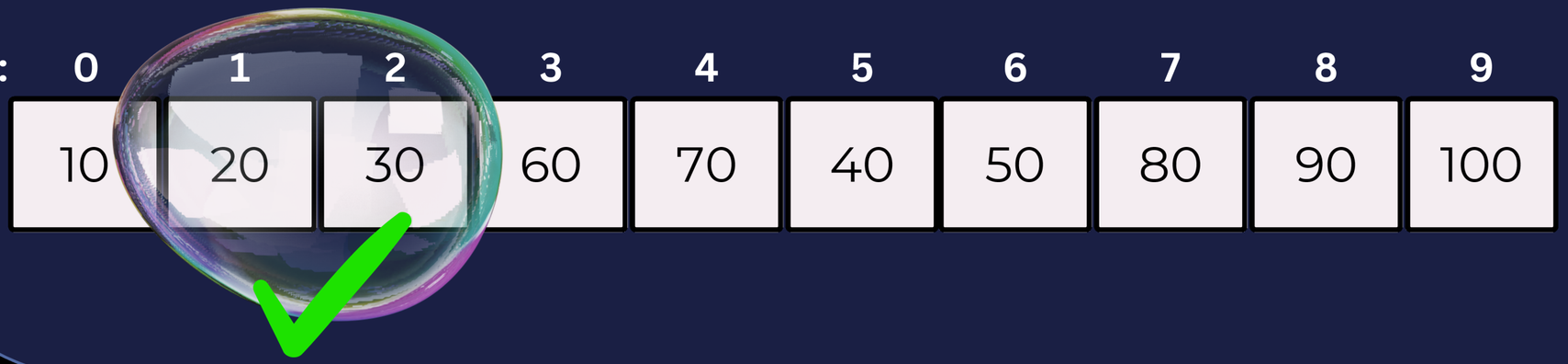
After two rounds, we see more of the list in order. But we will need to continue sorting until no swaps are made



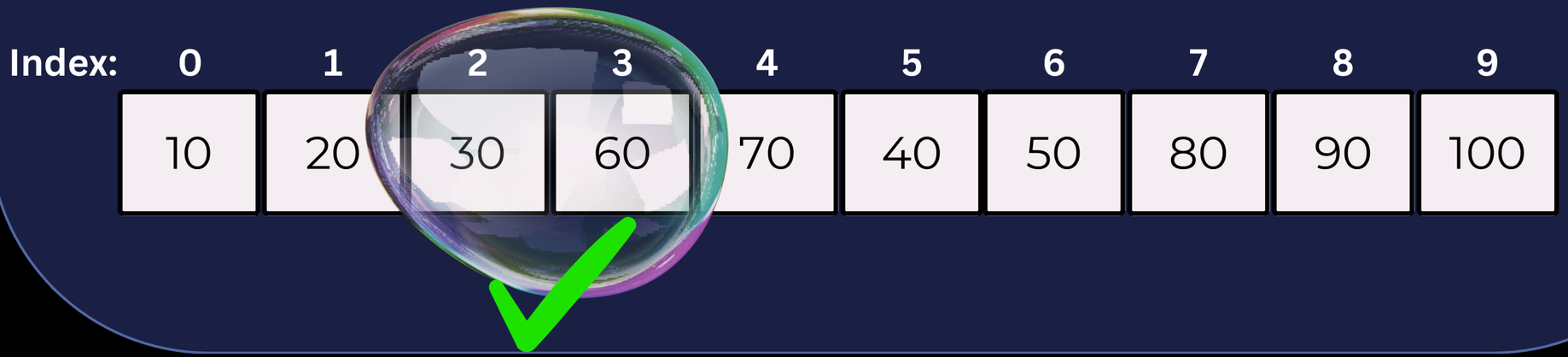
Round 3!

Index:	0	1	2	3	4	5	6	7	8	9
	20	10	30	60	70	40	50	80	90	100

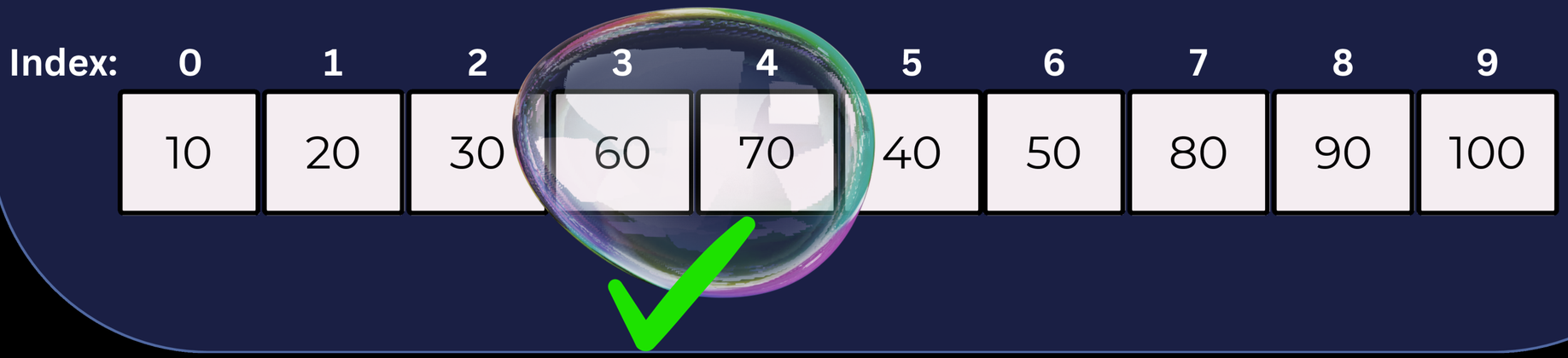
Round 3!

