ALOGRITHMS
SORTING & SEARCHING

ROWAN MEREWOOD

LET'S START WITH THE BIG PHILOSOPHICAL QUESTIONS

WHO AM I?

Software Engineer and Technical Team Lead

```
$inviqaGroup = ['Inviqa', 'Sensio Labs UK', 'Session Digital'];
```

```
$social = [
    'blog'     => 'http://merewood.org',
    'facebook' => $this->graphSearch('people i am stalking'),
    'github'   => 'https://github.com/rowan-m',
    'google+'  => 'https://plus.google.com/111813845727005160164',
    'identica' => 'http://identi.ca/rowanm',
    'imdb'     => 'http://imdb.com/name/nm1412348',
    'twitter'  => 'https://twitter.com/rowan_m',
];
```

WHY AM I HERE?

1. Because straight-up, pure computer science is **AWESOME**.
2. `^E0F`

Also, understanding the lower level detail helps you make better use of the higher level abstractions.

WHY SORT?

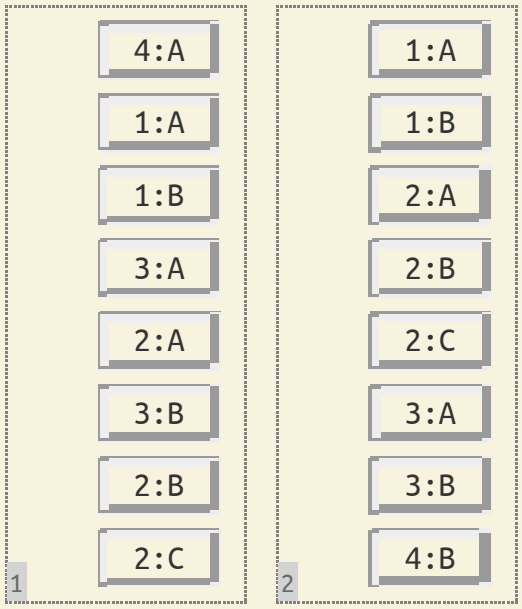
- Displaying lists to humans
- Categorising data
- Preparing data for merging
- Preparing data for searching

In general: to display or to prepare for another operation.

COMPARING ALGORITHMS

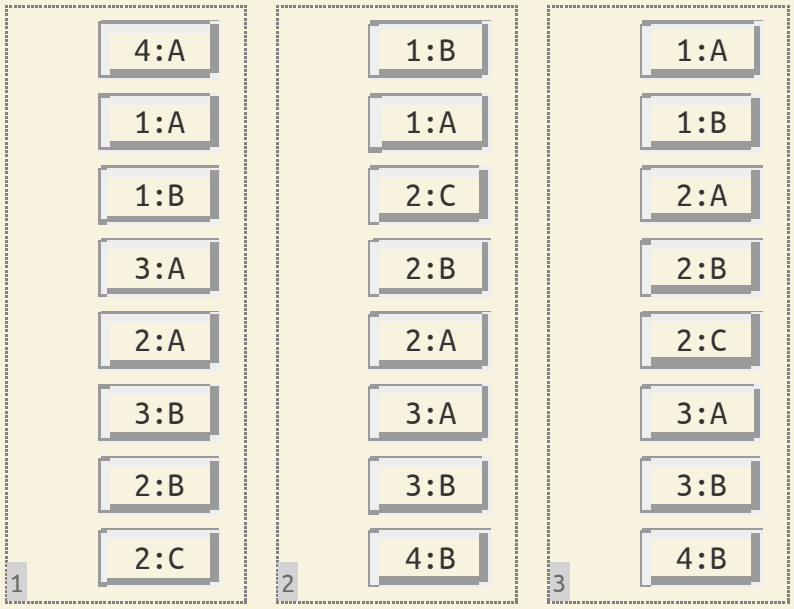
STABILITY

STABLE



Order of unsorted portion maintained

UNSTABLE



Order of unsorted portion may change

BIG O NOTATION

Rate of growth for resource usage based on the size of its input.

- Resource usage: *CPU cycles / time, memory usage*
- Size of input: *number of elements*

BIG O NOTATION *(CONT.)*

- $O(1)$: *Constant*
- $O(n)$: *Linear growth*
- $O(n \log n)$: *Logarithmic growth*
- $O(n^2)$: *Quadratic growth*

ADAPTABILITY

An adaptive algorithm has better performance when the list is already partially sorted

SORTING ALGORITHMS

INSERTION SORT

CODE

```
class InsertionSort {
    public function sort(array $elements) {
        $iterations = count($elements);
        for ($index = 1; $index < $iterations; $index++) {
            $elementToInsert = $elements[$index];
            $insertIndex = $index;

            while ($insertIndex > 0 && $elementToInsert < $elements[$insertIndex - 1]) {
                $elements[$insertIndex] = $elements[$insertIndex - 1];
                $elements[$insertIndex - 1] = $elementToInsert;
                $insertIndex--;
            }
        }
        return $elements;
    }
}
```

INSERTION SORT *(CONT.)*

CODE: ITERATE THROUGH THE LIST

```
public function sort(array $elements) {
    // At least one iteration per element
    $iterations = count($elements);

    for ($index = 1; $index < $iterations; $index++) {
        // If no other variable operations happen here:
        // algorithm is O(n)
    }
}
```

