ALGORITHMS

SORTING & SEARCHING

ROWAN MEREWOOD

LET'S START WITH THE
BIG
PHILOSOPHICAL QUESTIONS

WHO AM I?

Software Engineer and Technical Team Lead

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

```php
$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. `^EOF`

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

WHY SORT?

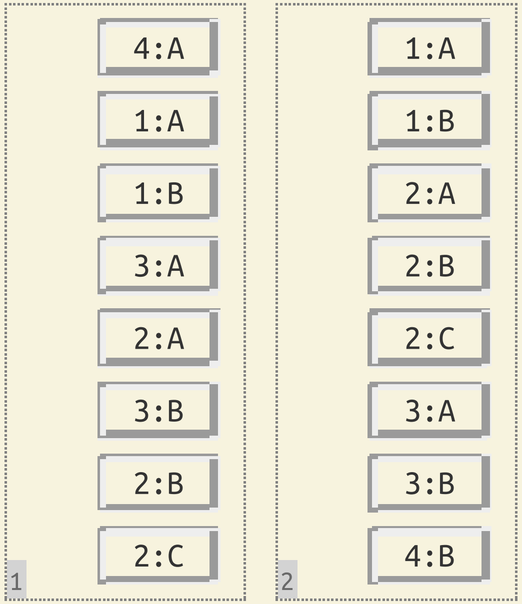
- Displaying lists to humans
- Categorising data
- Preparing data for merging
- Preparing data for searching

