Topic 6.1: Searching & Sorting

Learning Goals (Week 7):

- Write code that implements linear search on an array
- Write code that implements binary search on an array
- **Describe sorting algorithms such as insertion sort in plain English.**
- Describe sorting algorithms such as selection sort in plain English.
- Describe sorting algorithms such as merge sort in English.
- **Write code that implements insertion sort**
- Write code that implements selection sort
- Write code implementing merging two arrays from merge sort

Keeping a sorted list

- If we know we might need a sorted array, why not try to keep it ordered at all times, after each insertion?

- The idea is: always keep it sorted as you're creating it

- When adding a new element to a (partially-filled) array
  - the old way (adding it to the end) won't work:

```
data[numItems++] = newItem;
```

Keeping a sorted list

- We must now insert it into the proper spot to keep the array sorted (an “ordered insert” – slower, harder):

0	1	2	3	4	5	6	7
1	2	3	5	6	-	-	-

currSize

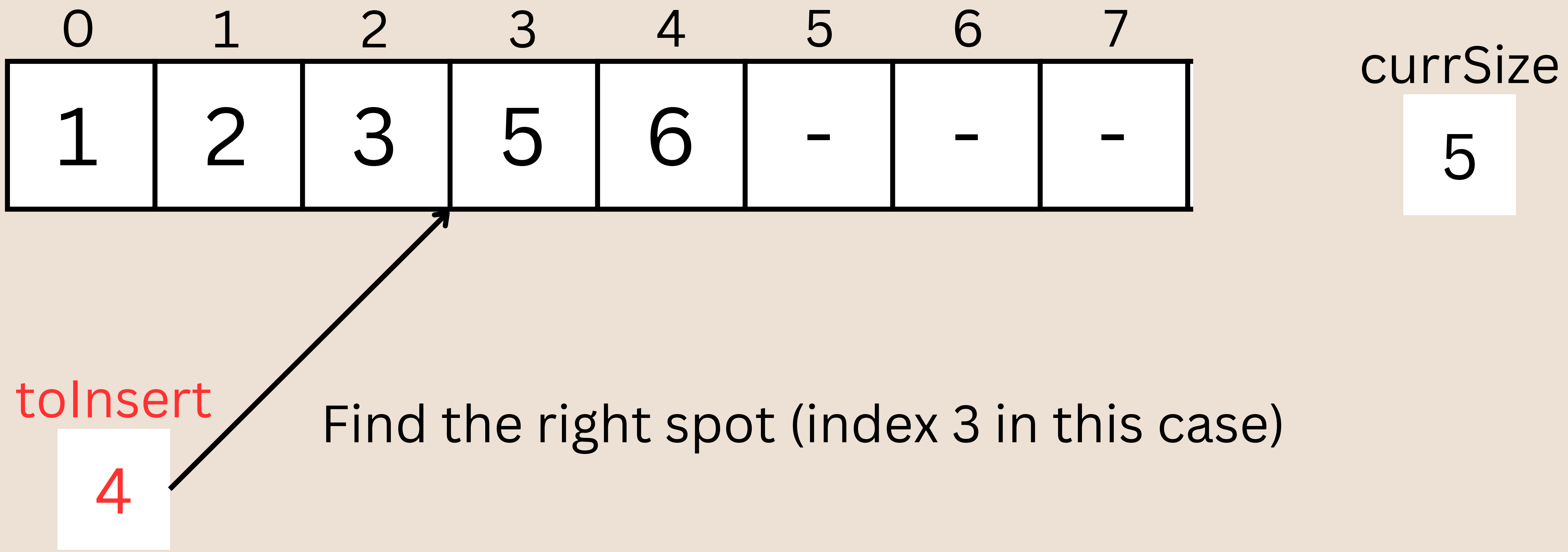
5

toInsert

4

Keeping a sorted list

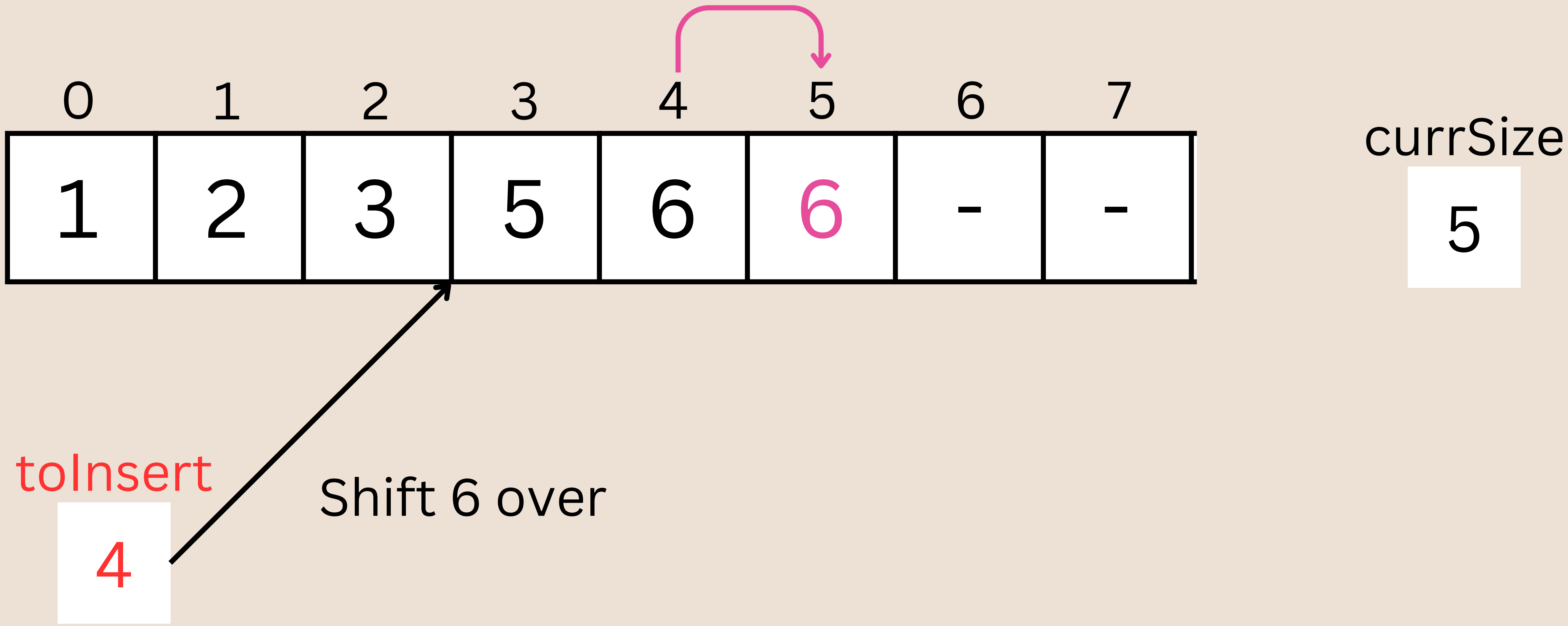
- We must now insert it into the proper spot to keep the array sorted (an “ordered insert” – slower, harder):



Find the right spot (index 3 in this case)