INSERTION SORT *(CONT.)*

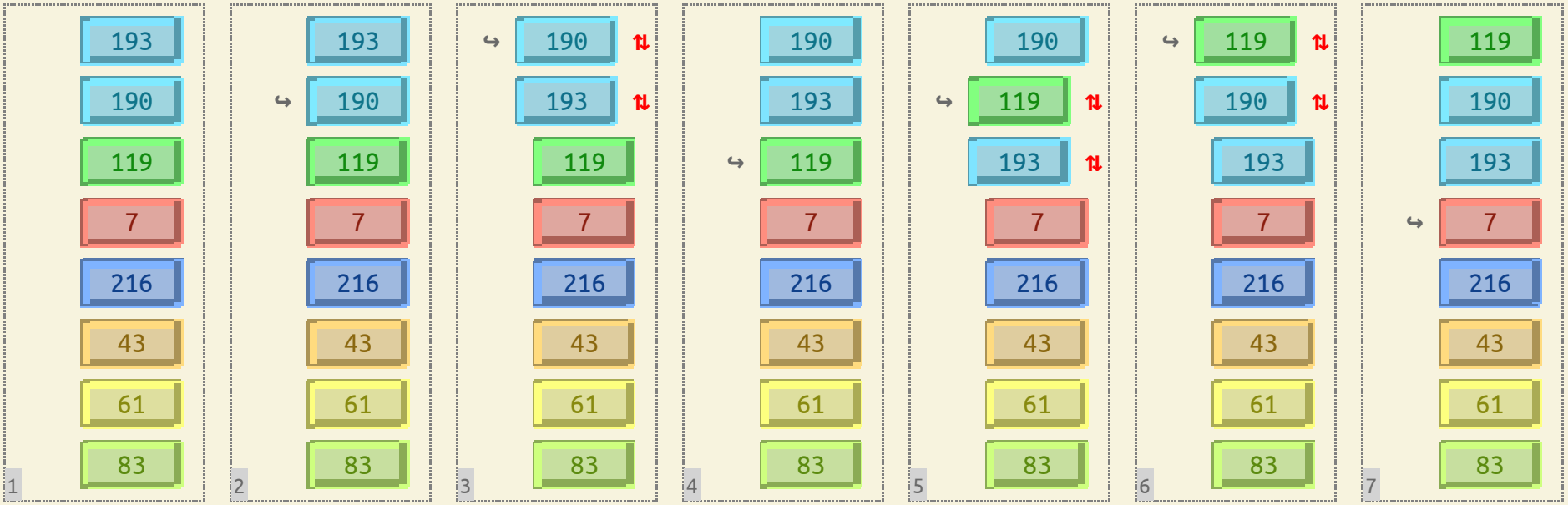
CODE: COMPARE ELEMENTS

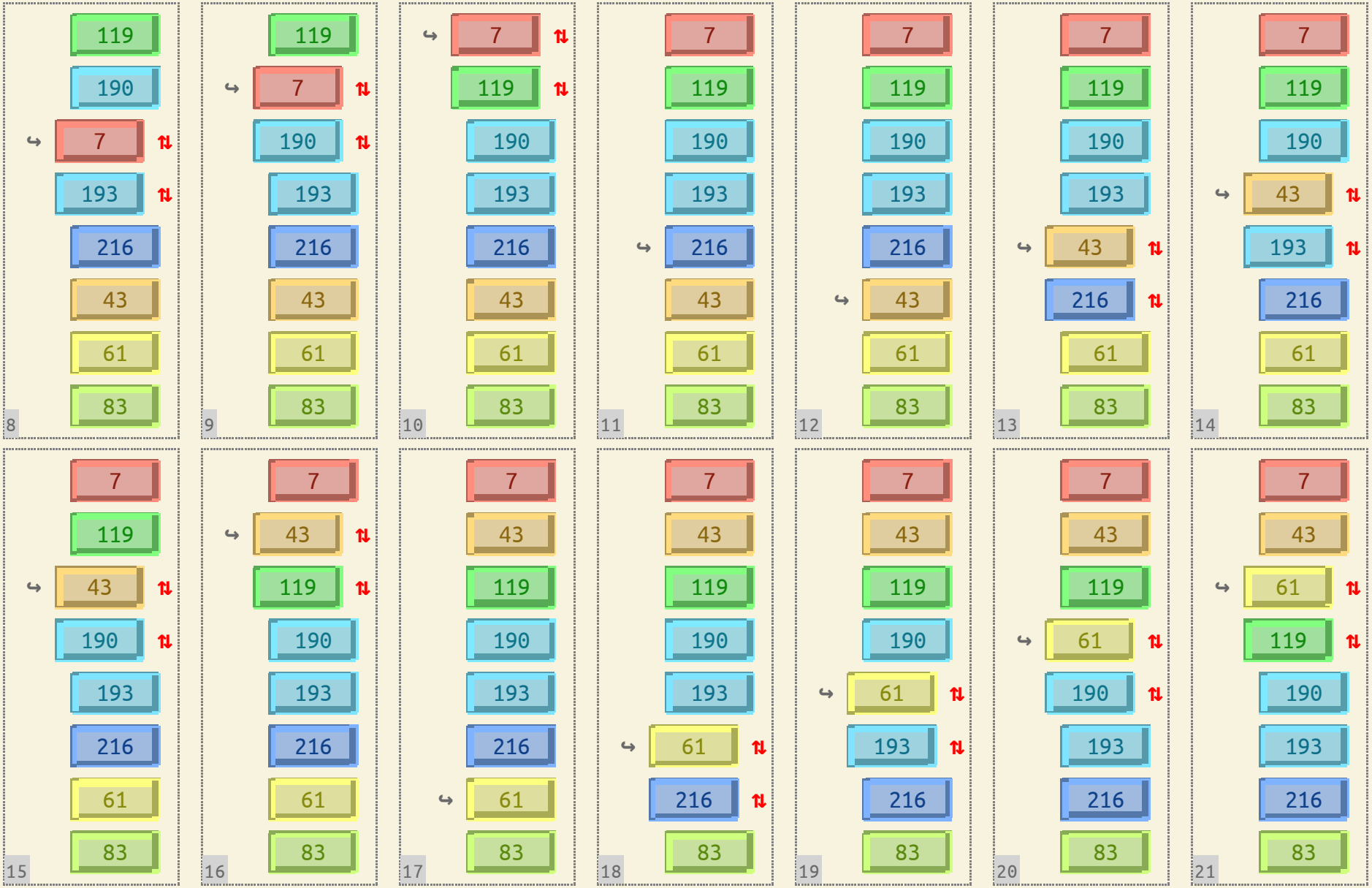
```
for ($index = 1; $index < $iterations; $index++) {
    // "Pick up" the current element and its position
    $elementToInsert = $elements[$index];
    $insertIndex = $index;

    // Iterate back through the elements
    // until the correct position it reached
    while ($insertIndex > 0 && $elementToInsert < $elements[$insertIndex - 1]) {
        // Swap out of order elements
        $elements[$insertIndex] = $elements[$insertIndex - 1];
        $elements[$insertIndex - 1] = $elementToInsert;
        $insertIndex--;
    }
}
```

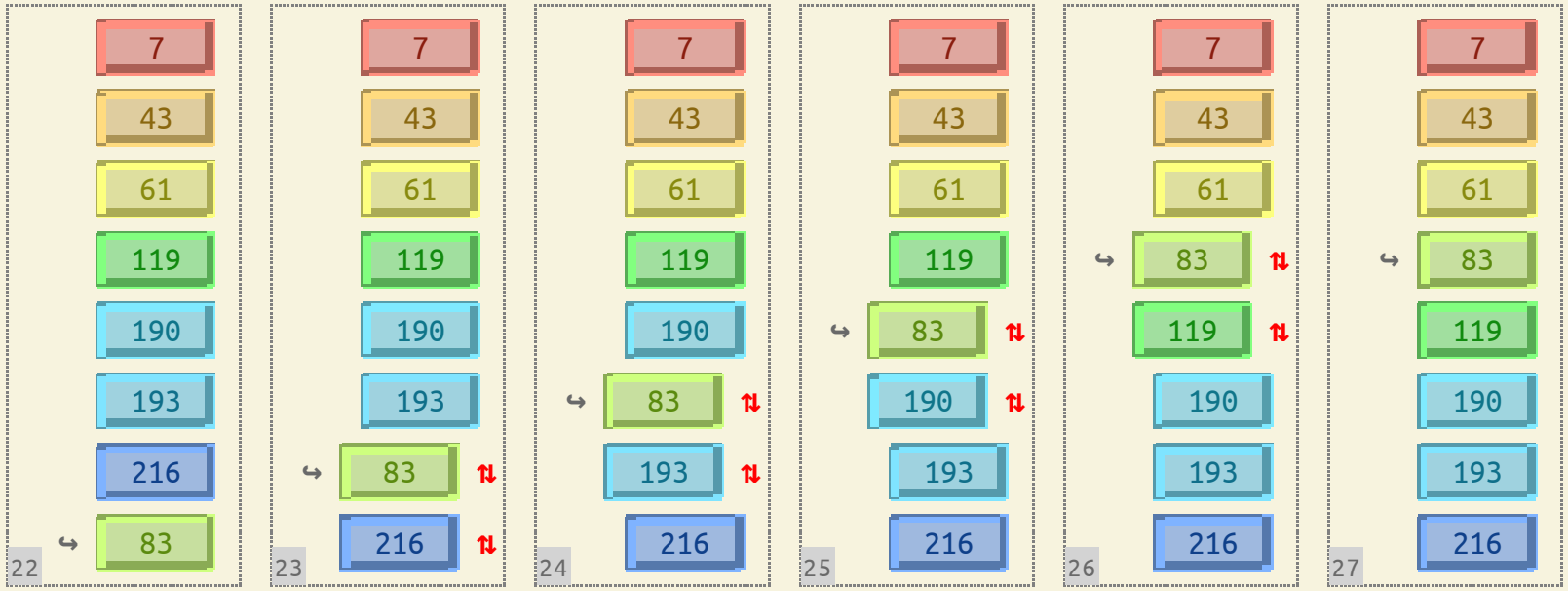
If the list is in order, the `while` loop is not entered.

INSERTION SORT *(CONT.)*

ITERATIONS




8
119
190
7
193
216
43
61
83
9
119
7
190
193
216
43
61
83
10
7
119
190
193
216
43
61
83
11
7
119
190
193
216
43
61
83
12
7
119
190
193
216
43
61
83
13
7
119
190
193
43
216
61
83
14
7
119
190
43
193
216
61
83
15
7
119
43
190
193
216
61
83
16
7
43
119
190
193
216
61
83
17
7
43
119
190
193
216
61
83
18
7
43
119
190
193
61
216
83
19
7
43
119
190
61
193
216
83
20
7
43
119
61
190
193
216
83
21
7
43
61
119
190
193
216
83


7
43
61
119
190
193
216
83
22
7
43
61
119
190
193
83
216
23
7
43
61
119
190
83
193
216
24
7
43
61
119
83
190
193
216
25
7
43
61
83
119
190
193
216
26
7
43
61
83
119
190
193
216
27

Sorted!

INSERTION SORT *(CONT.)*

SUMMARY

- Best case: $O(n)$
- Average / worst case: $O(n^2)$
- Memory usage: $O(n)$
- Adaptive, stable, in place, and on line(n)

BUBBLE SORT

CODE

```
class BubbleSort {
    public function sort(array $elements) {
        for ($index = count($elements); $index > 0; $index--) {
            $swapped = false;
            for ($swapIndex = 0; $swapIndex < $index - 1; $swapIndex++) {
                if ($elements[$swapIndex] > $elements[$swapIndex + 1]) {
                    $tmp = $elements[$swapIndex];
                    $elements[$swapIndex] = $elements[$swapIndex + 1];
                    $elements[$swapIndex + 1] = $tmp;
                    $swapped = true;
                }
            }
            if (!$swapped) { return $elements; }
        }
    }
}
```

BUBBLE SORT *(CONT.)*

CODE: ITERATE THROUGH THE LIST

```
// Iterate through the elements
for ($index = count($elements); $index > 0; $index--) {
    $swapped = false;
    // Swap out of order elements
    // until there's nothing left to swap
    if (!$swapped) { return $elements; }
}
```