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

COMPARING ALGORITHMS

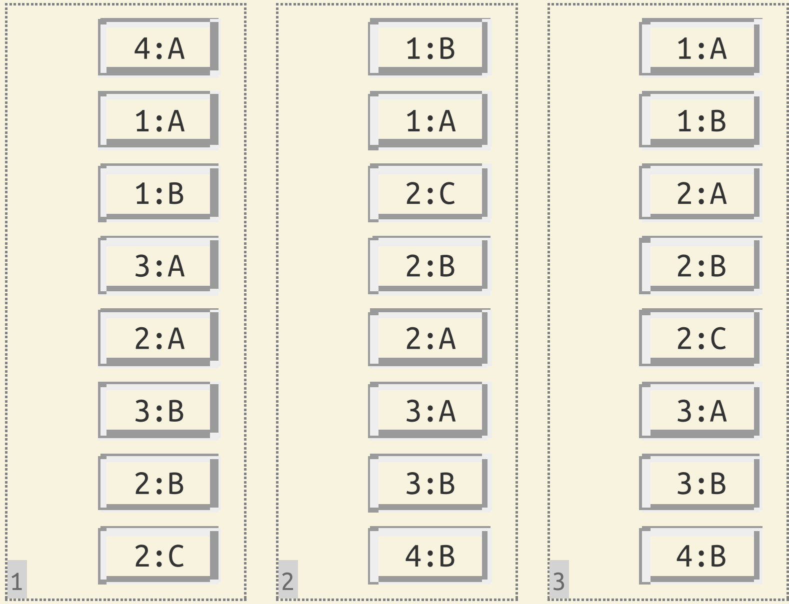
STABILITY

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.)*

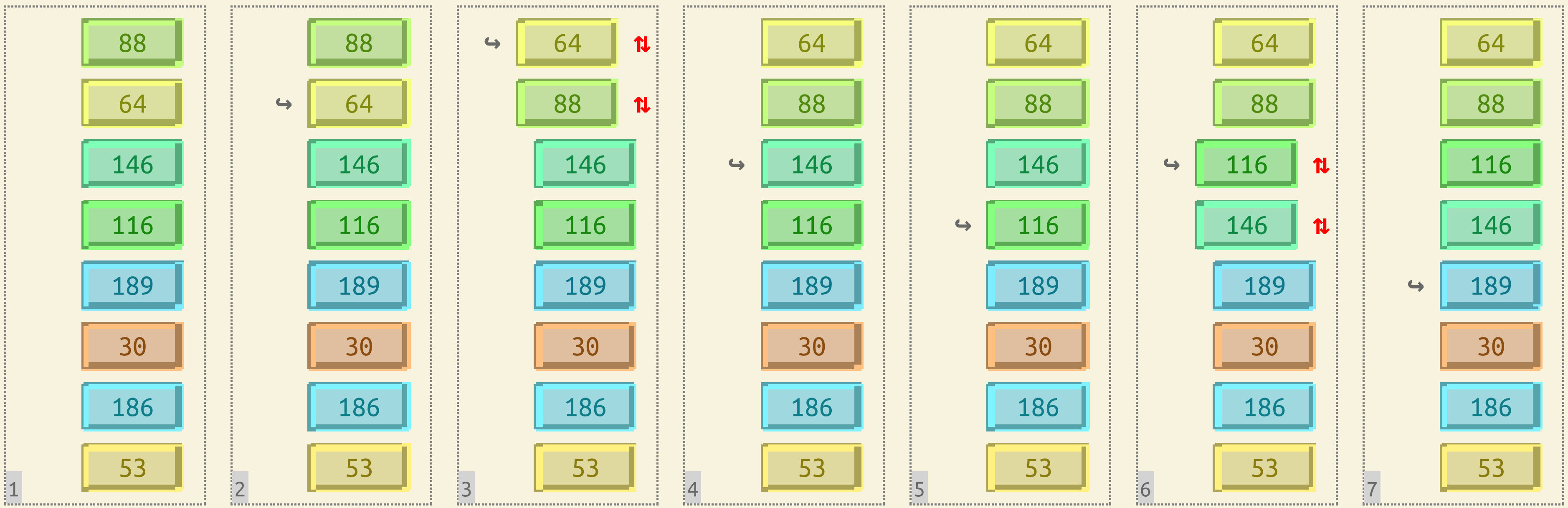
CODE: COMPARE ELEMENTS

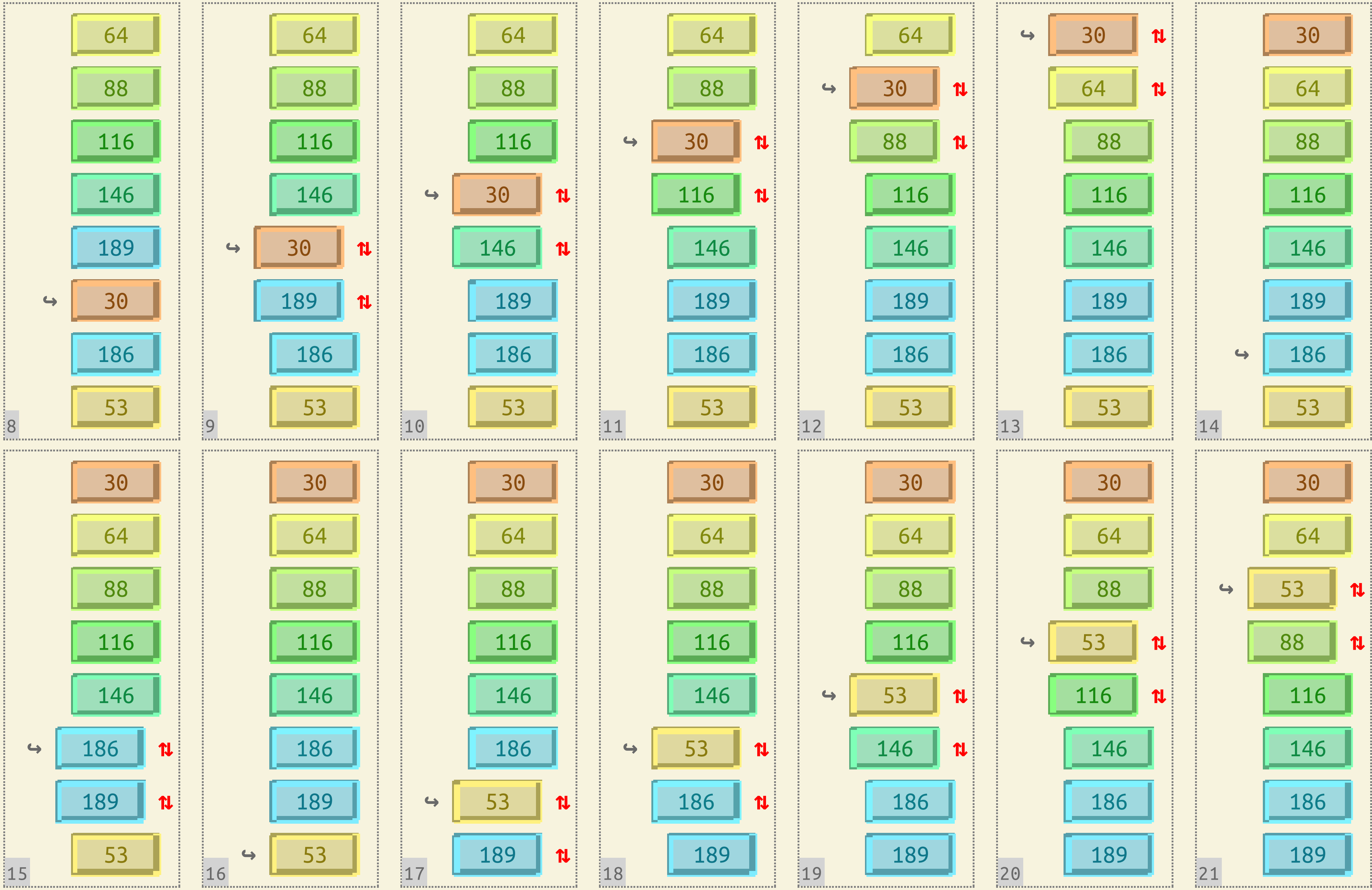
```
for ($index = 1; $index < $iterations; $index++) {
    // "Pick up" the current element and its position
    $elementToInsert = $elements[$index];
    $insertIndex = $index;