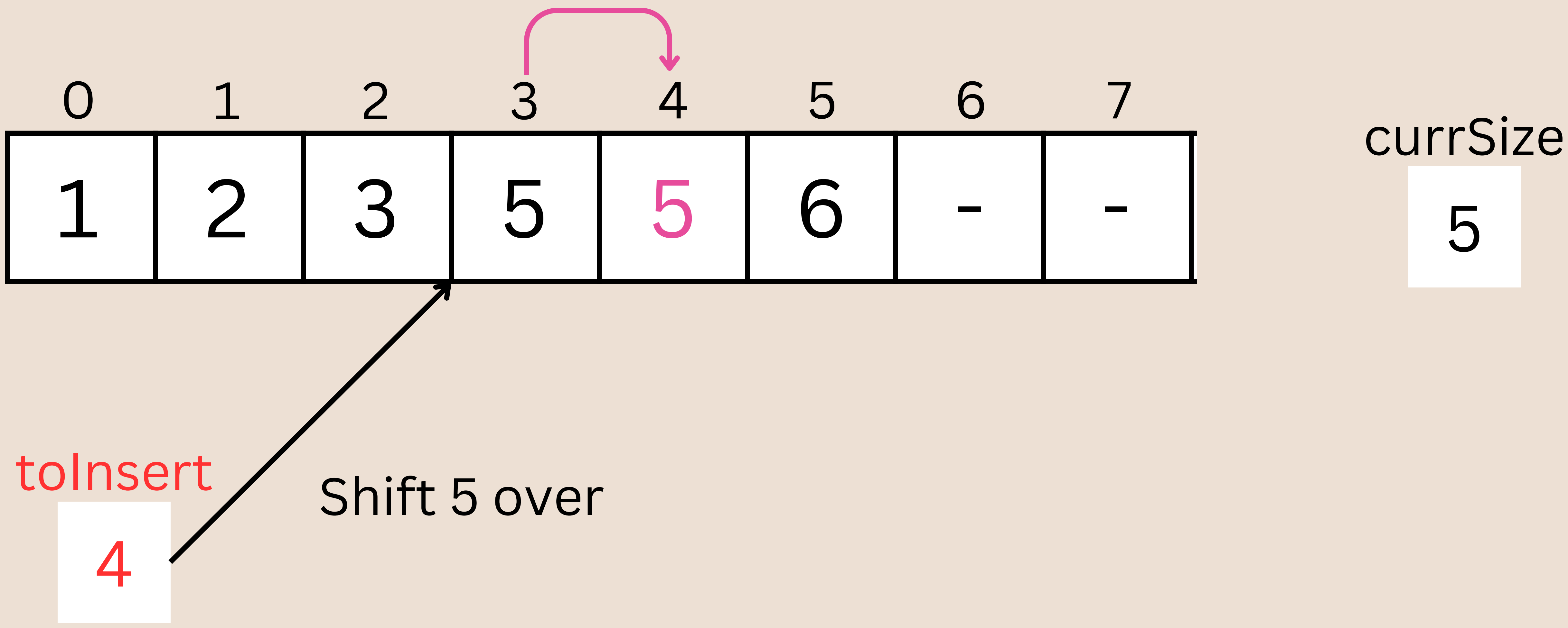
Keeping a sorted list

- We must now insert it into the proper spot to keep the array sorted (an “ordered insert” – slower, harder):



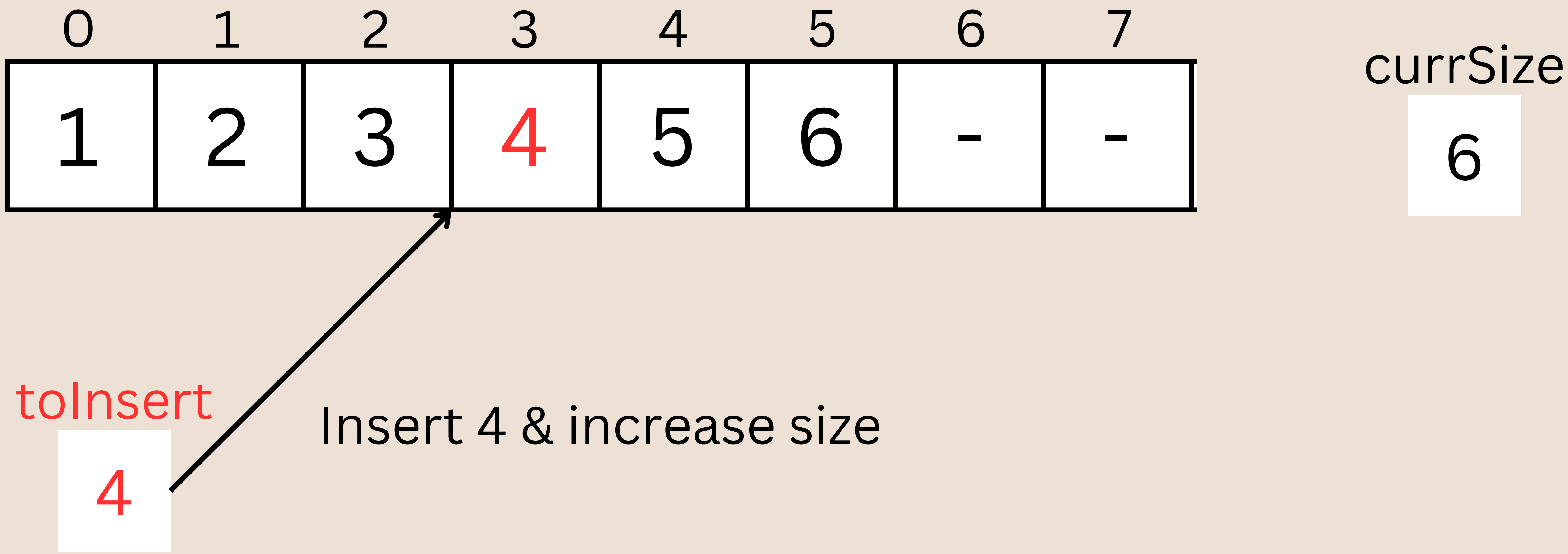
Keeping a sorted list

- We must now insert it into the proper spot to keep the array sorted (an “ordered insert” – slower, harder):



Keeping a sorted list

- We must now insert it into the proper spot to keep the array sorted (an “ordered insert” – slower, harder):



Ordered Insert

```
public static void OrderedInsert(array, value) {
    // Assume array is a sorted list with at least one empty space at the end
    // value is the new element to be inserted in the sorted order
    // assuming there is room to add a new value
    position = numArrayItems - 1  // Start from the last element in the array

    // Loop backwards through the array
    While position >= 0 and array[position] > value {
        // Shift elements to the right
        array[position + 1] = array[position]
        position = position - 1
    }

    // Insert the new value
    array[position + 1] = value
    numArrayItems++
}
```

OrderedInsert Visualized

- Lets take some code and go visualize the process with PythonTutor
- See the OrderedInsert.java file for the code

So now, what about sorting an array?