BUBBLE SORT *(CONT.)*

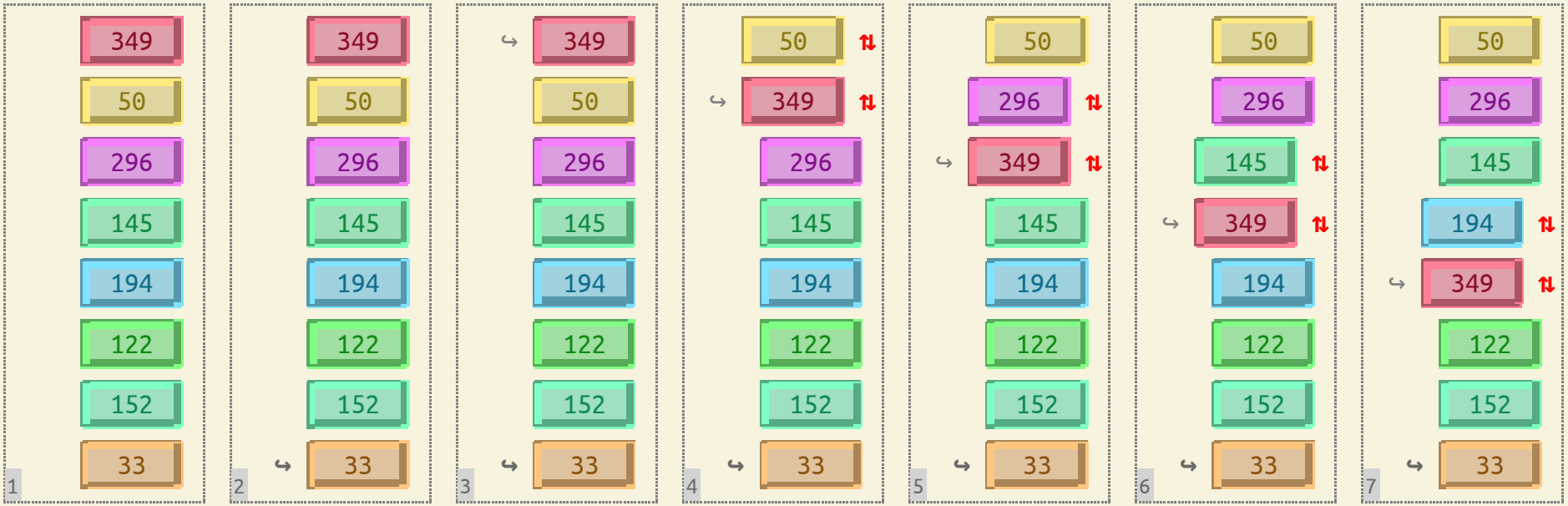
CODE

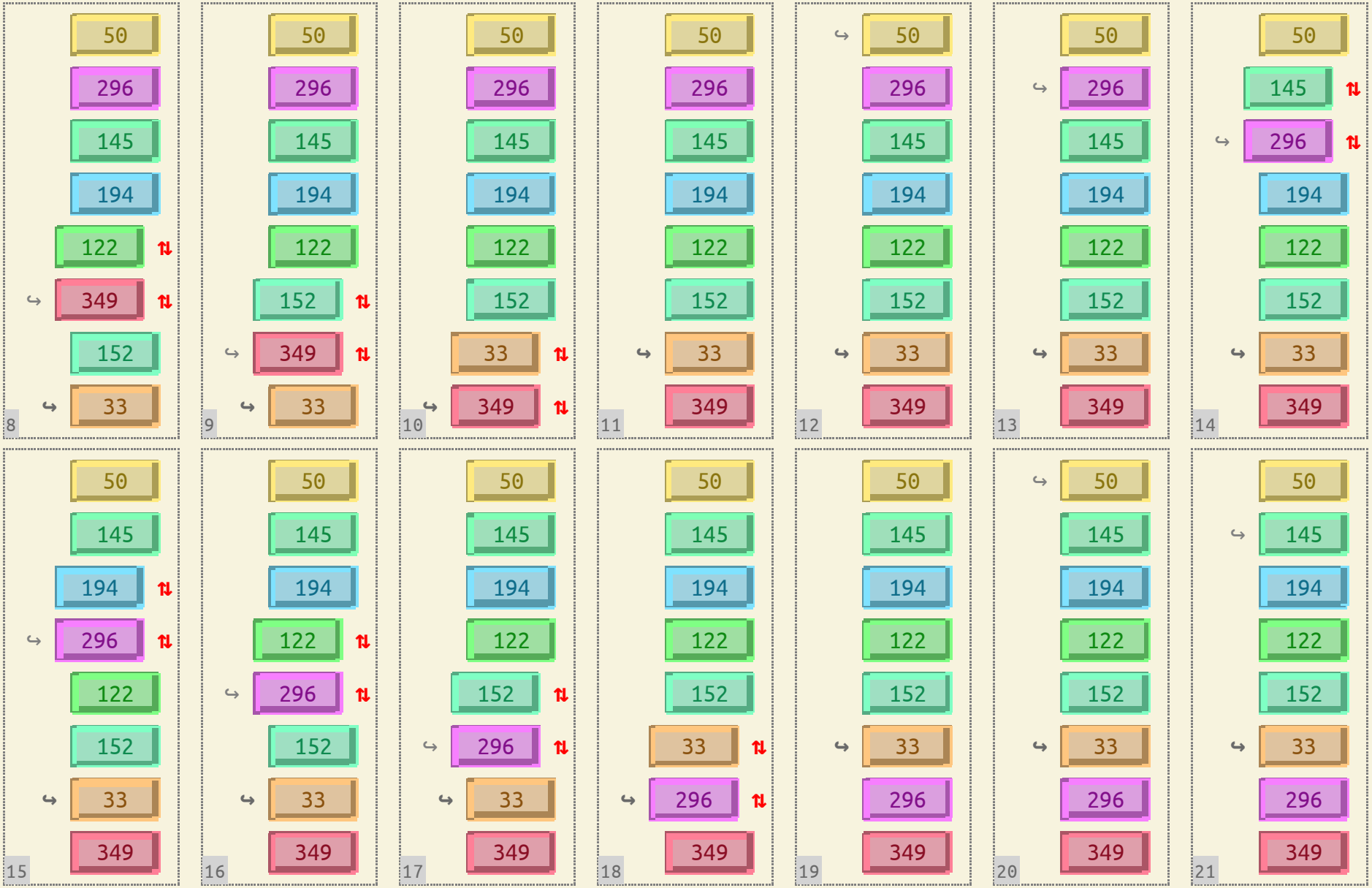
```
for ($index = count($elements); $index > 0; $index--) {
    $swapped = false;
    // Iterate through the unsorted portion of the list
    for ($swapIndex = 0; $swapIndex < $index - 1; $swapIndex++) {
        // Compare and swap elements
        if ($elements[$swapIndex] > $elements[$swapIndex + 1]) {
            $tmp = $elements[$swapIndex];
            $elements[$swapIndex] = $elements[$swapIndex + 1];
            $elements[$swapIndex + 1] = $tmp;
            $swapped = true;
        }
    }
    if (!$swapped) { return $elements; }
}
```

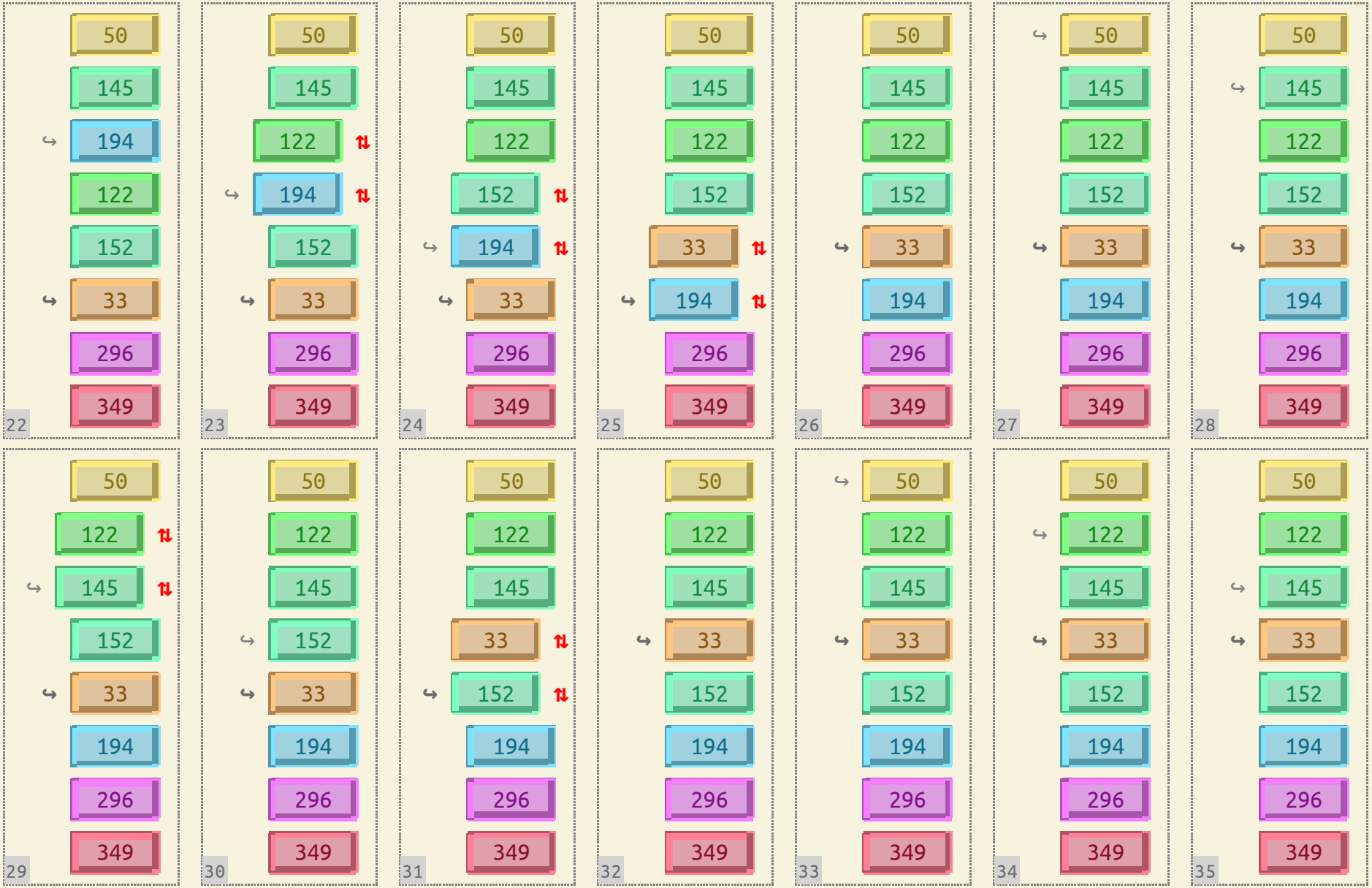
If the list is in order, then `$swapped` stays `false`.