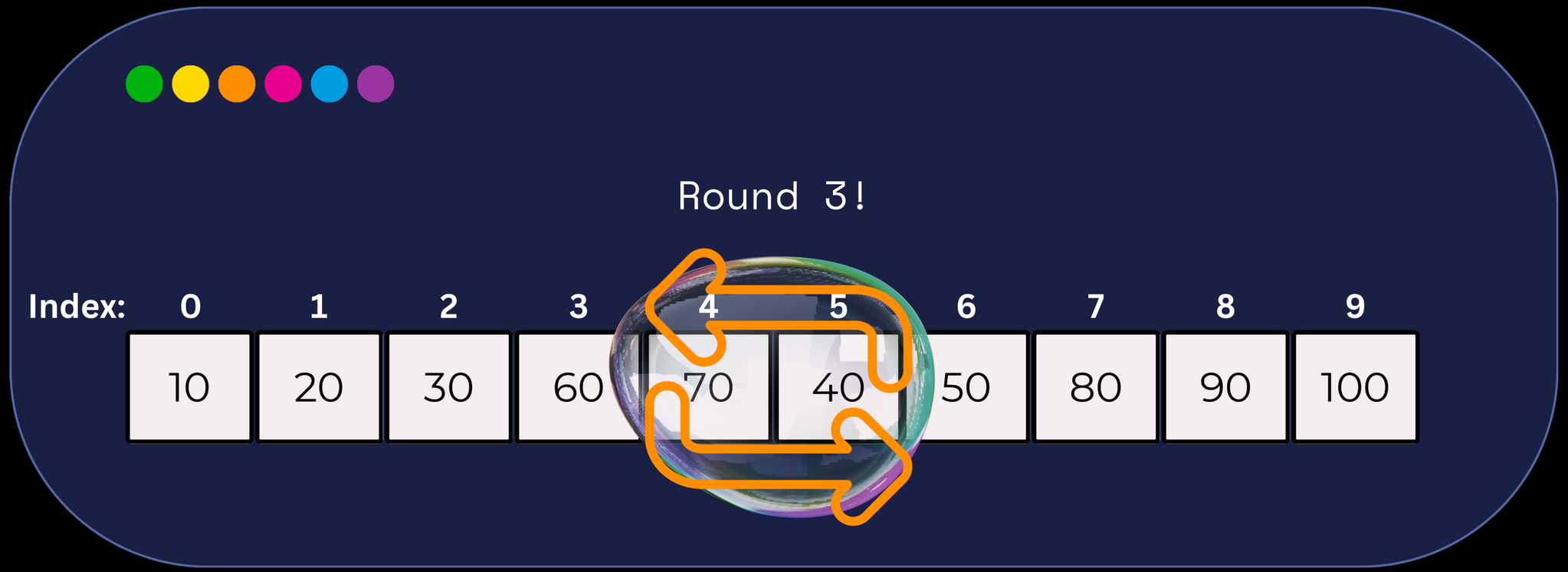


Round 3!