- Well there are lots of ways but the first (and simplest given our knowledge base) is to make use of the OrderedInsert we just learned

- It's called **Insertion Sort**

- The main idea of these in-place sorting algorithms is to separate the array into two parts: a sorted part (generally at the beginning) and an unsorted part (generally at the end) and gradually increase the size of the sorted part until everything is sorted

InsertionSort

0	1	2	3	4	5	6	7
1	4	0	5	9	-2	3	21

Sorted
Unsorted

First element is sorted at the beginning because 1 element alone is “technically” sorted

InsertionSort

0	1	2	3	4	5	6	7
1	**4**	0	5	9	-2	3	21

Sorted
Unsorted

Grab **first element** of unsorted part and do an ordered insert into the sorted part

InsertionSort

0	1	2	3	4	5	6	7
1	4	0	5	9	-2	3	21

Sorted
Unsorted

InsertionSort

0	1	2	3	4	5	6	7
1	4	**0**	5	9	-2	3	21

Sorted
Unsorted

nextUpToSort

0

InsertionSort

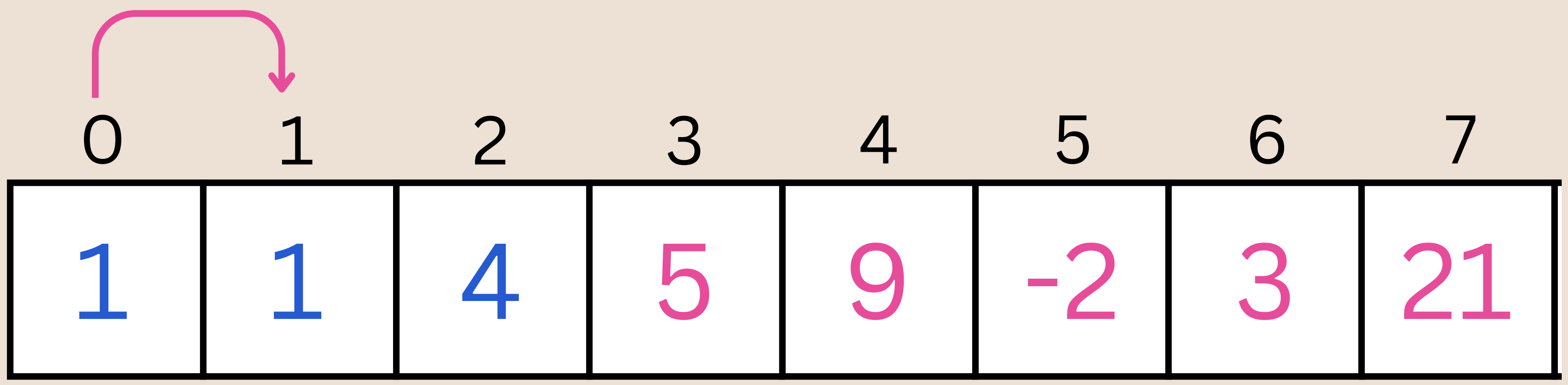
0	1	2	3	4	5	6	7
1	4	4	5	9	-2	3	21

Sorted
Unsorted

nextUpToSort

0

InsertionSort



Sorted
Unsorted

nextUpToSort

0

InsertionSort

0	1	2	3	4	5	6	7
0	1	4	5	9	-2	3	21

Sorted
Unsorted

nextUpToSort

0

InsertionSort

0	1	2	3	4	5	6	7
0	1	4	**5**	9	-2	3	21

Sorted
Unsorted

nextUpToSort

5

InsertionSort

0	1	2	3	4	5	6	7
0	1	4	5	**9**	-2	3	21

Sorted
Unsorted

nextUpToSort

9

InsertionSort

0	1	2	3	4	5	6	7
0	1	4	5	9	**-2**	3	21

Sorted
Unsorted

nextUpToSort

-2

InsertionSort

0	1	2	3	4	5	6	7
0	1	4	5	9	9	3	21

Sorted

Unsorted

nextUpToSort

-2

InsertionSort

0	1	2	3	4	5	6	7
0	1	4	5	5	9	3	21

Sorted

Unsorted

nextUpToSort

-2

InsertionSort

0	1	2	3	4	5	6	7
0	1	4	4	5	9	3	21

Sorted

Unsorted

nextUpToSort

-2

InsertionSort

0	1	2	3	4	5	6	7
0	1	1	4	5	9	3	21

Sorted
Unsorted

nextUpToSort

-2

InsertionSort

0	1	2	3	4	5	6	7
0	0	1	4	5	9	3	21

Sorted

Unsorted

nextUpToSort

-2

InsertionSort

0	1	2	3	4	5	6	7
-2	0	1	4	5	9	3	21

Sorted

Unsorted

nextUpToSort

-2

InsertionSort

0	1	2	3	4	5	6	7
-2	0	1	4	5	9	**3**	21

Sorted

Unsorted

nextUpToSort

3

InsertionSort

0	1	2	3	4	5	6	7
-2	0	1	**3**	4	5	9	**21**

Sorted
Unsorted

nextUpToSort

21

InsertionSort

0	1	2	3	4	5	6	7
-2	0	1	3	4	5	9	**21**

Sorted
Unsorted

nextUpToSort

21 That was the last element, we are sorted

InsertionSort

```
Method InsertionSort(Array)
    // Assume Array is an array of numbers

    For i = 1 To Length(Array) - 1
        // Select the element to be inserted
        currentValue = Array[i]
        position = i

        // Shift elements of the sorted segment of the array to the right
        // to create the correct position for the currentValue
        While position > 0 and Array[position - 1] > currentValue
            Array[position] = Array[position - 1]
            position = position - 1
        EndWhile

        // Insert the current element into its correct position
        Array[position] = currentValue
    EndFor
EndMethod
```

Inner While loop we have seen before (orderedInsert)
Outer for loop does the orderedInsert for each element (started at the second (index 0 is already “sorted”)

InsertionSort: Basic Worst Case Analysis

- What is the worst possible running time for this algorithm?
- i.e. What is the most number of operations we could possibly require (in terms of ‘n’) in order to have successfully sorted with insertion sort?