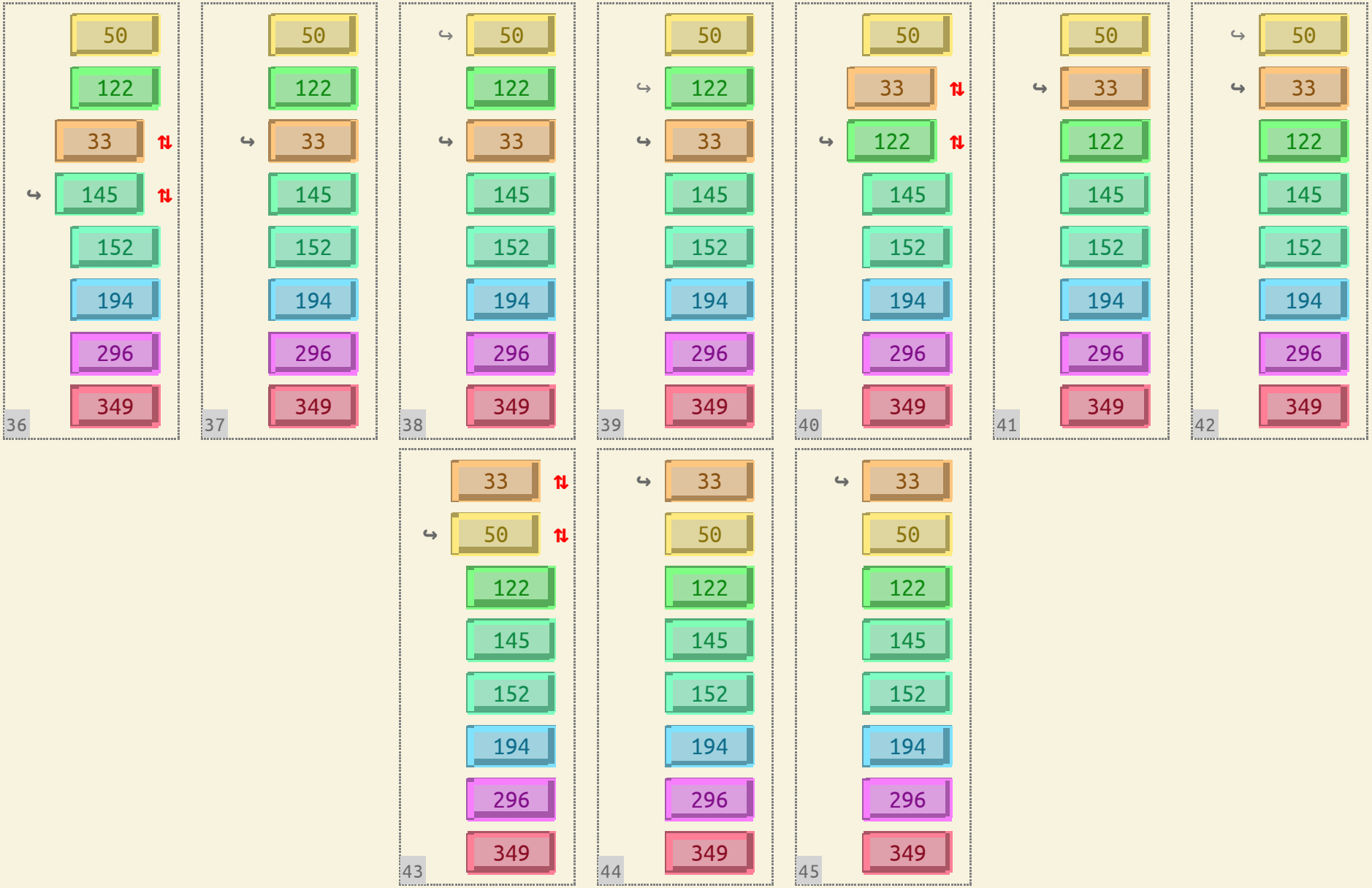
BUBBLE SORT *(CONT.)*

ITERATIONS




50
296
145
194
122
349
152
33
8
50
296
145
194
122
152
349
33
9
50
296
145
194
122
152
33
349
10
50
296
145
194
122
152
33
349
11
50
296
145
194
122
152
33
349
12
50
296
145
194
122
152
33
349
13
50
145
296
194
122
152
33
349
14
50
145
194
296
122
152
33
349
15
50
145
194
122
296
152
33
349
16
50
145
194
122
152
296
33
349
17
50
145
194
122
152
33
296
349
18
50
145
194
122
152
33
296
349
19
50
145
194
122
152
33
296
349
20
50
145
194
122
152
33
296
349
21


50
145
194
122
152
33
296
349
22
50
145
122
194
152
33
296
349
23
50
145
122
152
194
33
296
349
24
50
145
122
152
33
194
296
349
25
50
145
122
152
33
194
296
349
26
50
145
122
152
33
194
296
349
27
50
145
122
152
33
194
296
349
28
50
122
145
152
33
194
296
349
29
50
122
145
152
33
194
296
349
30
50
122
145
33
152
194
296
349
31
50
122
145
33
152
194
296
349
32
50
122
145
33
152
194
296
349
33
50
122
145
33
152
194
296
349
34
50
122
145
33
152
194
296
349
35



Sorted!

BUBBLE SORT *(CONT.)*

SUMMARY

- Best case: $O(n)$
- Average / worst case: $O(n^2)$
- Memory usage: $O(n)$

BUBBLE SORT *(CONT.)*

THE UGLY KNUTH

> *"The bubble sort seems to have nothing to recommend it, except a catchy name and the fact that it leads to some interesting theoretical problems"*

QUICK SORT

CODE

```
class QuickSort {
    public function sort(array $elements) {
        $this->doQuickSort($elements, 0, count($elements) - 1);
        return $elements;
    }

    function doQuickSort($elements, $leftIndex, $rightIndex) {
        // Divide the array in two, creating a “pivot” value
        // Move any value lower than the pivot to the left array
        // Move any value higher than the pivot to the right array
        // Recursively repeat the same operation on both arrays
    }
}
```

QUICK SORT *(CONT.)*

CODE: CREATE A PIVOT

```
function doQuickSort($elements, $leftIndex, $rightIndex) {
    // Divide the array in two, creating a “pivot” value
    $pivotIndex = ceil($leftIndex + (($rightIndex - $leftIndex) / 2));
    $pivotElement = $elements[$pivotIndex];
    $leftSwapIndex = $leftIndex;
    $rightSwapIndex = $rightIndex;
    while ($leftSwapIndex <= $rightSwapIndex) {
        // Move the left index until we find an out of order element
        // Move the right index until we find an out of order element
        // Swap them
    }
}
```