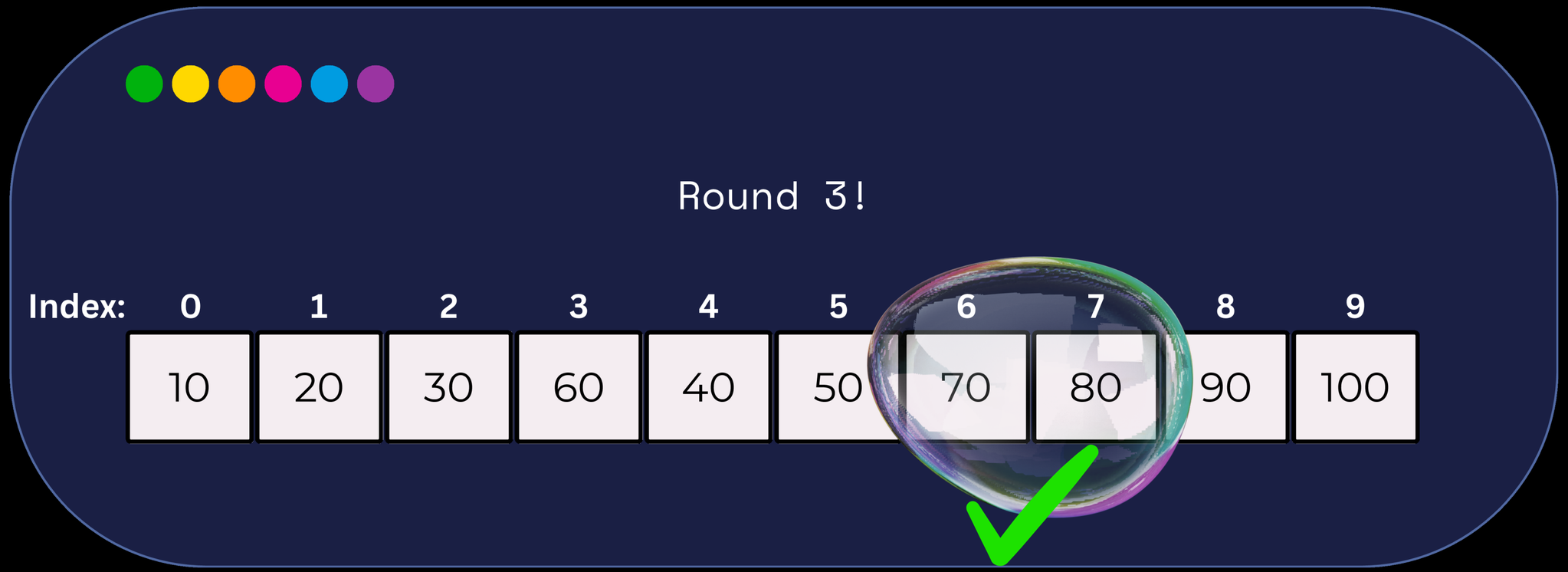


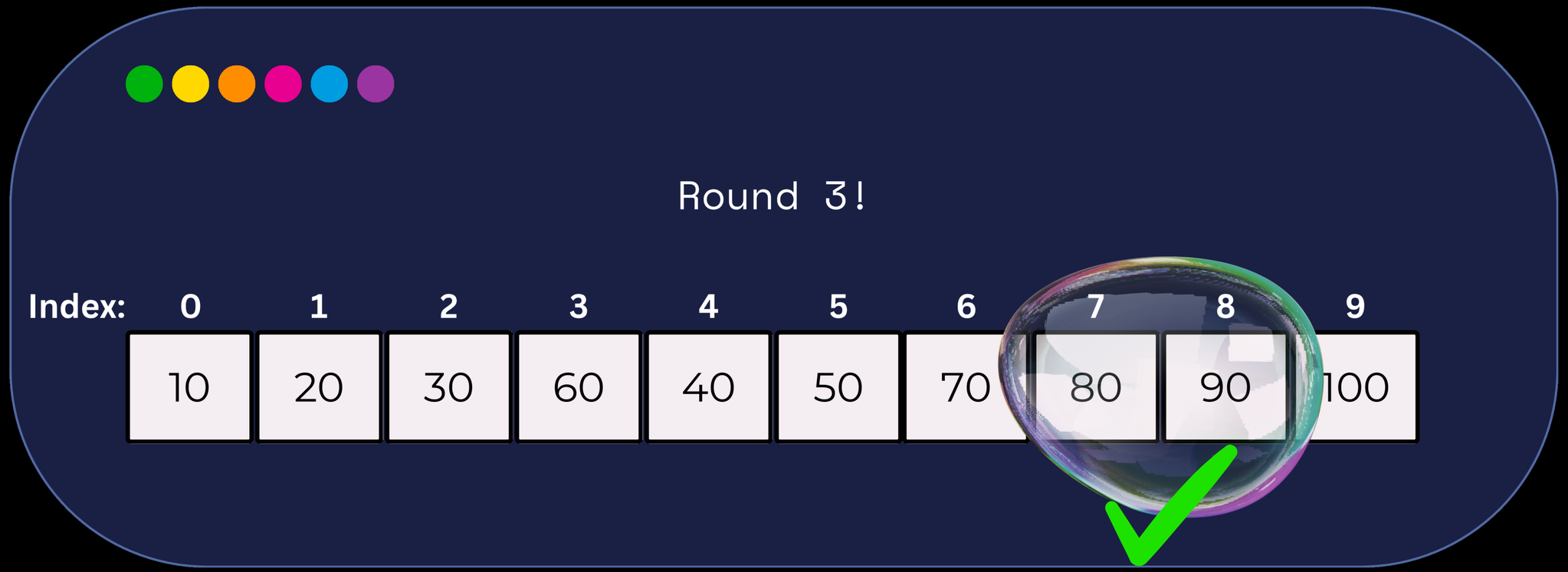
Round 3!