- remember that linear search was, worst case, n operations for n elements?
  - Often written as O(n)
  - this meant as n got bigger, the largest number of possible operations grew at the same rate (rate of n)

- Well for insertion sort
  - for every element in the array (for all n elements)
  - WORST POSSIBLE SCENARIO: we compare to every other element in the list EVERY SINGLE TIME

InsertionSort: Basic Worst Case Analysis

- The outer loop runs for every element (except the first)
  - while technically this is n-1, we just say n (removing the constants)
- The inner orderedInsert can also take up to n steps
  - technically the first time it does 1 step, the second time 2 steps, the third 3, etc.
  - this averages out to n/2 (/2 is a constant) so we just say n steps again
- **this means n steps, n times (nxn). We call this a $O(n^2)$ algorithm**

- **sorted array in reverse order actually takes n-squared steps**

Another sorting Algorithm

- **Selection Sort**
- (worst-case scenario) is the same (n-squared)
    - even though sometimes selection sort is better than insertion sort, or vice versa
    - I will explain why after we see the algorithm

SelectionSort

0	1	2	3	4	5	6	7
1	4	0	5	9	**-2**	3	21

Sorted
Unsorted

Find Minimum Value in unsorted portion

SelectionSort

0	1	2	3	4	5	6	7
1	4	0	5	9	**-2**	3	21

Sorted
Unsorted

Swap with first spot of unsorted portion to become part of sorted

SelectionSort

0	1	2	3	4	5	6	7
-2	4	0	5	9	**1**	3	21

Sorted
Unsorted

Now sorted part is + 1 length and unsorted is -1 length

SelectionSort

0	1	2	3	4	5	6	7
-2	4	**0**	5	9	1	3	21

Sorted
Unsorted

Find Minimum Value in unsorted portion

SelectionSort

0	1	2	3	4	5	6	7
-2	4	**0**	5	9	1	3	21

Sorted

Unsorted

Swap with first spot of unsorted portion to become part of sorted

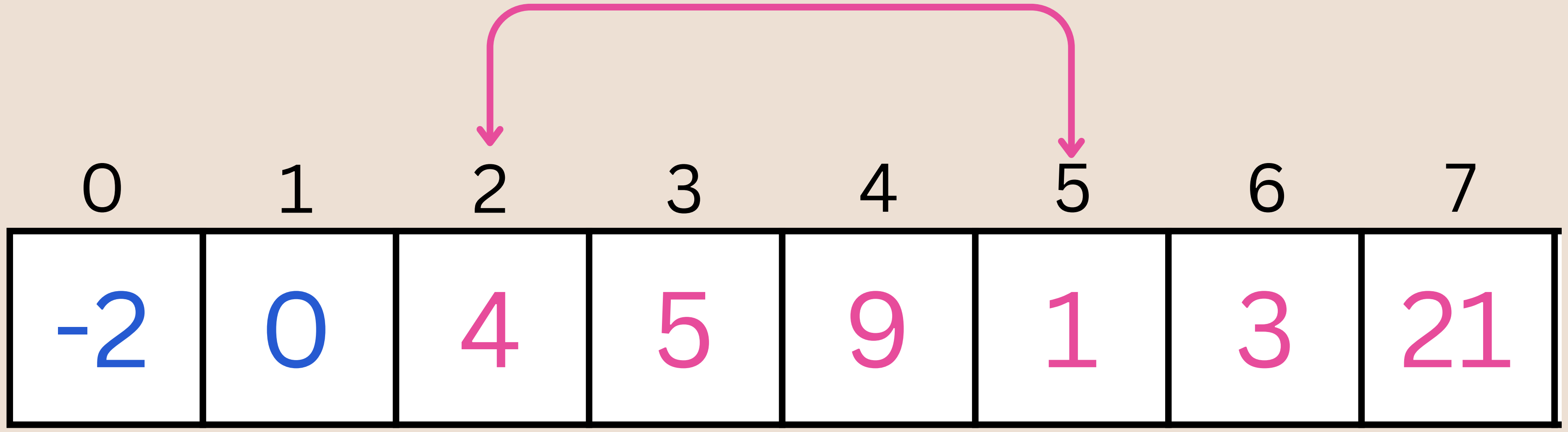
SelectionSort

0	1	2	3	4	5	6	7
-2	0	4	5	9	1	3	21

Sorted
Unsorted

Now sorted part is + 1 length and unsorted is -1 length

SelectionSort



Sorted

Unsorted

Continue until all blue

SelectionSort

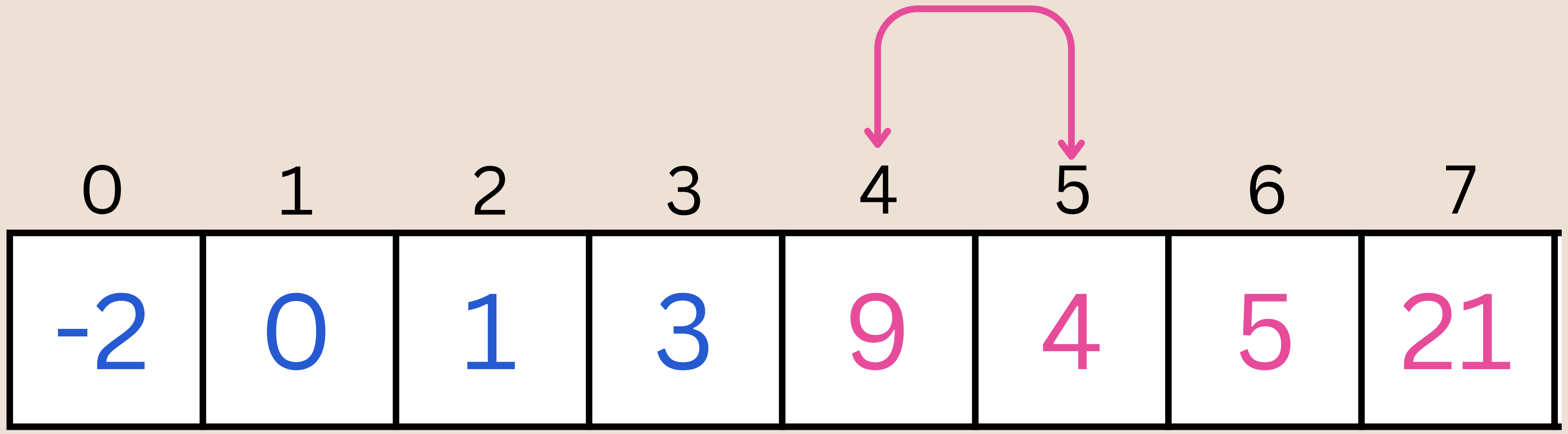
0	1	2	3	4	5	6	7
-2	0	1	5	9	4	3	21

Sorted

Unsorted

Continue until all blue

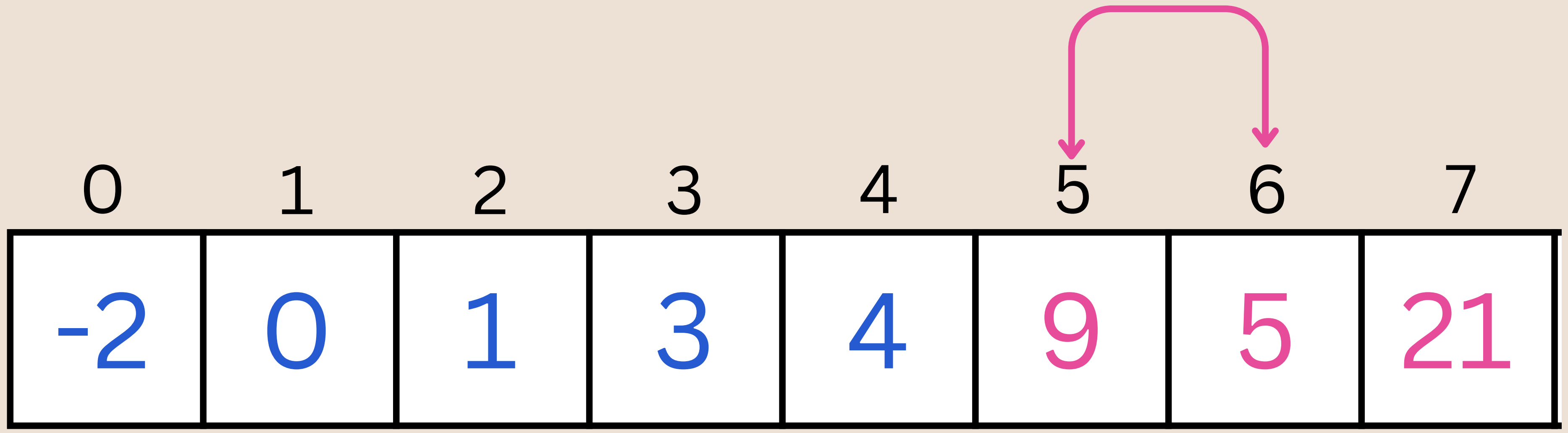
SelectionSort



Sorted
Unsorted

Continue until all blue

SelectionSort



Sorted
Unsorted

Continue until all blue

SelectionSort

0	1	2	3	4	5	6	7
-2	0	1	3	4	5	9	21