QUICK SORT *(CONT.)*

CODE: SWAP VALUES

```
while ($leftSwapIndex <= $rightSwapIndex) {
    // Move the left index until we find an out of order element
    while ($elements[$leftSwapIndex] < $pivotElement) { $leftSwapIndex++; }
    // Move the right index until we find an out of order element
    while ($elements[$rightSwapIndex] > $pivotElement) { $rightSwapIndex--; }
    // Swap them
    if ($leftSwapIndex <= $rightSwapIndex) {
        $tmp = $elements[$leftSwapIndex];
        $elements[$leftSwapIndex] = $elements[$rightSwapIndex];
        $elements[$rightSwapIndex] = $tmp;
        $leftSwapIndex++;
        $rightSwapIndex--;
    }
}
```

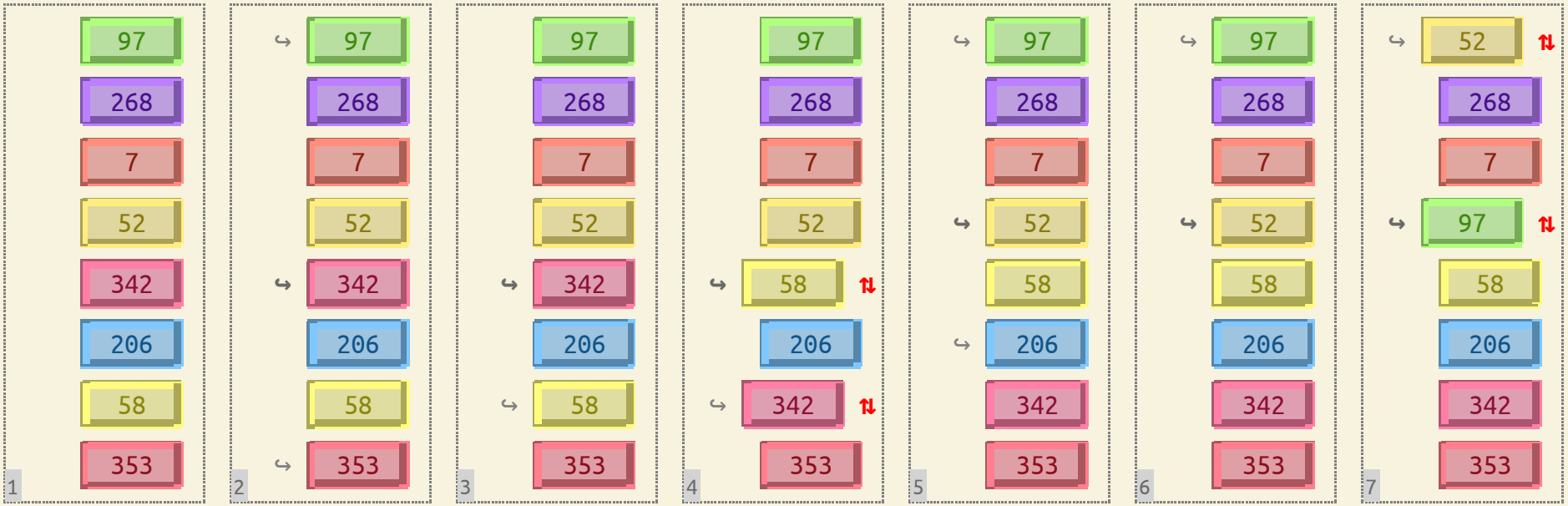
QUICK SORT *(CONT.)*

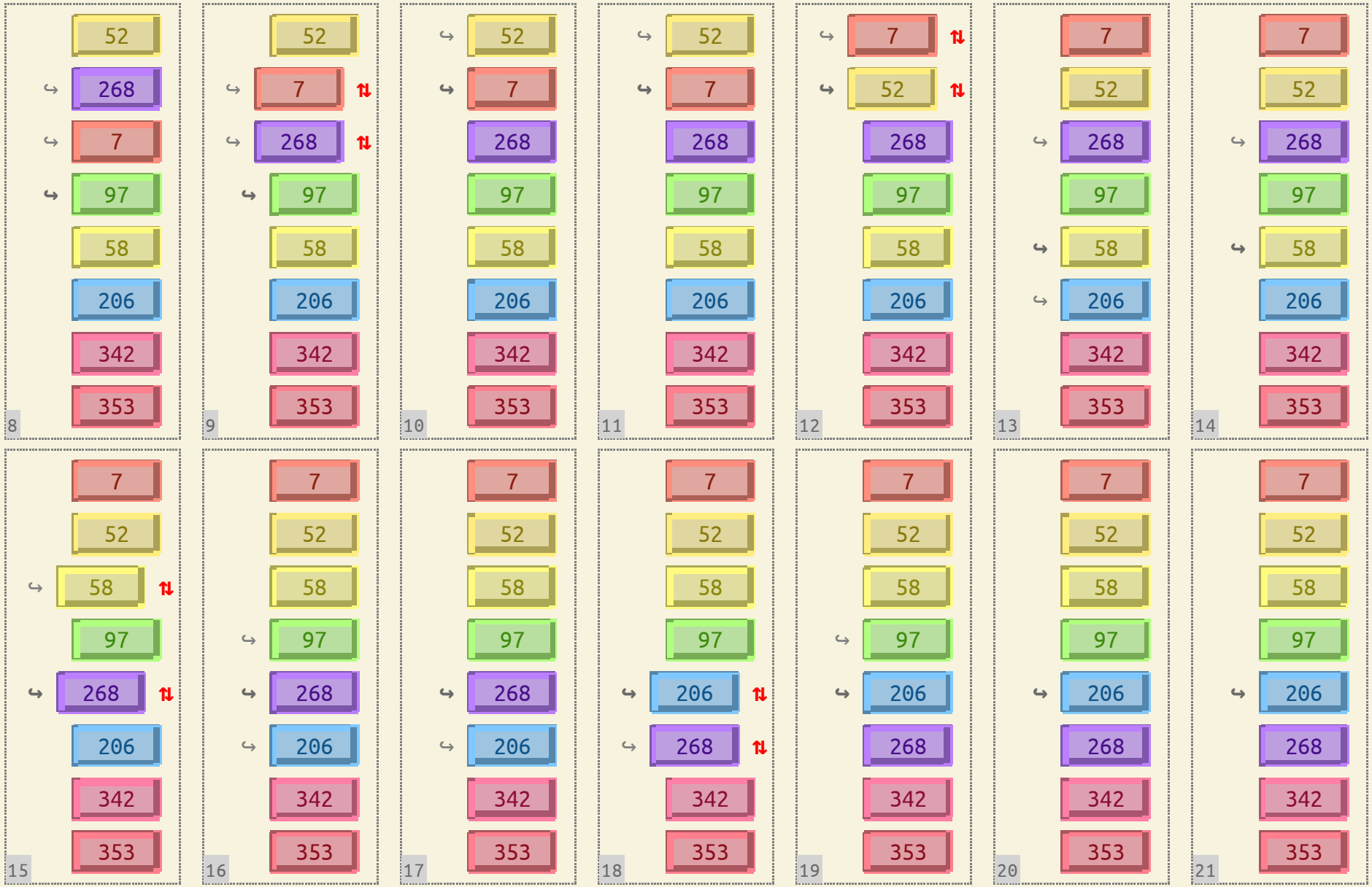
CODE: RECURSE

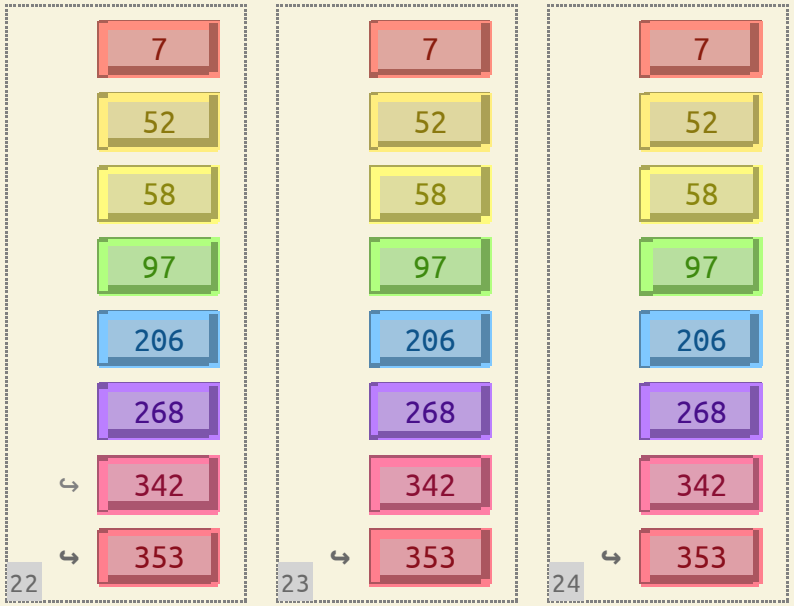
```
function doQuickSort($elements, $leftIndex, $rightIndex) {
    // Divide the array in two, creating a “pivot” value
    // Move any value lower than the pivot to the left array
    // Move any value higher than the pivot to the right array
    // Recursively repeat the same operation on both arrays
    if ($leftIndex < $rightSwapIndex) {
        $this->doQuickSort($elements, $leftIndex, $rightSwapIndex);
    }
    if ($leftSwapIndex < $rightIndex) {
        $this->doQuickSort($elements, $leftSwapIndex, $rightIndex);
    }
}
```

QUICK SORT *(CONT.)*

ITERATIONS




52
268
7
97
58
206
342
353
8
52
7
268
97
58
206
342
353
9
52
7
268
97
58
206
342
353
10
52
7
268
97
58
206
342
353
11
7
52
268
97
58
206
342
353
12
7
52
268
97
58
206
342
353
13
7
52
268
97
58
206
342
353
14
7
52
58
97
268
206
342
353
15
7
52
58
97
268
206
342
353
16
7
52
58
97
268
206
342
353
17
7
52
58
97
206
268
342
353
18
7
52
58
97
206
268
342
353
19
7
52
58
97
206
268
342
353
20
7
52
58
97
206
268
342
353
21


7
52
58
97
206
268
342
353
22
7
52
58
97
206
268
342
353
23
7
52
58
97
206
268
342
353
24

Sorted!