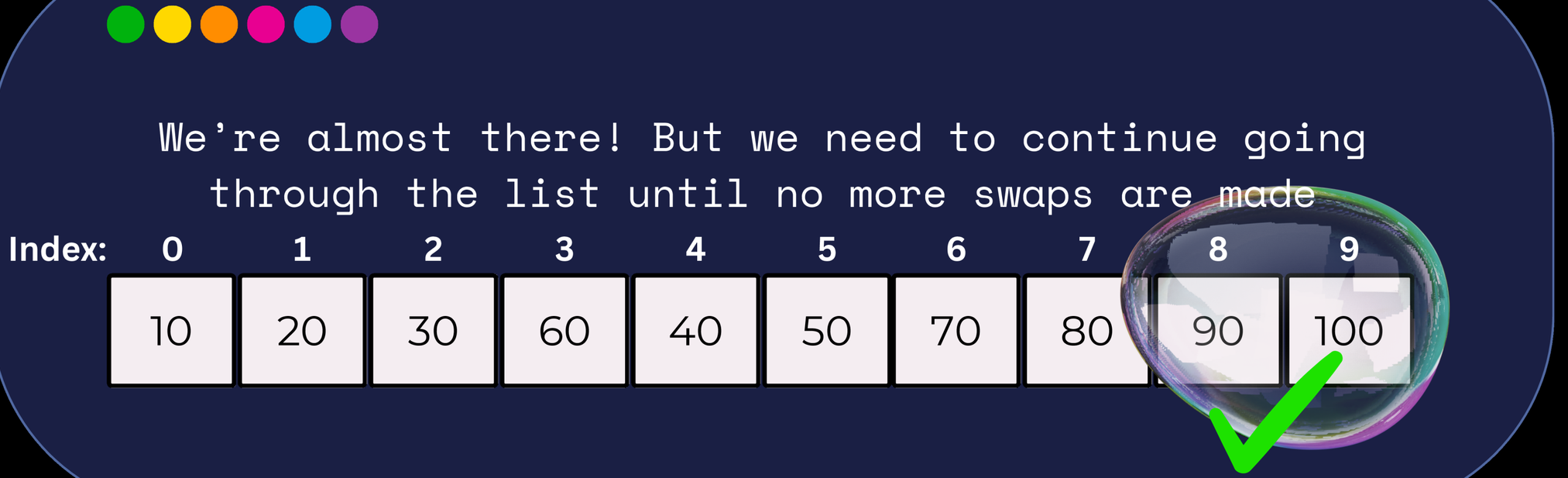

Round 3!
Index:
0
1
2
3
4
5
6
7
8
9
10
20
30
60
70
40
50
80
90
100

Round 3!

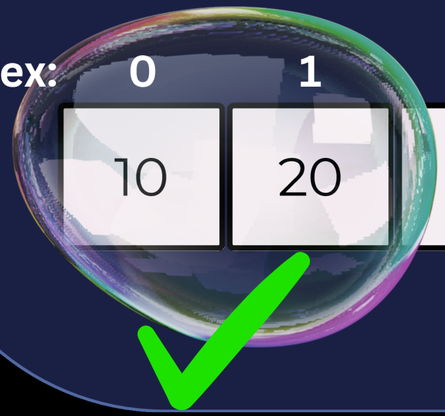

Round 3!
Index:
0
1
2
3
4
5
6
7
8
9
10
20
30
60
40
50
70
80
90
100


Round 3!
Index:
0
1
2
3
4
5
6
7
8
9
10
20
30
60
40
50
70
80
90
100

We're almost there! But we need to continue going through the list until no more swaps are made