Sorted
Unsorted

Continue until all blue

SelectionSort

0	1	2	3	4	5	6	7
-2	0	1	3	4	5	9	21

Sorted

Unsorted

Continue until all blue

SelectionSort

0	1	2	3	4	5	6	7
-2	0	1	3	4	5	9	21

Sorted
Unsorted

Continue until all blue

SelectionSort

```
Procedure SelectionSort(Array)
    // Assume Array is an array of numbers

    For i = 0 To Length(Array) - 2
        // Set the current position as the minimum
        minIndex = i

        // Find the minimum element in the remaining unsorted array
        For j = i + 1 To Length(Array) - 1
            If Array[j] < Array[minIndex] Then
                minIndex = j
            EndIf
        EndFor

        // Swap the found minimum element with the first element of the unsorted part
        If minIndex != i Then
            Swap Array[minIndex] and Array[i]
        EndIf
    EndFor
EndProcedure
```

Inner for loop is just doing a find Min for each element from the outer for loop. At the end of each inner loop we swap our found min into place.

Why are two n-squared algorithms not the same speed in practice?

- We can analyze our sorting algorithms by:
  - swaps (actually moving values from unsorted to sorted)
  - comparisons (comparing numbers with a given value)

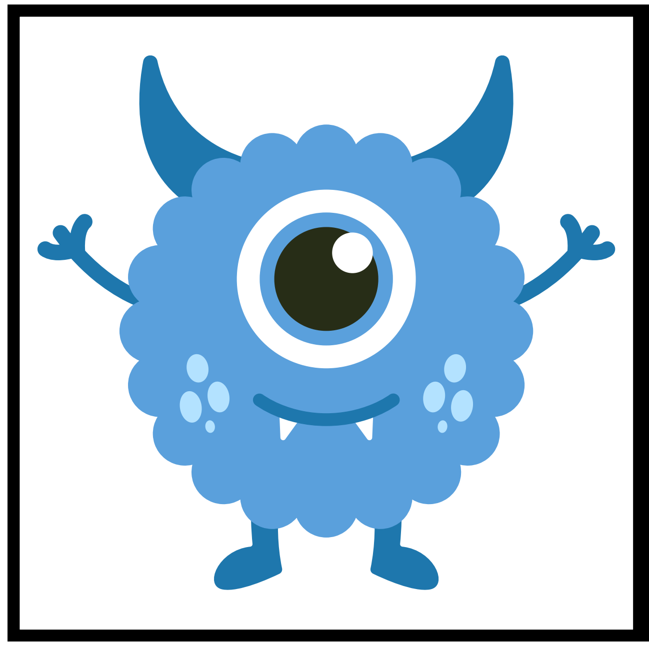
- Consider the following array [5,4,3,2,1]
  - Selection sort requires: **10 comparisons**
  - Insertion sort requires: **10 comparisons**
- Consider the following array [2,3,4,5,6,1]
  - Selection sort requires: **15 comparisons**
  - Insertion sort requires: **10 comparisons**

- We can also analyze these algorithms (or others) by time and space complexity(that big-O notation), and other ways too!

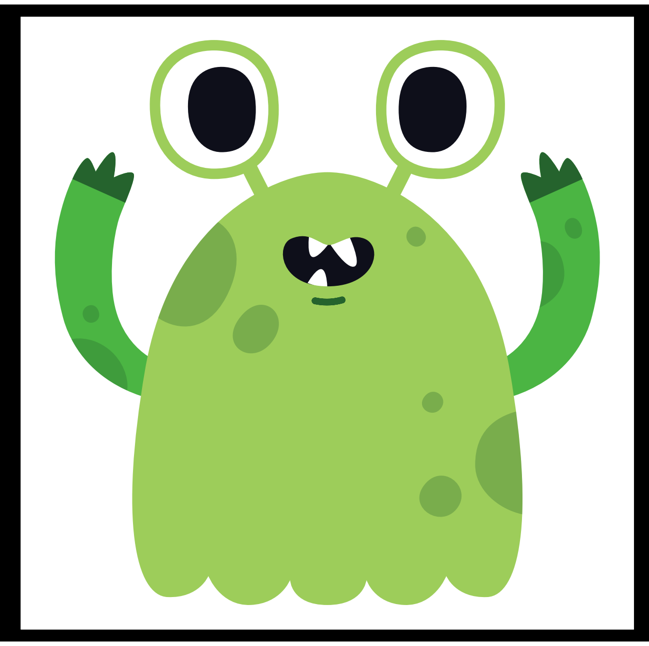
Another sorting Algorithm

- **Merge Sort**
- (worst-case scenario) much better than the other two algorithms (nlogn)
    - We split our array into two subarrays until we only have 1-2 element left
        - 1 element array = already sorted
        - 2 element array (swap or no swap)
    - Then we merge these sorted sub arrays back up into one large sorted array
    - over and over until we are done!
- We will just look at the algorithm in pictures **however**
- **Given two sorted arrays, you should be able to merge them into one sorted array**

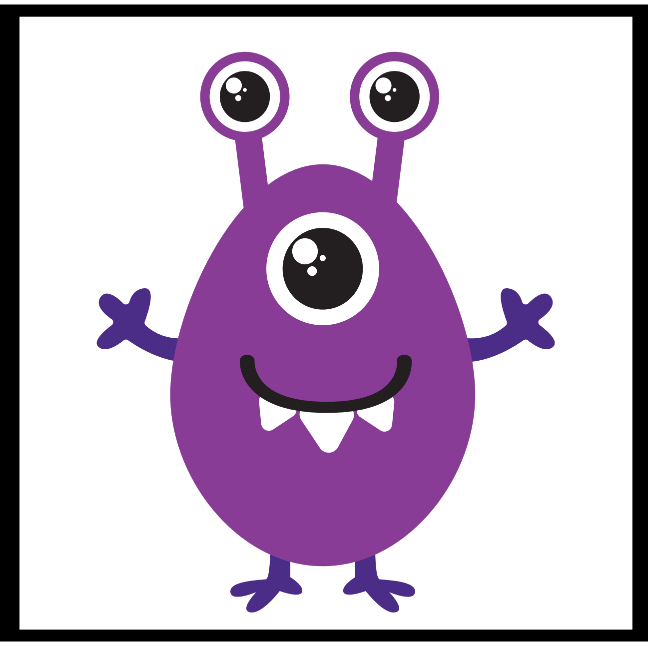
MergeSorting Monsters by their Cuteness Rankings