    // 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.

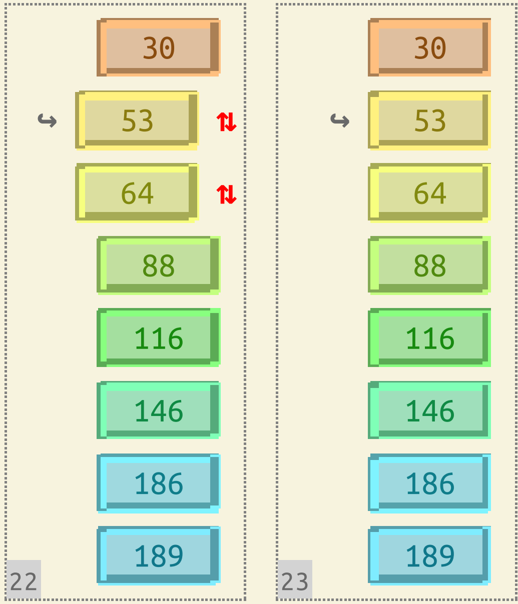
INSERTION SORT *(CONT.)*

ITERATIONS





8 64 88 116 146 189 30 186 53
9 64 88 116 146 30 189 186 53
10 64 88 116 30 146 189 186 53
11 64 88 30 116 146 189 186 53
12 64 30 88 116 146 189 186 53
13 30 64 88 116 146 189 186 53
14 30 64 88 116 146 189 186 53
15 30 64 88 116 146 186 189 53
16 30 64 88 116 146 186 189 53
17 30 64 88 116 146 186 53 189
18 30 64 88 116 146 53 186 189
19 30 64 88 116 53 146 186 189
20 30 64 88 53 116 146 186 189
21 30 64 53 88 116 146 186 189