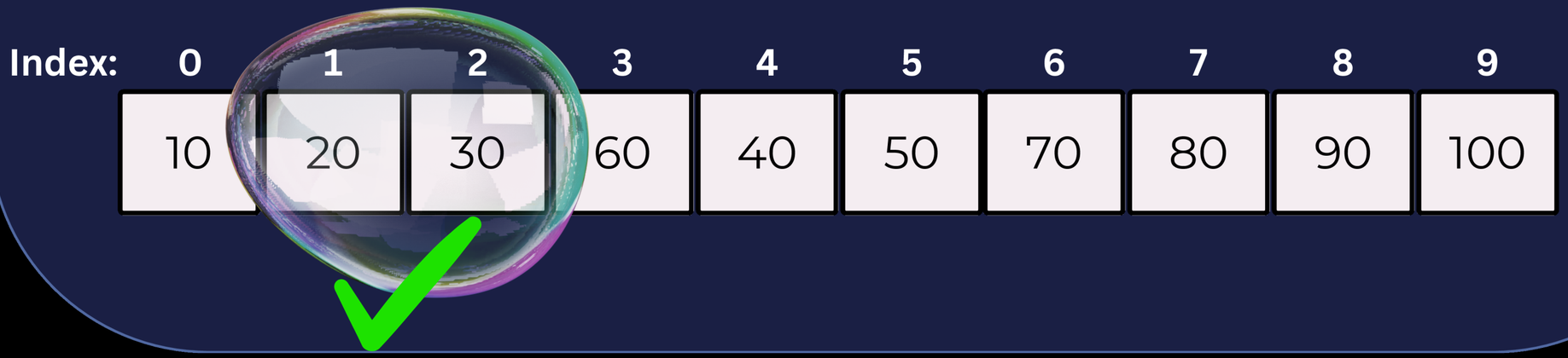


Round 4!

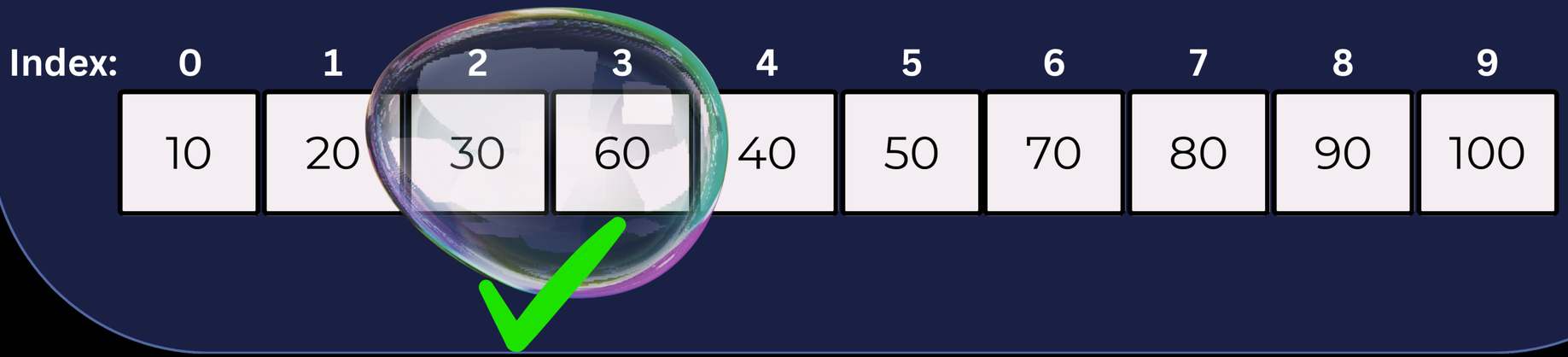


Index:	0	1	2	3	4	5	6	7	8	9
	10	20	30	60	40	50	70	80	90	100

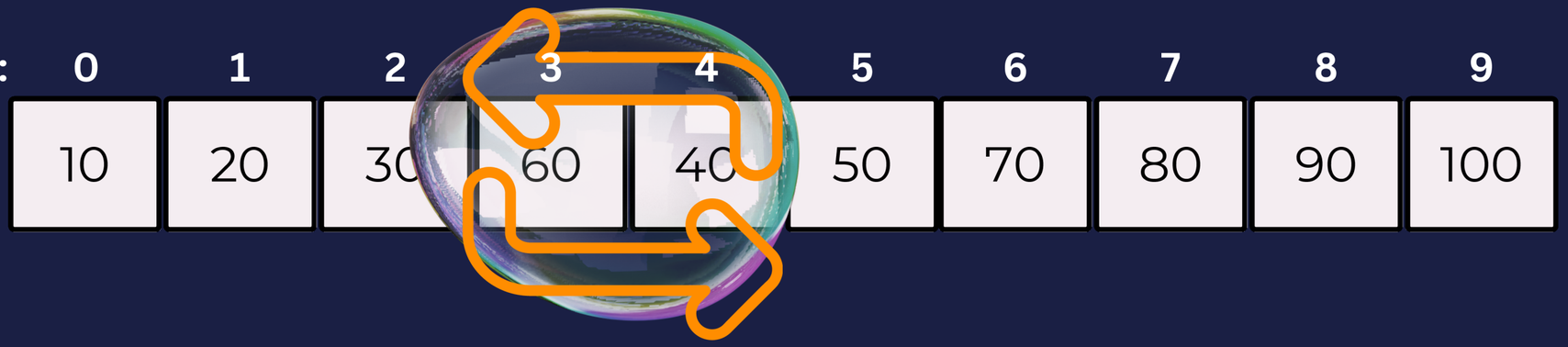
Round 4!