Name: **Chickee**
Cuteness[1-10]: **4**



Name: **Goosh**
Cuteness[1-10]: **4**

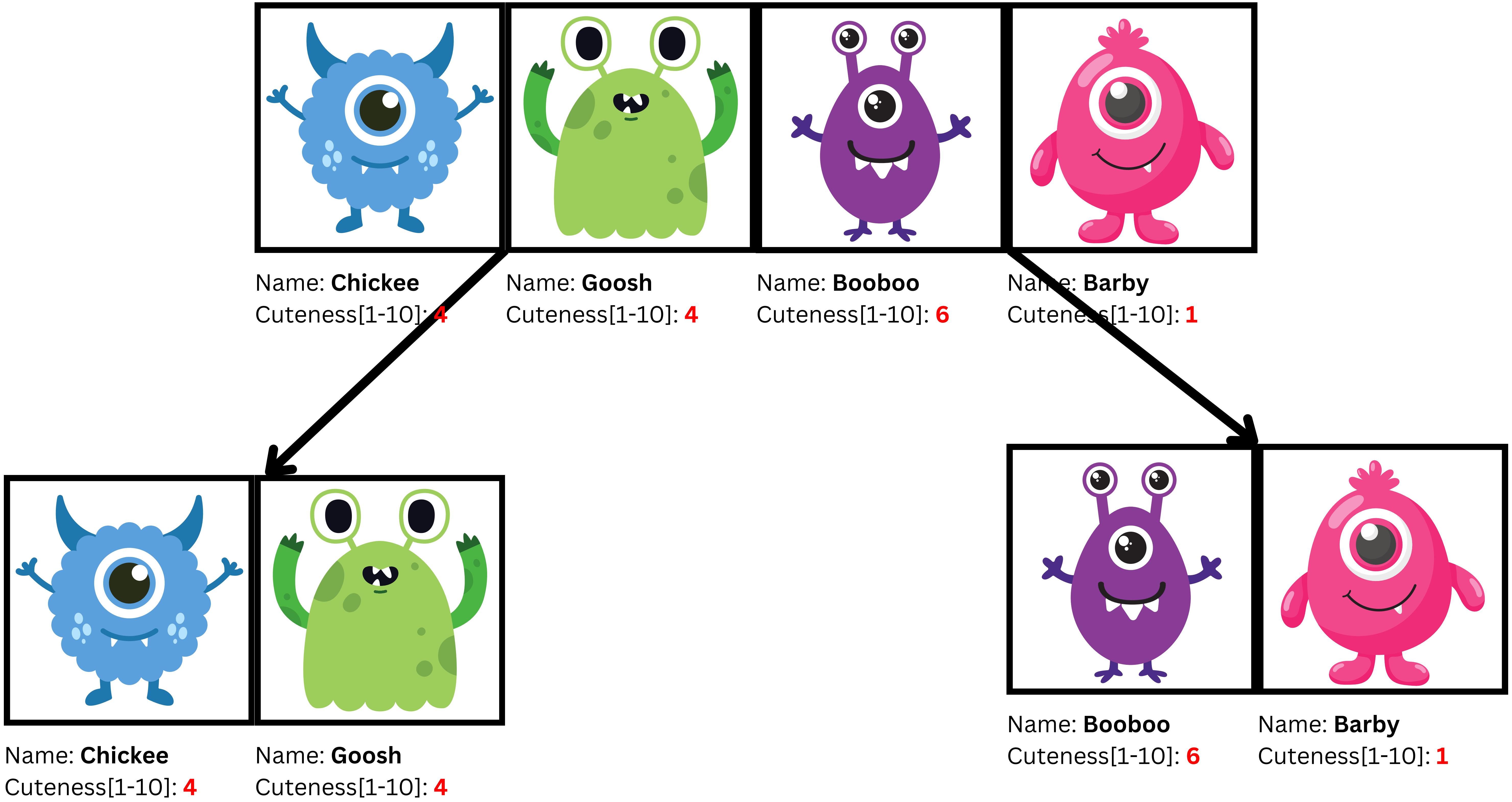


Name: **Booboo**
Cuteness[1-10]: **6**