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.)*

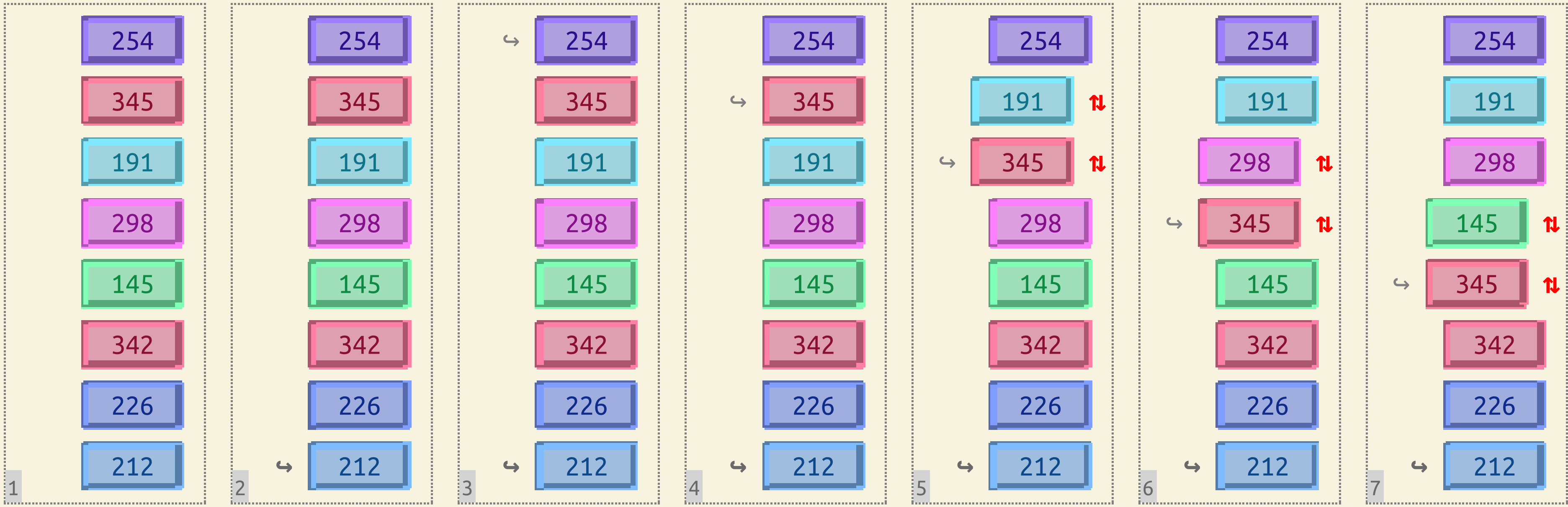
CODE

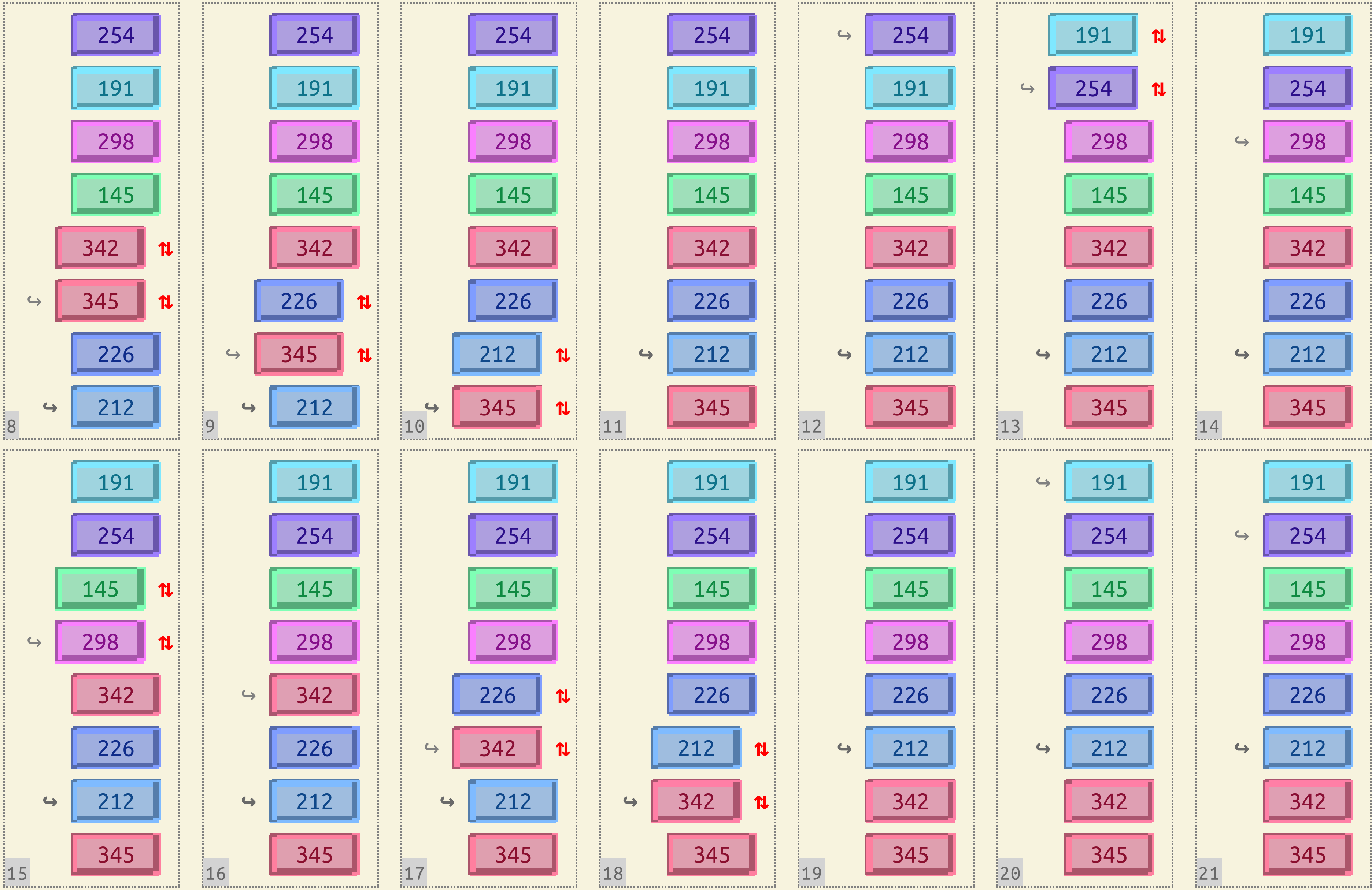
```
for ($index = count($elements); $index > 0; $index--) {
    $swapped = false;
    // Iterate through the unsorted portion of the list
    for ($swapIndex = 0; $swapIndex < $index - 1; $swapIndex++) {
        // Compare and swap elements
        if ($elements[$swapIndex] > $elements[$swapIndex + 1])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

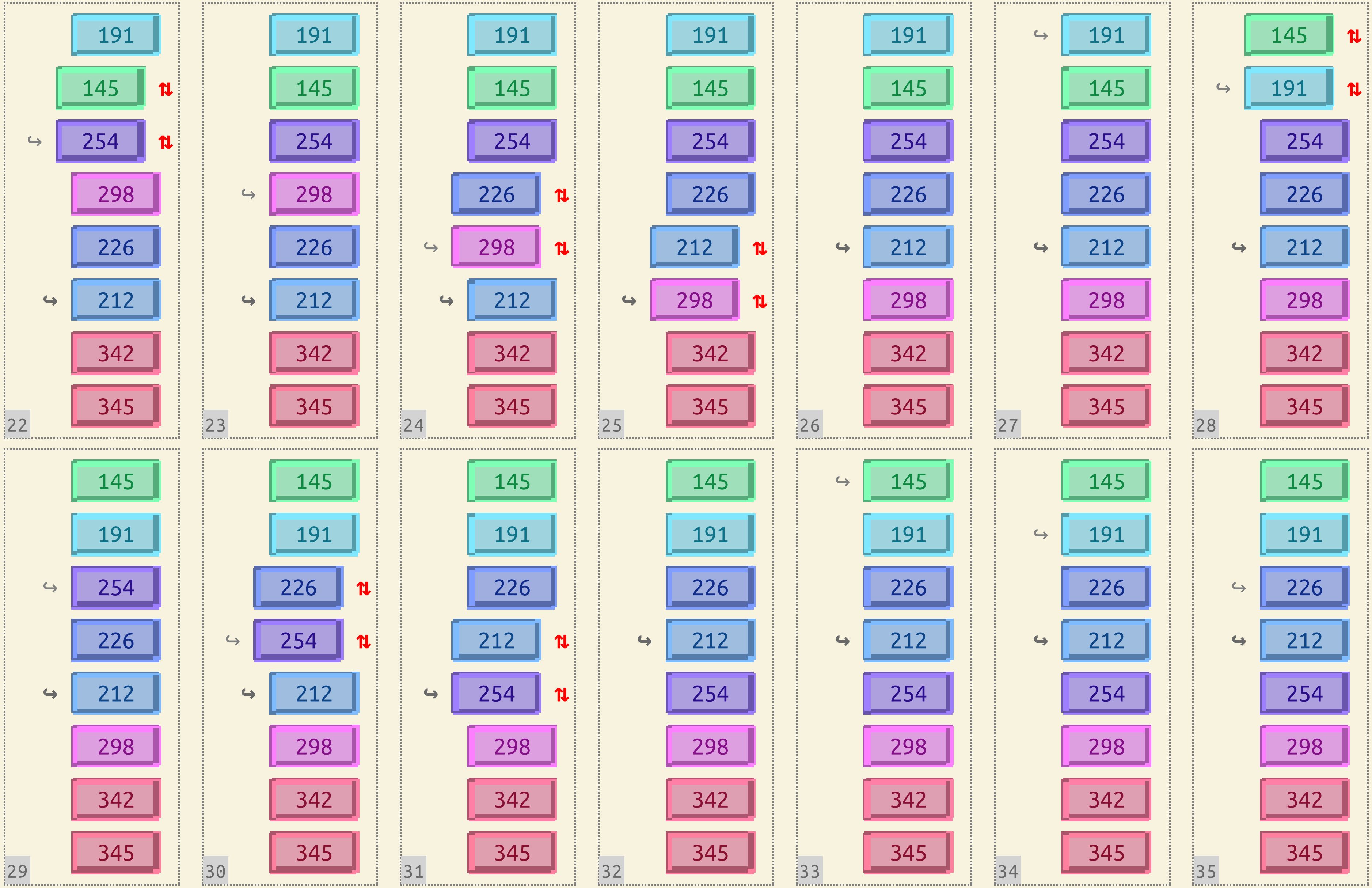
If the list is in order, then `$swapped` stays `false`.

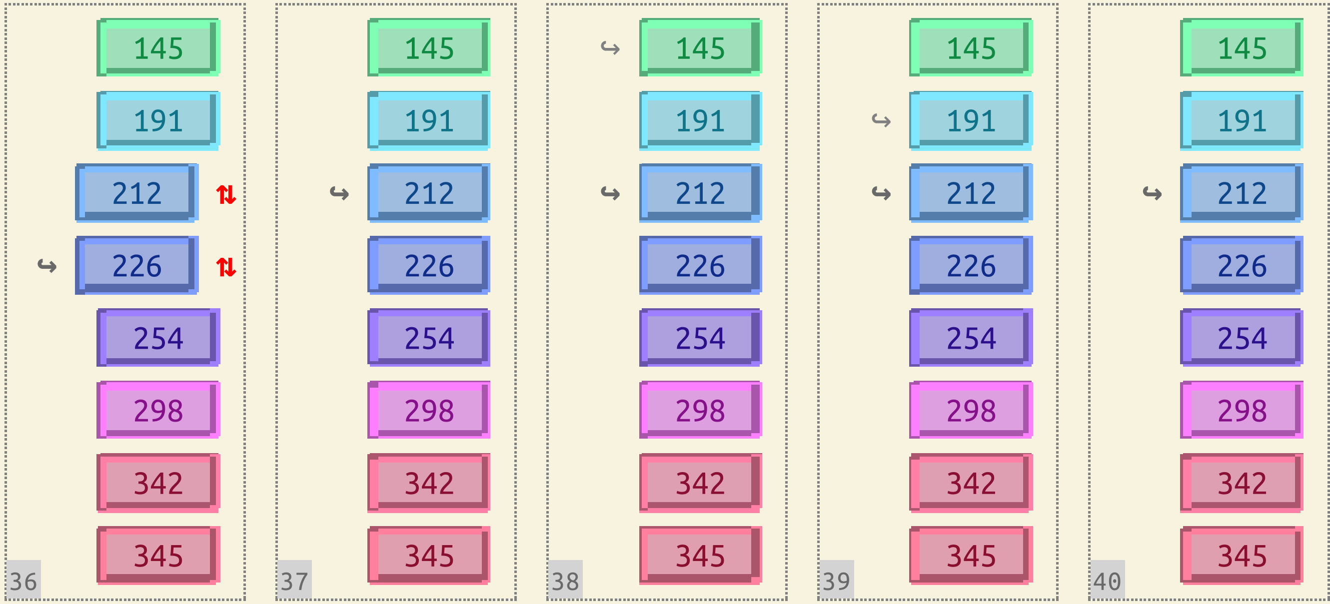
BUBBLE SORT *(CONT.)*

ITERATIONS