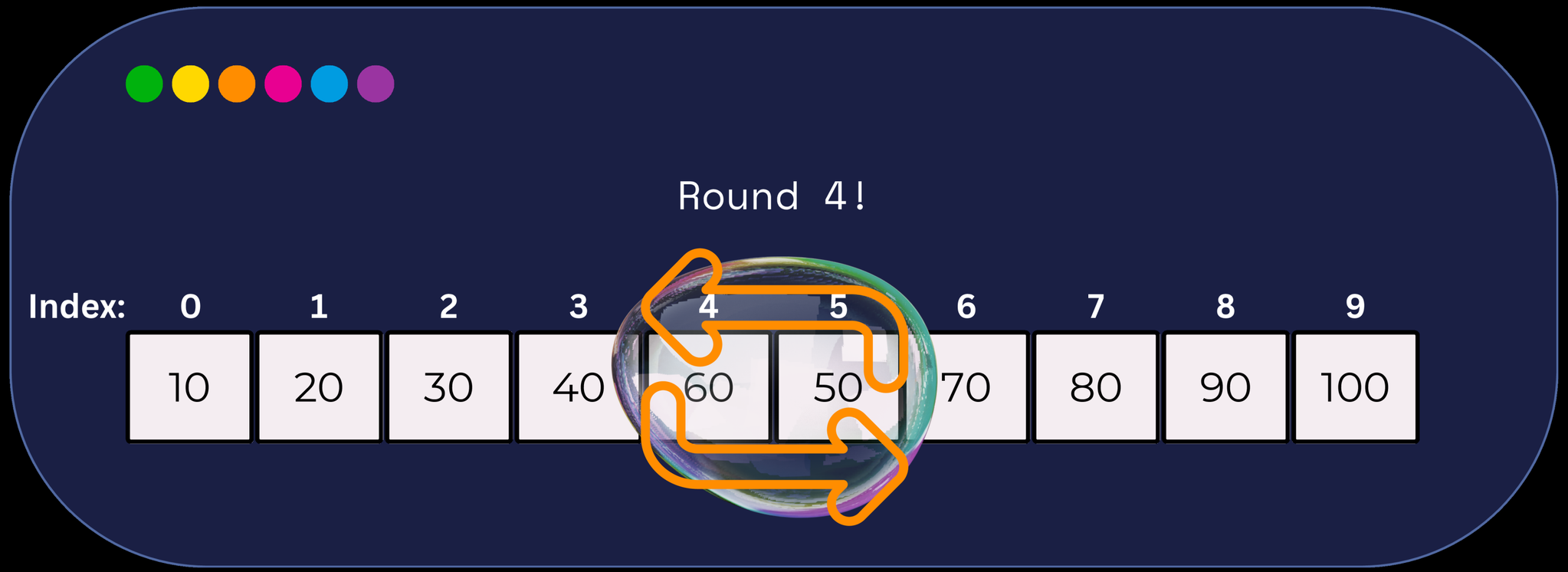


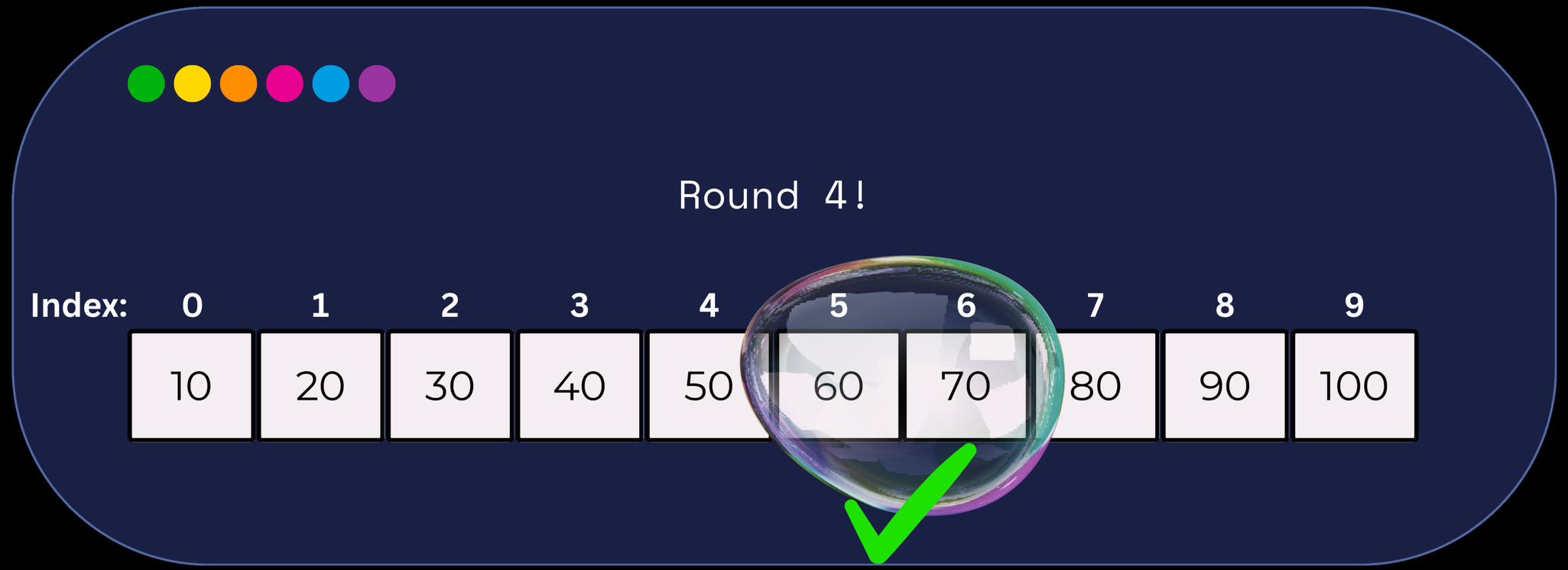
Round 4!