QUICK SORT *(CONT.)*

SUMMARY

- Best / average case: $\mathrm{O}(n \log n)$
- Worst case: $\mathrm{O}(n^2)$
- Implemented by `sort()`
- Easily parallelized

HEAP SORT

CODE

```
class HeapSort {
    public function sort(array $elements) {
        $size = count($elements);
        for ($index = floor(($size / 2)) - 1; $index >= 0; $index--) {
            $elements = $this->siftDown($elements, $index, $size);
        }
        for ($index = $size - 1; $index >= 1; $index--) {
            $tmp = $elements[0];
            $elements[0] = $elements[$index];
            $elements[$index] = $tmp;
            $elements = $this->siftDown($elements, 0, $index - 1);
        }
        return $elements;
    }
}
```

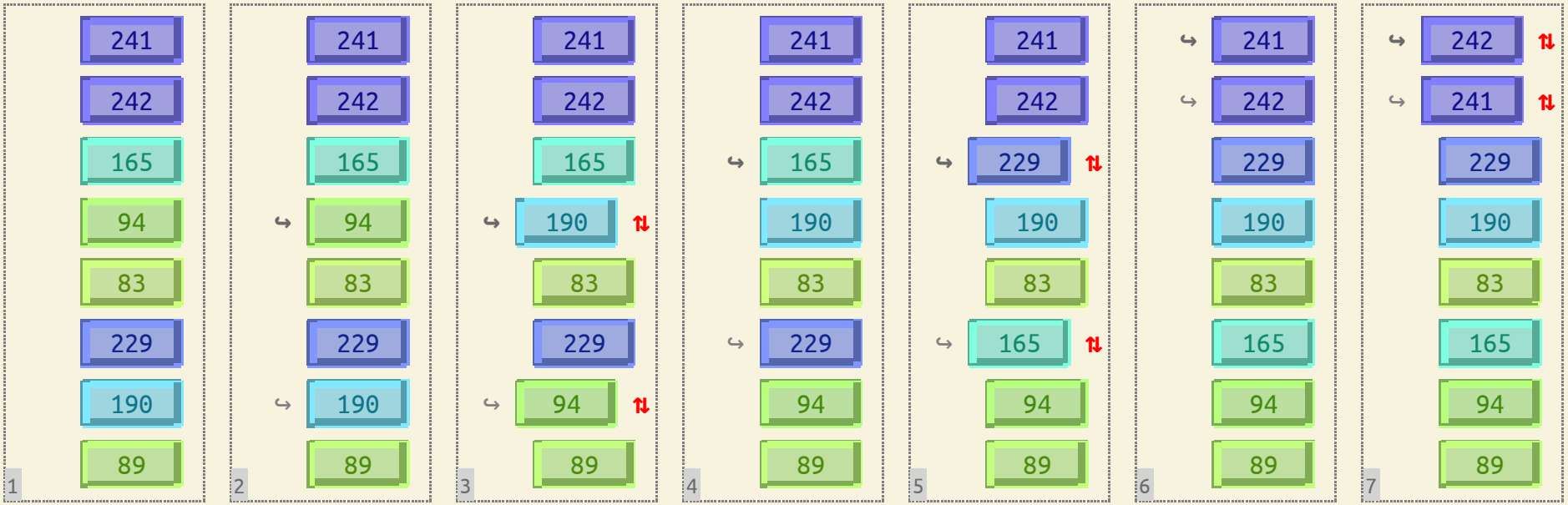
HEAP SORT *(CONT.)*

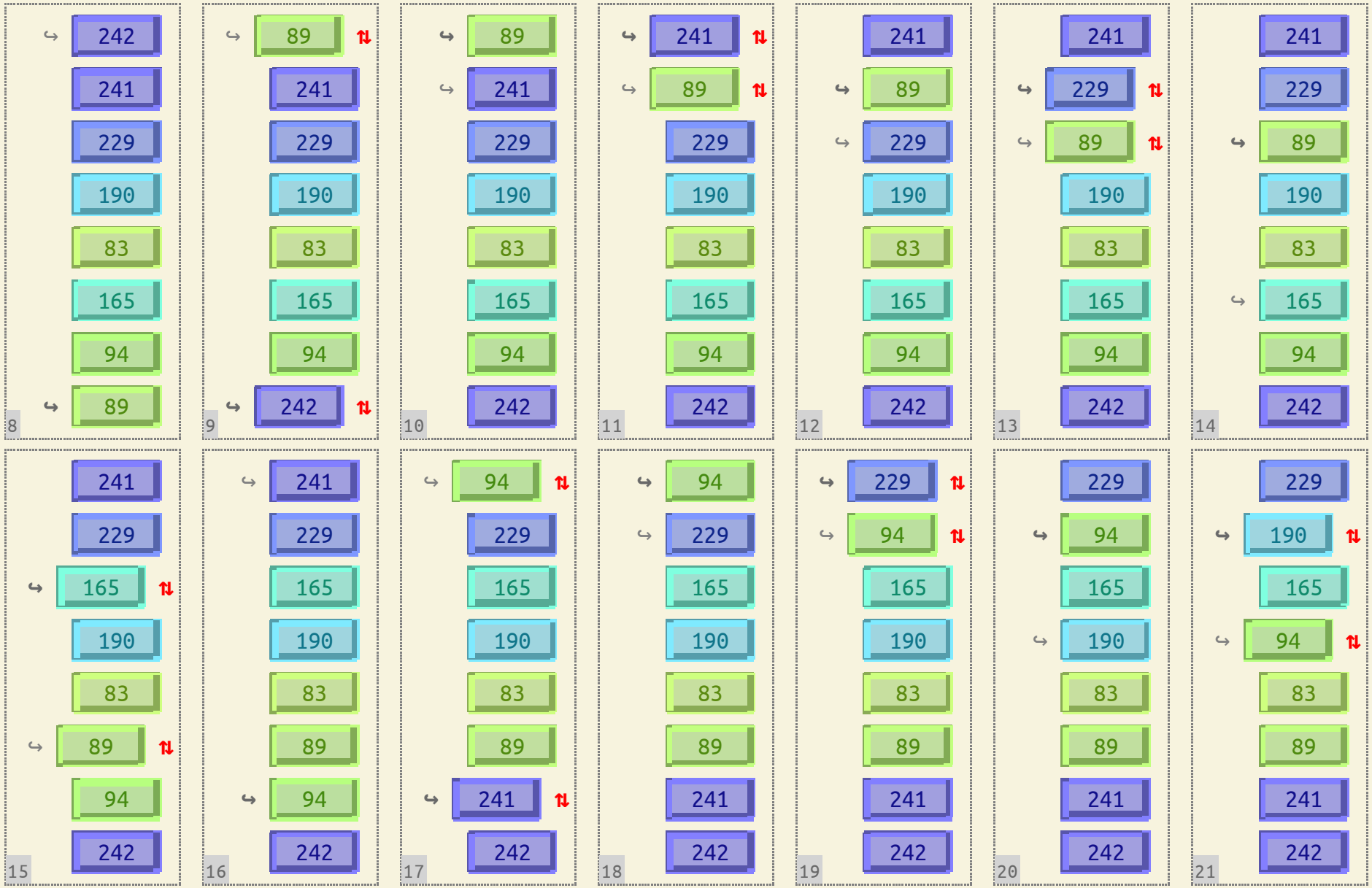
CODE: SIFT THE HEAP

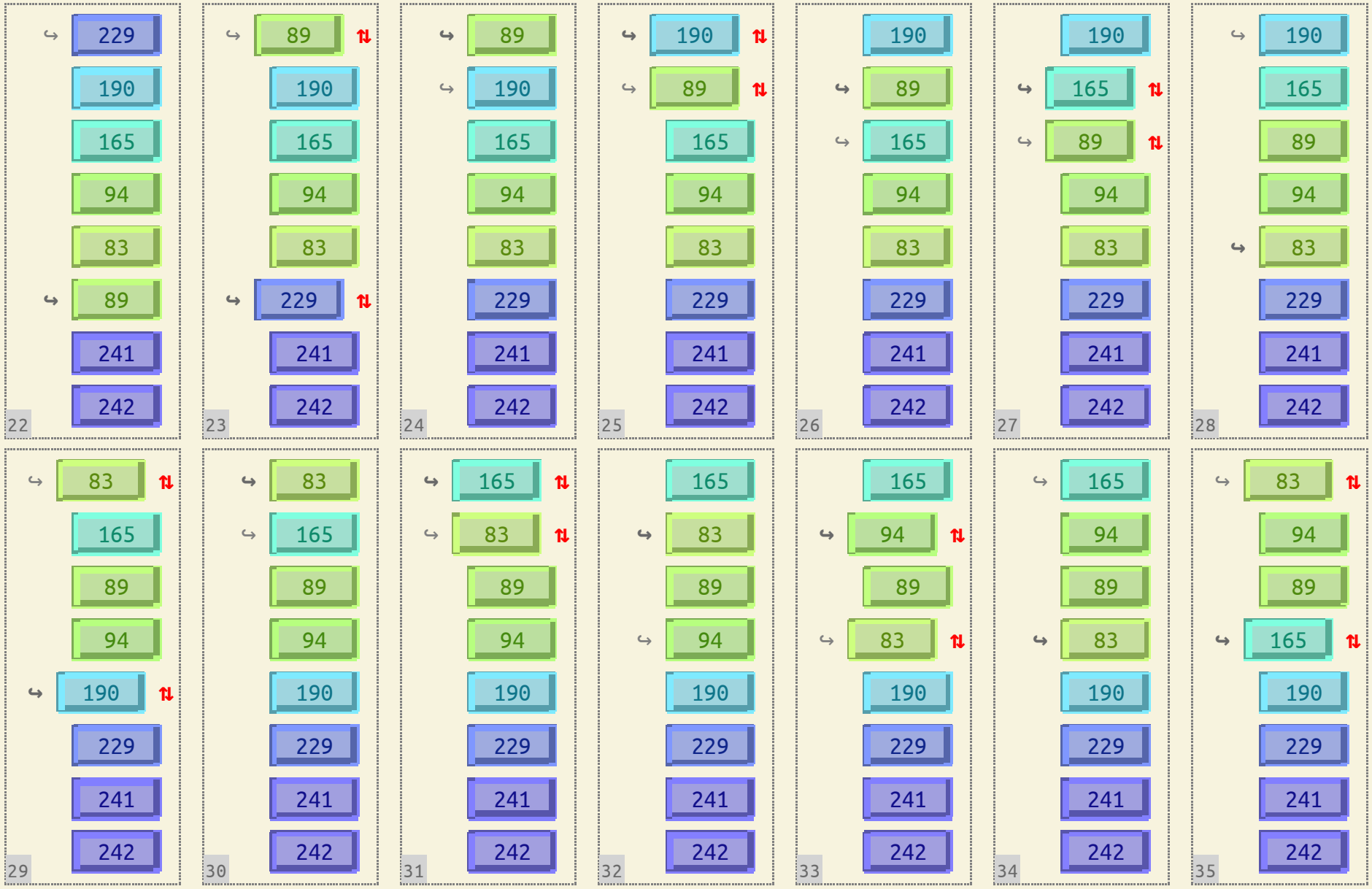
```
public function siftDown(array $elements, $root, $bottom) {
    $done = false;
    while (($root * 2 <= $bottom) && (!$done)) {
        if ($root * 2 == $bottom) $maxChild = $root * 2;
        elseif ($elements[$root * 2] > $elements[$root * 2 + 1]) $maxChild = $root * 2;
        else $maxChild = $root * 2 + 1;

        if ($elements[$root] < $elements[$maxChild]) {
            $tmp = $elements[$root];
            $elements[$root] = $elements[$maxChild];
            $elements[$maxChild] = $tmp;