254 191 298 145 342 345 226 212 8
254 191 298 145 342 226 345 212 9
254 191 298 145 342 226 212 345 10
254 191 298 145 342 226 212 345 11
254 191 298 145 342 226 212 345 12
191 254 298 145 342 226 212 345 13
191 254 298 145 342 226 212 345 14
191 254 145 298 342 226 212 345 15
191 254 145 298 342 226 212 345 16
191 254 145 298 226 342 212 345 17
191 254 145 298 226 212 342 345 18
191 254 145 298 226 212 342 345 19
191 254 145 298 226 212 342 345 20
191 254 145 298 226 212 342 345 21


191 145 254 298 226 212 342 345
22
191 145 254 298 226 212 342 345
23
191 145 254 226 298 212 342 345
24
191 145 254 226 212 298 342 345
25
191 145 254 226 212 298 342 345
26
191 145 254 226 212 298 342 345
27
145 191 254 226 212 298 342 345
28
145 191 254 226 212 298 342 345
29
145 191 226 254 212 298 342 345
30
145 191 226 212 254 298 342 345
31
145 191 226 212 254 298 342 345
32
145 191 226 212 254 298 342 345
33
145 191 226 212 254 298 342 345
34
145 191 226 212 254 298 342 345
35


145
191
212
226
254
298
342
345
36
145
191
212
226
254
298
342
345
37
145
191
212
226
254
298
342
345
38
145
191
212
226
254
298
342
345
39
145
191
212
226
254
298
342
345
40

Sorted!

BUBBLE SORT *(CONT.)*

SUMMARY

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

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.)*

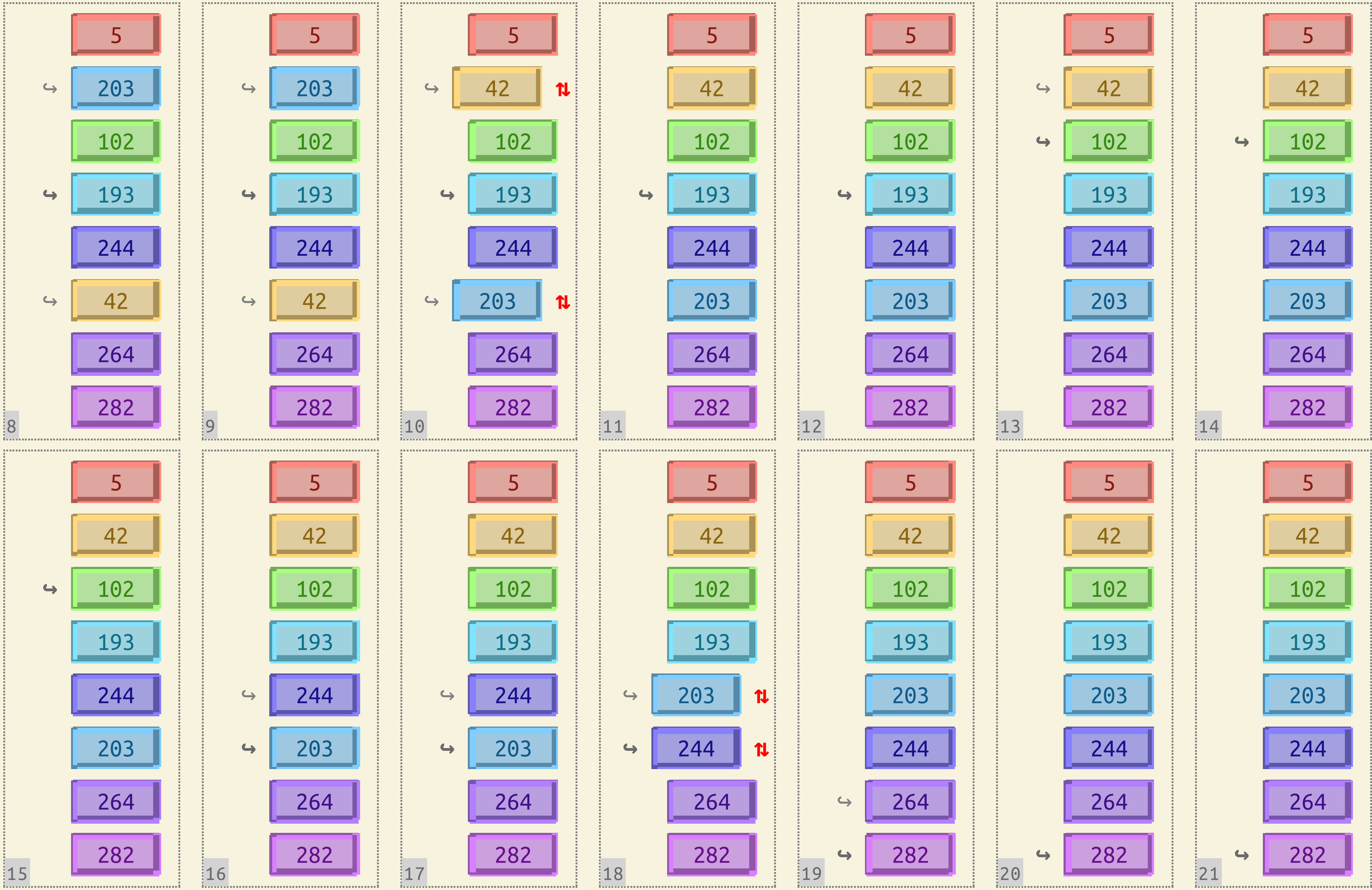
CODE: RECURSE