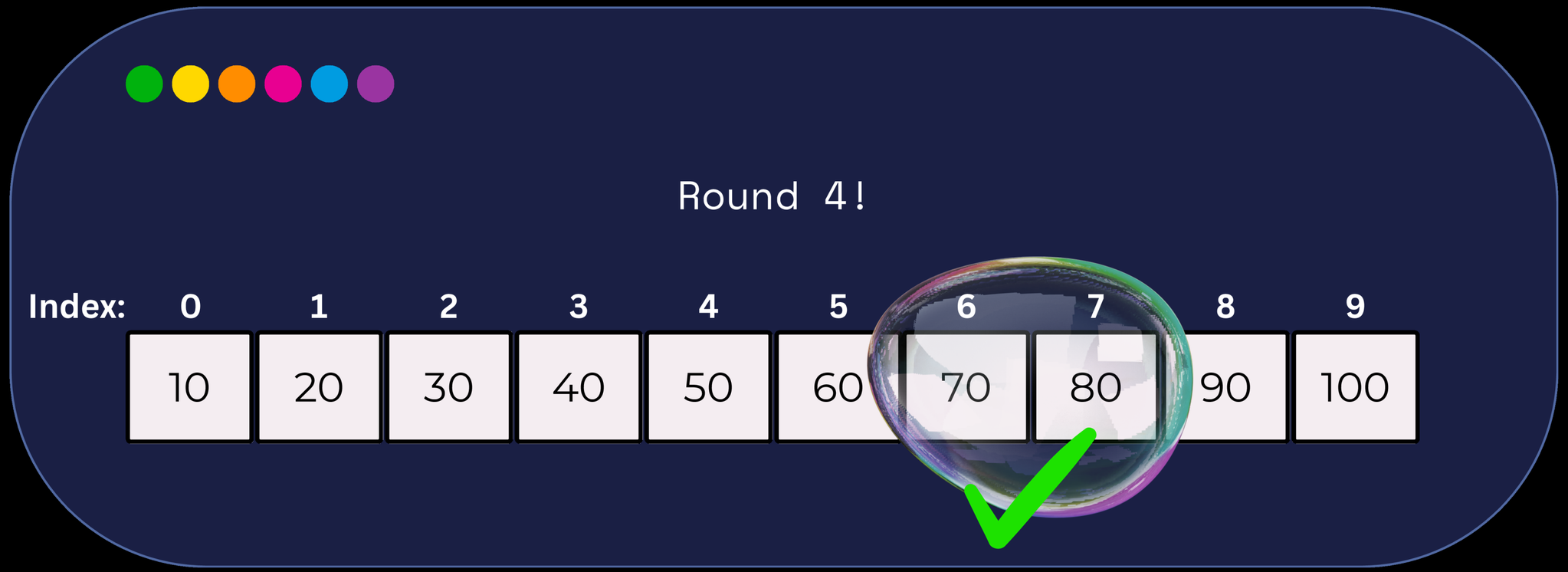


Round 4!




Round 4!
Index:
0
1
2
3
4
5
6
7
8
9
10
20
30
40
60
50
70
80
90
100

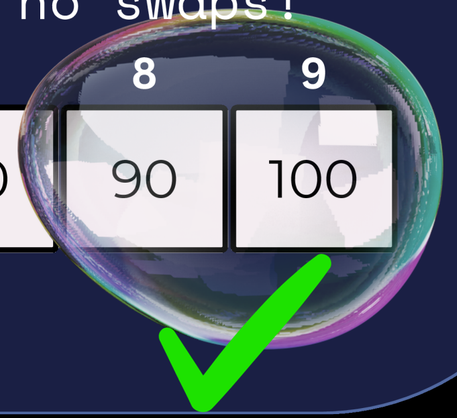

Round 4!
Index:
0
1
2
3
4
5
6
7
8
9
10
20
30
40
50
60
70
80
90
100


Round 4!
Index:
0
1
2
3
4
5
6
7
8
9
10
20
30
40
50
60
70
80
90
100

Round 4!

The array is now in the right order! When we go through the last round, there will be no swaps!