            $root = $maxChild;
        } else
            $done = true;
    }
    return $elements;
}
```

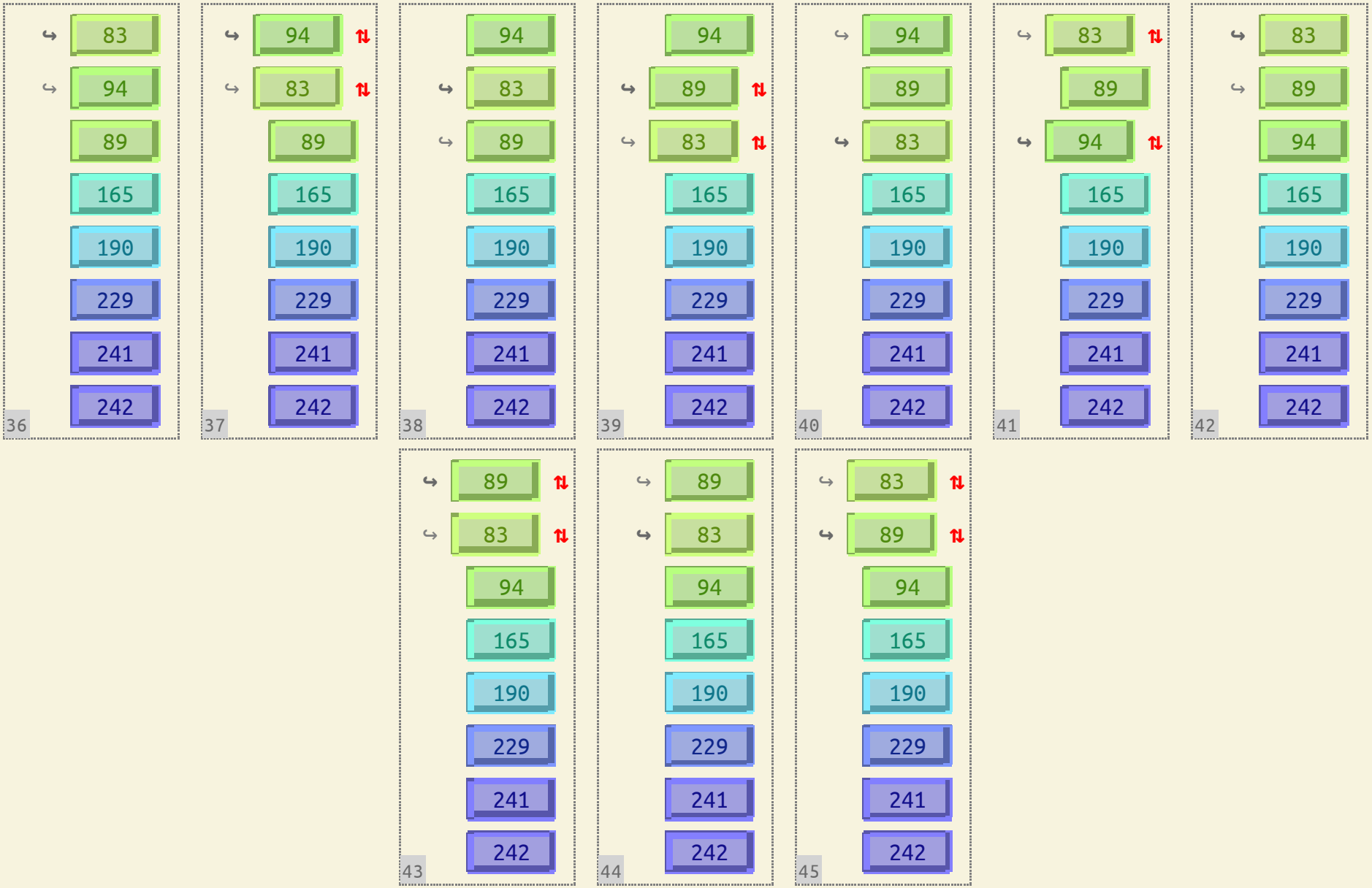
HEAP SORT *(CONT.)*

ITERATIONS




242 241 229 190 83 165 94 89 8
89 241 229 190 83 165 94 242 9
89 241 229 190 83 165 94 242 10
241 89 229 190 83 165 94 242 11
241 89 229 190 83 165 94 242 12
241 229 89 190 83 165 94 242 13
241 229 89 190 83 165 94 242 14
241 229 165 190 83 89 94 242 15
241 229 165 190 83 89 94 242 16
94 229 165 190 83 89 241 242 17
94 229 165 190 83 89 241 242 18
229 94 165 190 83 89 241 242 19
229 94 165 190 83 89 241 242 20
229 190 165 94 83 89 241 242 21


229 190 165 94 83 89 241 242 22
89 190 165 94 83 229 241 242 23
89 190 165 94 83 229 241 242 24
190 89 165 94 83 229 241 242 25
190 89 165 94 83 229 241 242 26
190 165 89 94 83 229 241 242 27
190 165 89 94 83 229 241 242 28
83 165 89 94 190 229 241 242 29
83 165 89 94 190 229 241 242 30
165 83 89 94 190 229 241 242 31
165 83 89 94 190 229 241 242 32
165 94 89 83 190 229 241 242 33
165 94 89 83 190 229 241 242 34
83 94 89 165 190 229 241 242 35


83
94
89
165
190
229
241
242
36
94
83
89
165
190
229
241
242
37
94
83
89
165
190
229
241
242
38
94
89
83
165
190
229
241
242
39
94
89
83
165
190
229
241
242
40
83
89
94
165
190
229
241
242
41
83
89
94
165
190
229
241
242
42
89
83
94
165
190
229
241
242
43
89
83
94
165
190
229
241
242
44
83
89
94
165
190
229
241
242
45

Sorted!