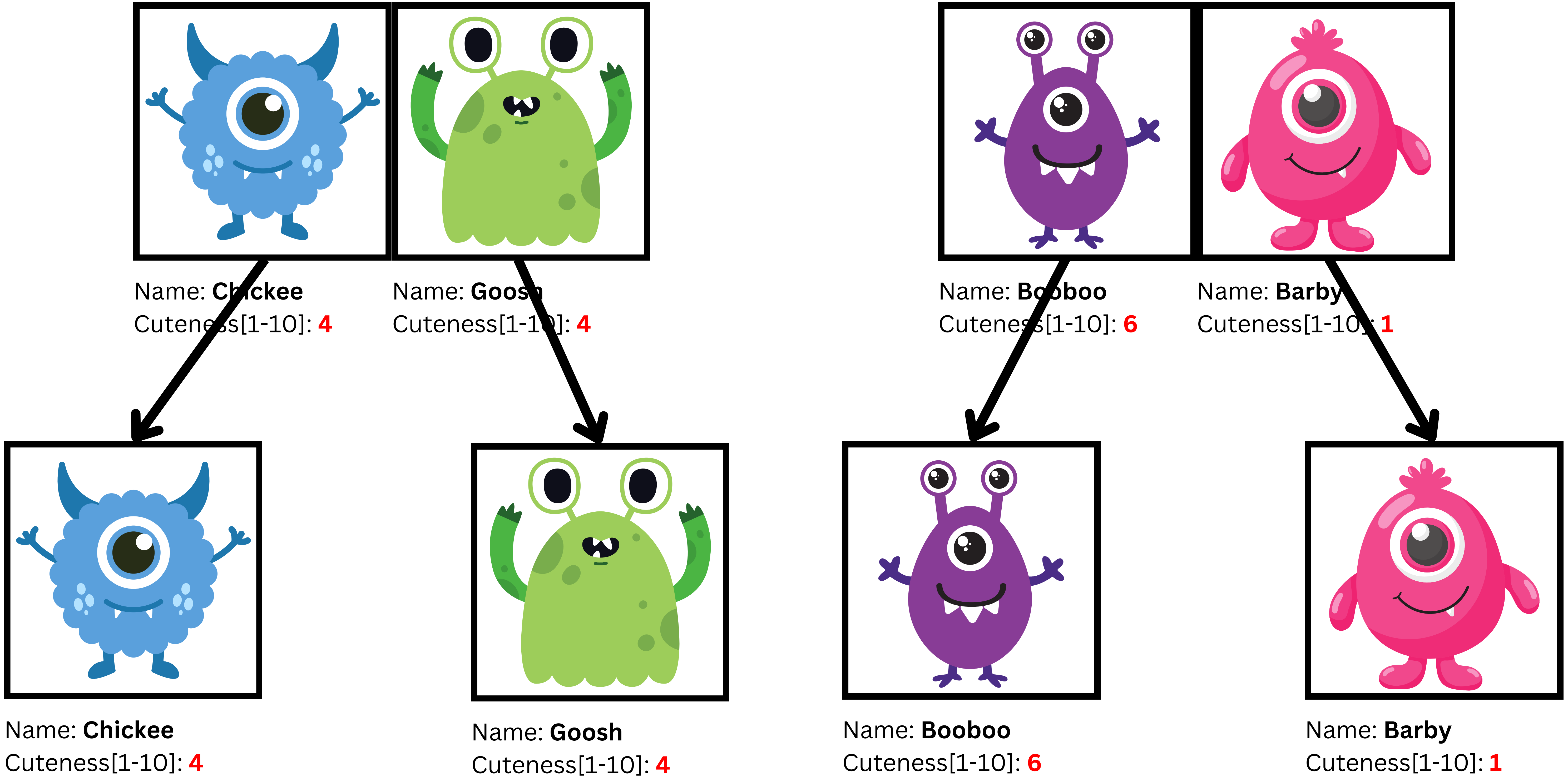
Name: **Barby**
Cuteness[1-10]: **1**

MergeSort(Cuteness)

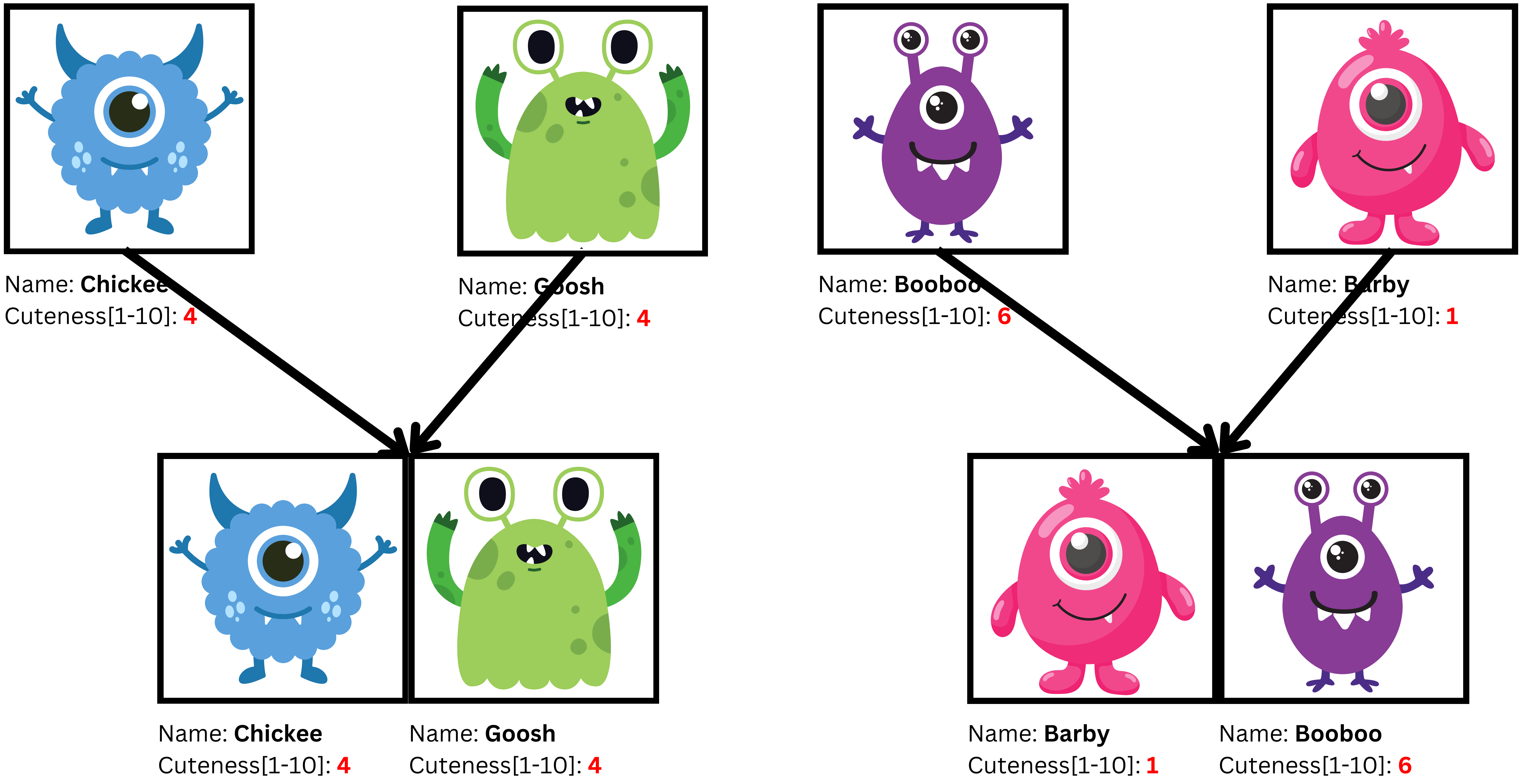
MergeSorting Monsters by their Cuteness Rankings