Index:	0	1	2	3	4	5	6	7	8	9
	10	20	30	40	50	60	70	80	90	100



THE PSEUDOCODE

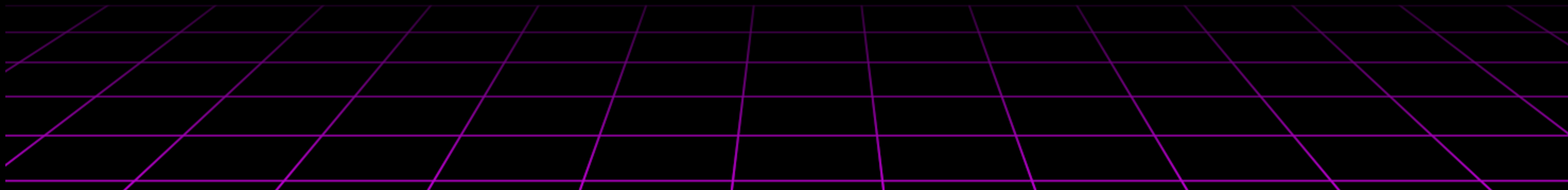
```
array = [20, 100, 30, 10, 60, 80, 70, 40, 50, 90]
n = len(array)

for i in range(n)
    for j in range(n-i-1):
        if Array[j] > Array[j+1]
            temp = Array[j]
            Array[j] = Array[j+1]
            Array[j+1] = temp
```

swa

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:

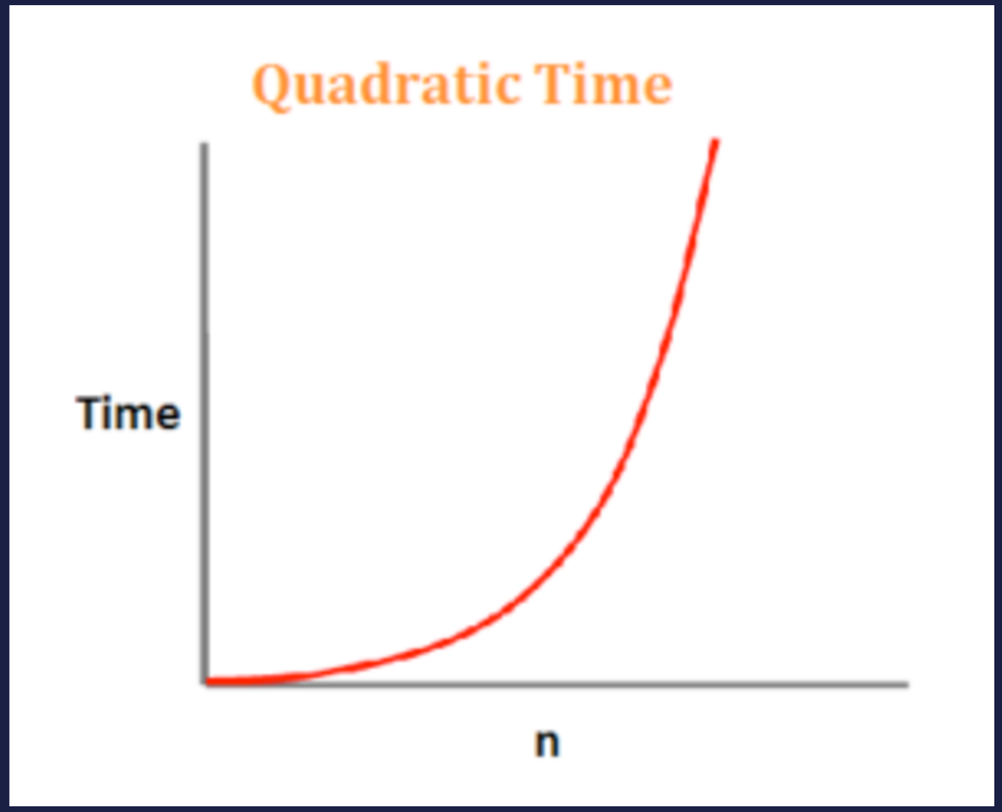
- Linear Time | O(n) - Covered with Linear Search
- Logarithmic Time | O(log n) -Covered with Binary Search
- Quadratic Time | O(n^2) - What we’ll cover today!
- Log-Linear Time | O(n log n) - cover with Merge Sort
- Constant Time | O(1) - cover with Hashing

QUADRATIC TIME - O(N^2)

When the input size doubles, the execution time quadruples.

Identifying characteristic: Algorithms with nested loops often exhibit quadratic time complexity.

QUADRATIC TIME - O(N^2)



Essentially, as the list gets bigger, the time it takes to complete drastically increases proportionally!

UP NEXT

Feb 19 - Selection Sort
Feb 26 - Insertion Sort
Mar 5 - Merge Sort
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!