```
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

1	2	3	4	5	6	7
193	↪ 193	193	193	↪ 193	↪ 193	↪ 5 ⇅
203	203	203	203	203	203	203
102	102	102	102	102	102	102
5	5	5	5	↪ 5	↪ 5	↪ 193 ⇅
264	↪ 264	↪ 264	↪ 244 ⇅	244	244	244
42	42	42	42	↪ 42	42	42
244	244	↪ 244	↪ 264 ⇅	264	264	264
282	↪ 282	282	282	282	282	282


5 203 102 193 244 42 264 282 8
5 203 102 193 244 42 264 282 9
5 42 102 193 244 203 264 282 10
5 42 102 193 244 203 264 282 11
5 42 102 193 244 203 264 282 12
5 42 102 193 244 203 264 282 13
5 42 102 193 244 203 264 282 14
5 42 102 193 244 203 264 282 15
5 42 102 193 244 203 264 282 16
5 42 102 193 244 203 264 282 17
5 42 102 193 203 244 264 282 18
5 42 102 193 203 244 264 282 19
5 42 102 193 203 244 264 282 20
5 42 102 193 203 244 264 282 21

Sorted!

QUICK SORT *(CONT.)*

SUMMARY

- Best / average case: $O(n \log n)$
- Worst case: $O(n^2)$
- 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;
    }
}
```

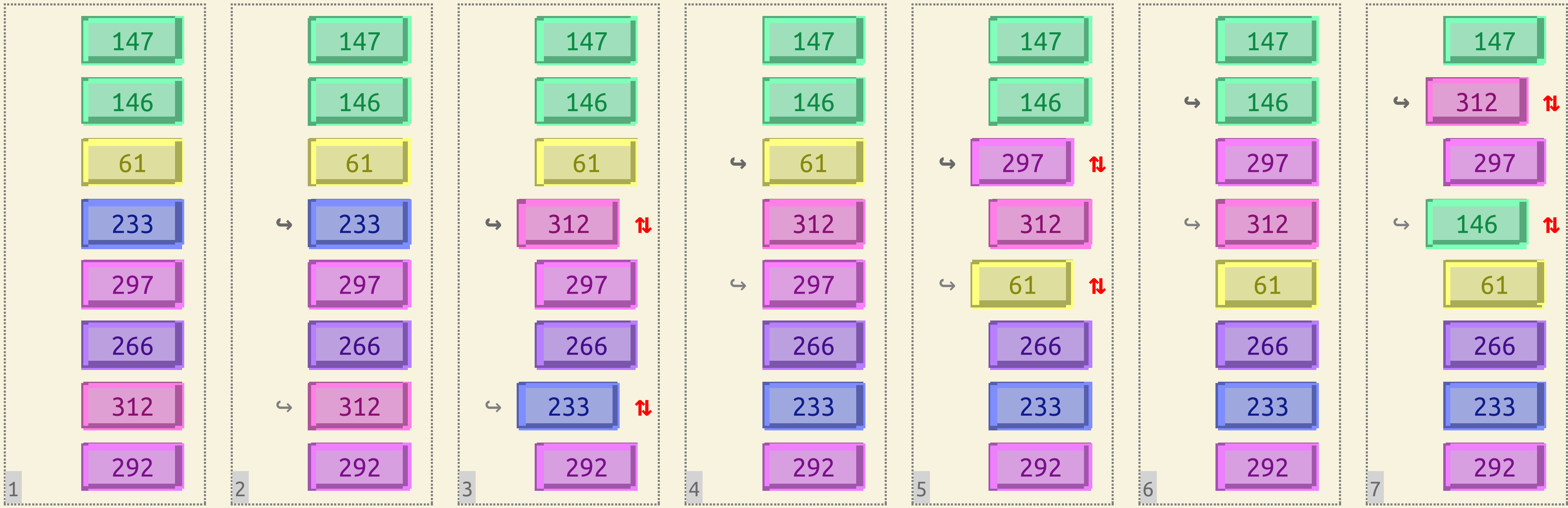
HEAP SORT *(CONT.)*

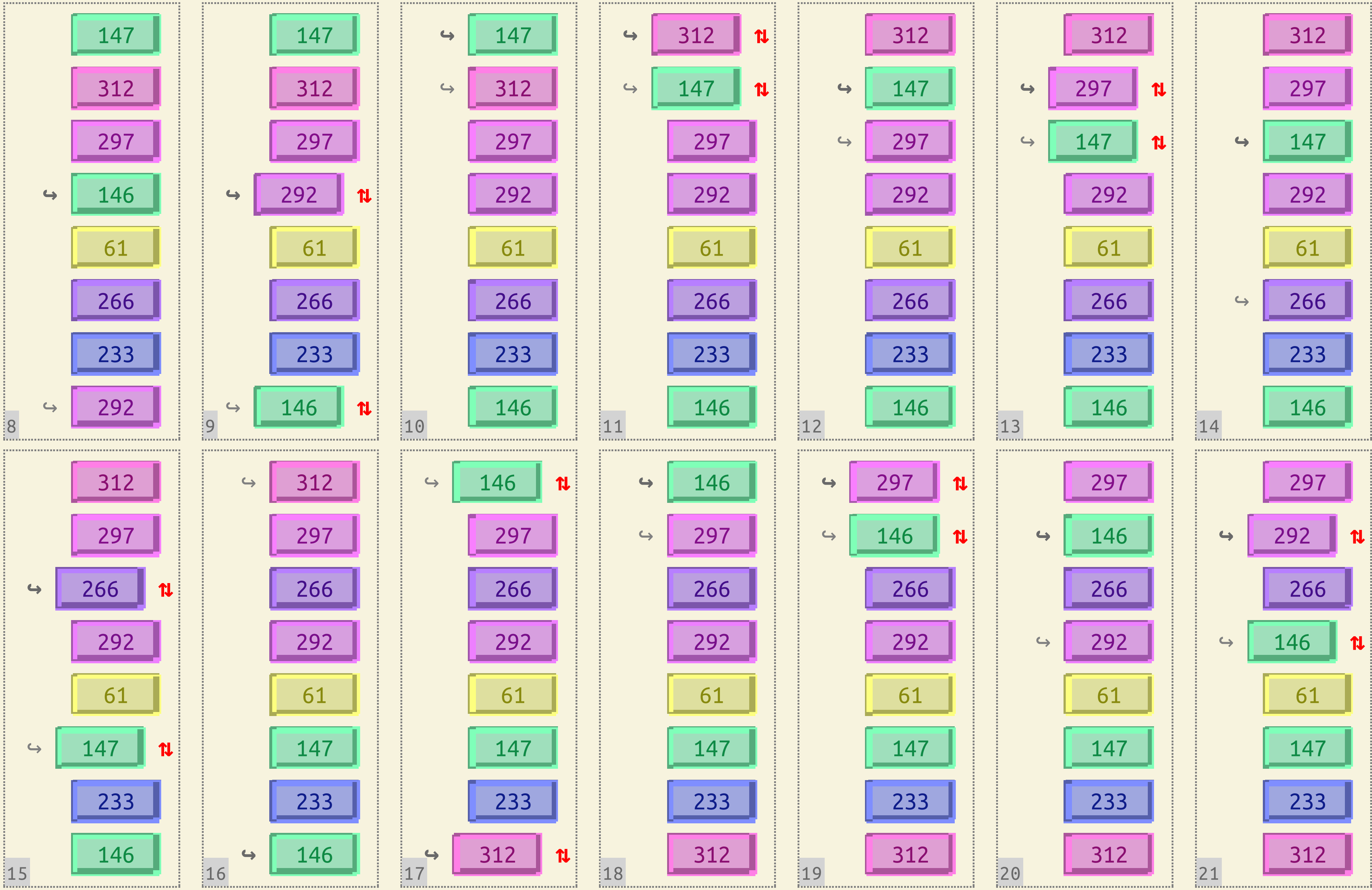
CODE: SIFT THE HEAP