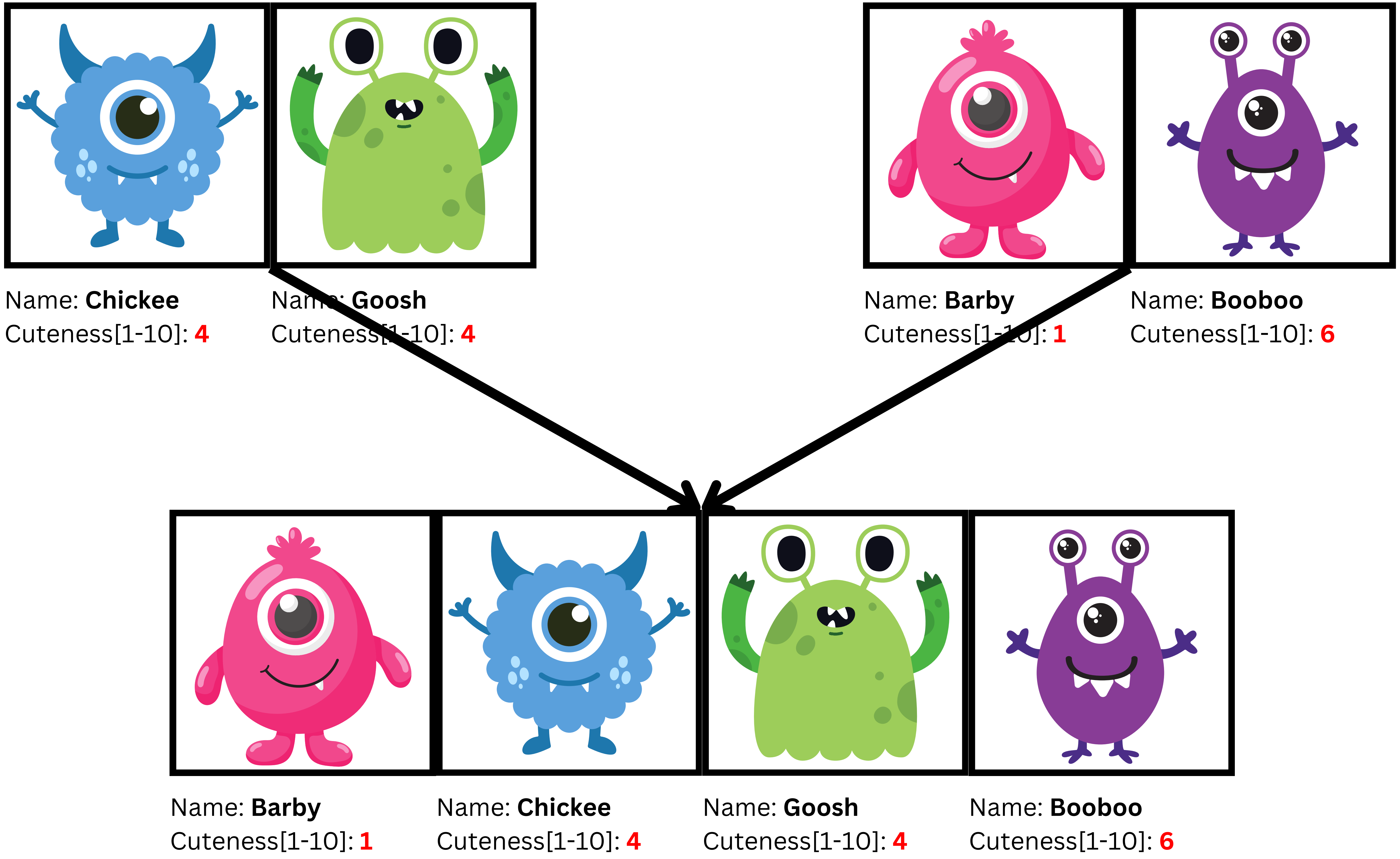
MergeSorting Monsters by their Cuteness Rankings



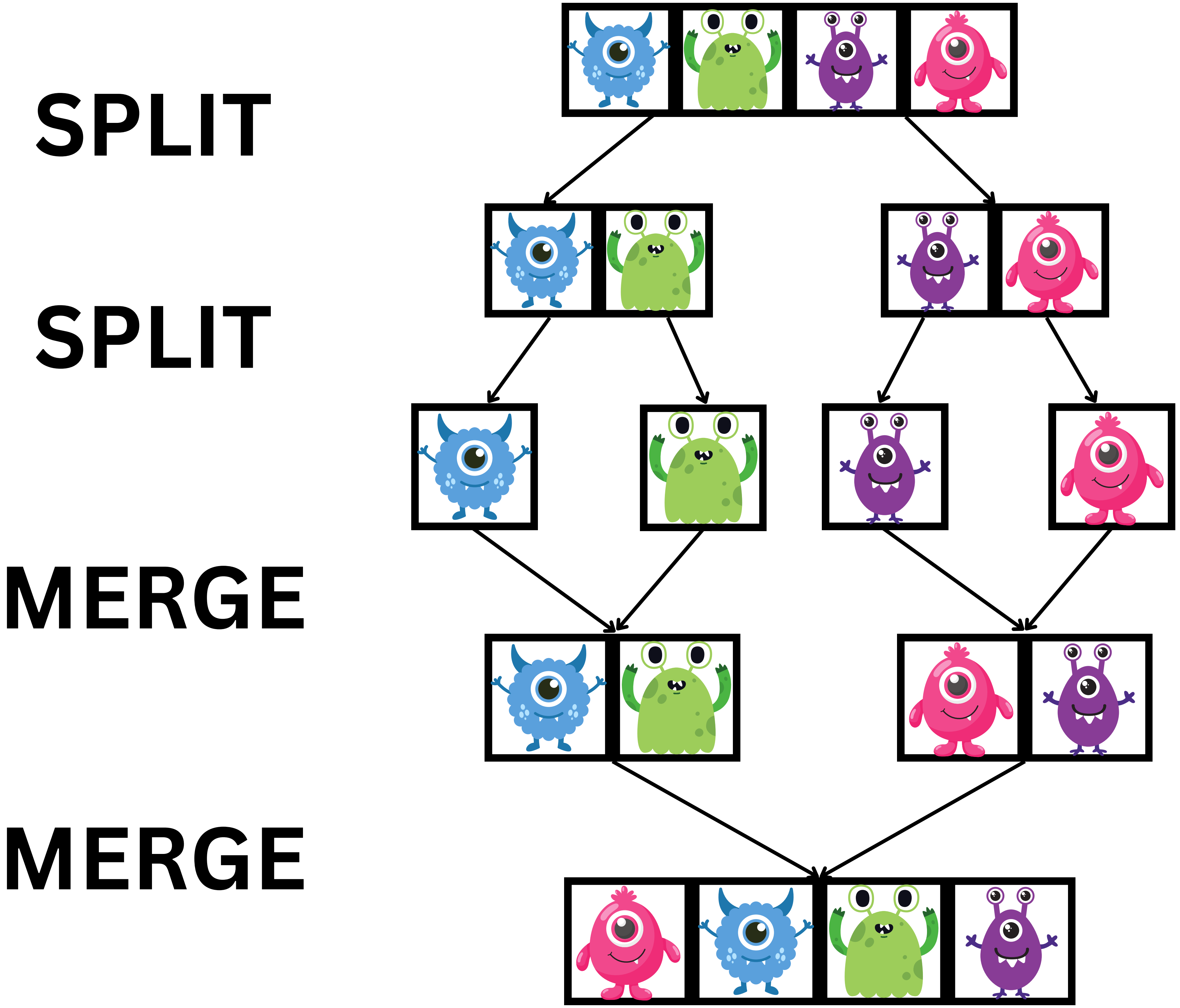
MergeSorting Monsters by their Cuteness Rankings



MergeSorting Monsters by their Cuteness Rankings



MergeSorting Monsters by their Cuteness Rankings



Pause & Practice (the merge part of merge sort)

- Given two sorted arrays, you should be able to merge them into one sorted array

```
public class Test {
  public static void main(String []args) {
    int[] array1 = [1,2,3,5,6,9,10];
    int[] array2 = [-4,6,9,11,12,15];
    int[] merged = merge(array1,array2);
    // -4,1,2,3,4,5,5,9,10,11,12,15
  }

  public static int[] merge(int[] a1, int[] a2) {
    int[] arr = new int[a1.length + a2.length];
    /* You can do this part */
    return arr;
  }
}
```

Pause & Practice (Insertion & Selection Sort)

- Given an unsorted array
- int[] array = [1,4,8,2,6,-1,7,12,57,21,0,-1]
- Sort it using Insertion Sort & Selection Sort
- **HINT:** make sure you make a **deep copy** of the original unsorted array in your program so you can sort insertion and selection sort all in one program (if you want to)