HEAP SORT *(CONT.)*

SUMMARY

- Best / average / worst case: $O(n \log n)$
- Implemented by `SplMinHeap()`

```php
$h = new SplMinHeap();
foreach ($unsorted as $val) $h->insert($val);
$h->top();
while($h->valid()) {
    echo $h->current()."\n";
    $h->next();
}
```

SEARCHING ALGORITHMS

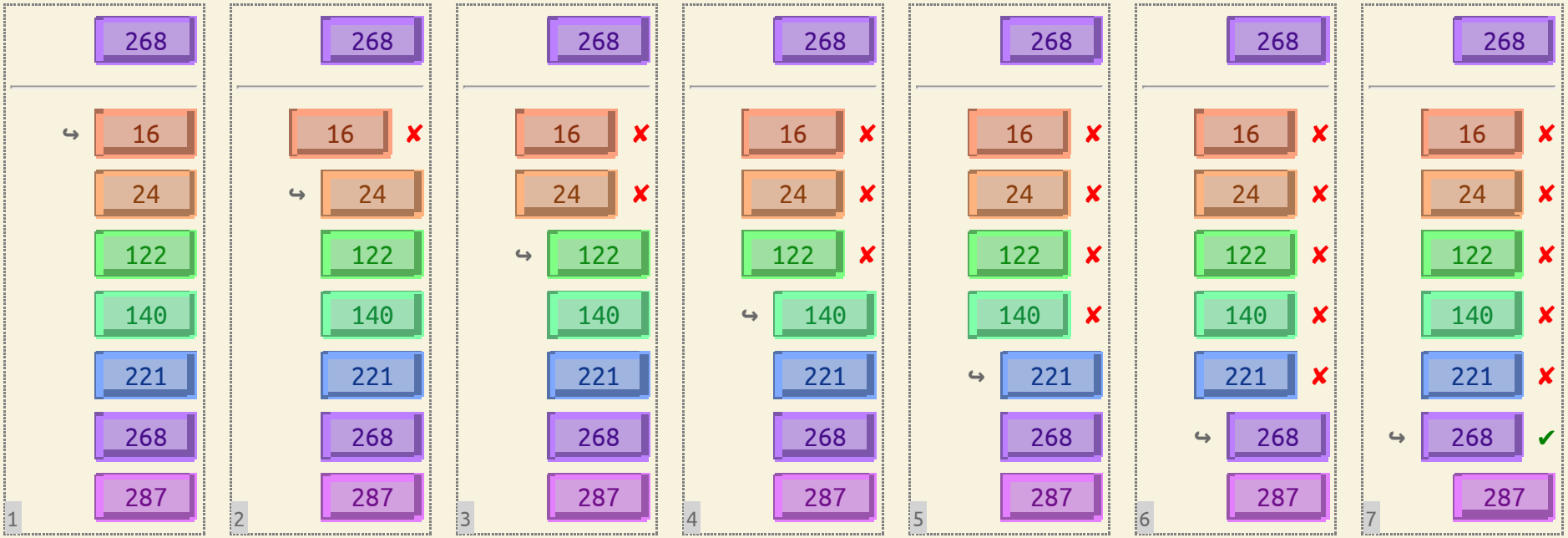
SEQUENTIAL SEARCH

CODE

```
class SequentialSearch {
    public function search($target, array $elements) {
        $iterations = count($elements);

        for($index = 0; $index <= $iterations; $index++) {
            if ($target == $elements[$index]) {
                $this->notifyObservers(array($index));
                return true;
            }
        }

        return false;
    }
}
```

SEQUENTIAL SEARCH *(CONT.)*

ITERATIONS



Found you!

SEQUENTIAL SEARCH *(CONT.)*

SUMMARY

- Best case: $O(1)$
- Average / Worst case: $O(n)$
- Not as pointless as it looks...

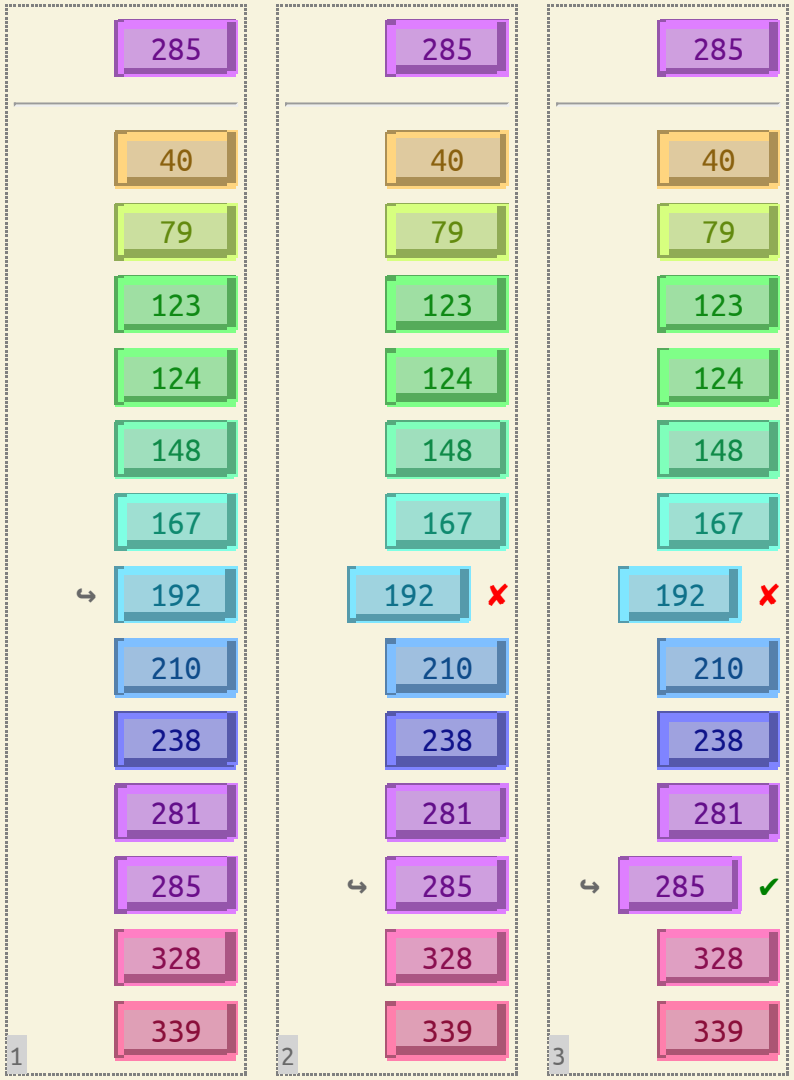
BINARY SEARCH

CODE

```
class BinarySearch {
    public function search($target, array $elements) {
        return $this->doBinarySearch($target, $elements, 0, count($elements));
    }

    public function doBinarySearch($target, array $elements, $minIndex, $maxIndex) {
        if ($maxIndex < $minIndex) { return false; }
        $midIndex = floor(($minIndex + $maxIndex) / 2);
        if ($elements[$midIndex] > $target) {
            return $this->doBinarySearch($target, $elements, $minIndex, $midIndex - 1);
        }
        if ($elements[$midIndex] < $target) {
            return $this->doBinarySearch($target, $elements, $midIndex + 1, $maxIndex);
        }
        return true;
    }
}
```

BINARY SEARCH *(CONT.)*

ITERATIONS


285
40
79
123
124
148
167
192
210
238
281
285
328
339
1
285
40
79
123
124
148
167
192
210
238
281
285
328
339
2
285
40
79
123
124
148
167
192
210
238
281
285
328
339
3

Found you!

BINARY SEARCH *(CONT.)*

SUMMARY

- Best case: $O(1)$
- Average / Worst case: $O(\log n)$
- Switch to linear search for smaller partitions

SUMMING UP

HISTORY

- Insertion Sort: optimised in 1959 (Shell Sort)
- Bubble Sort: improved in 1980 (Comb Sort)
- Quick Sort: developed in 1960 (C. A. R. Hoare)
- Heap Sort: improved in the '60s (Robert Floyd)

Oh, and there's Radix Sort used by Herman Hollerith in 1887

THANK YOU!

JOIND.IN/8454