```
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;
}
```

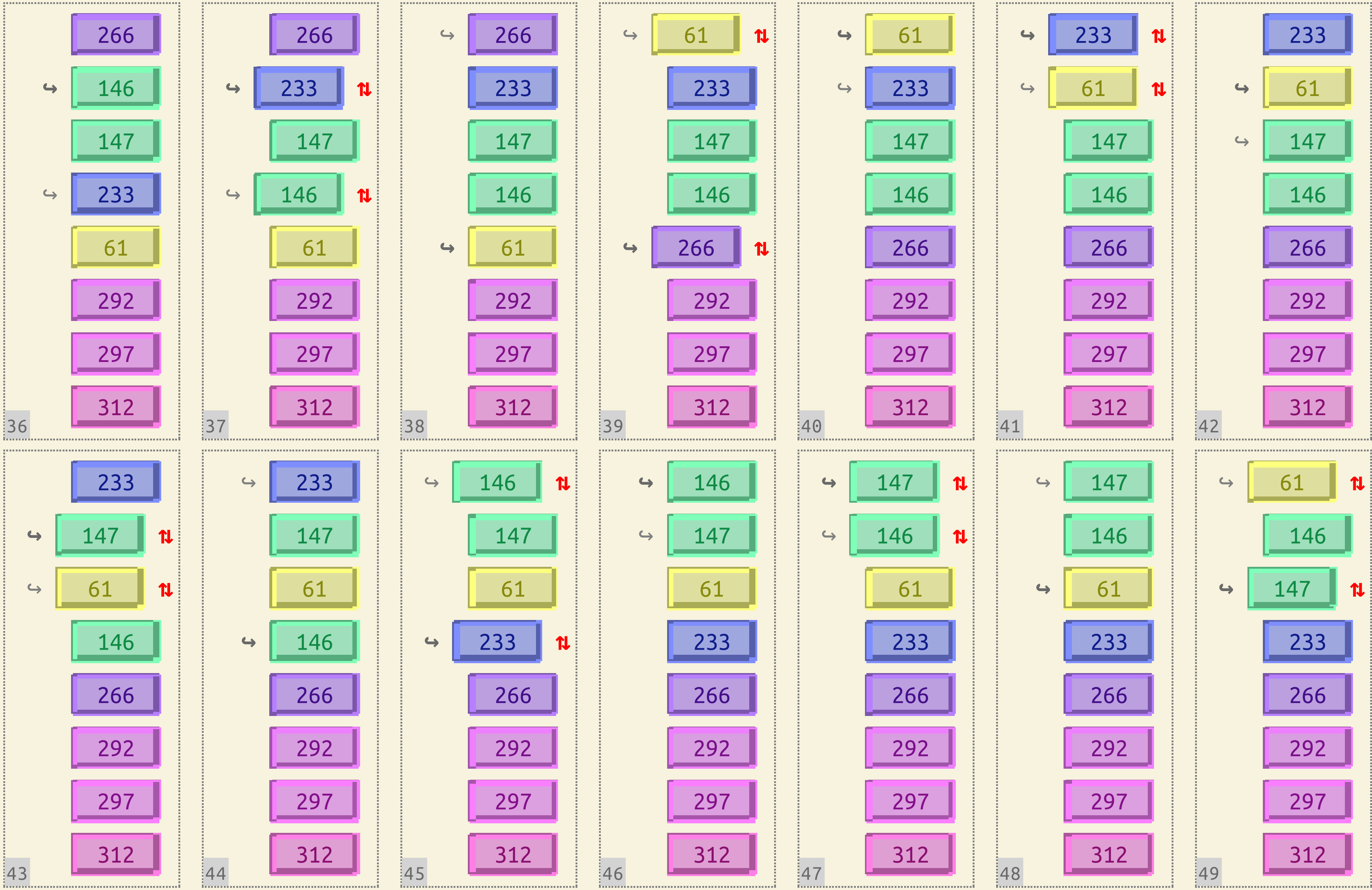
HEAP SORT *(CONT.)*

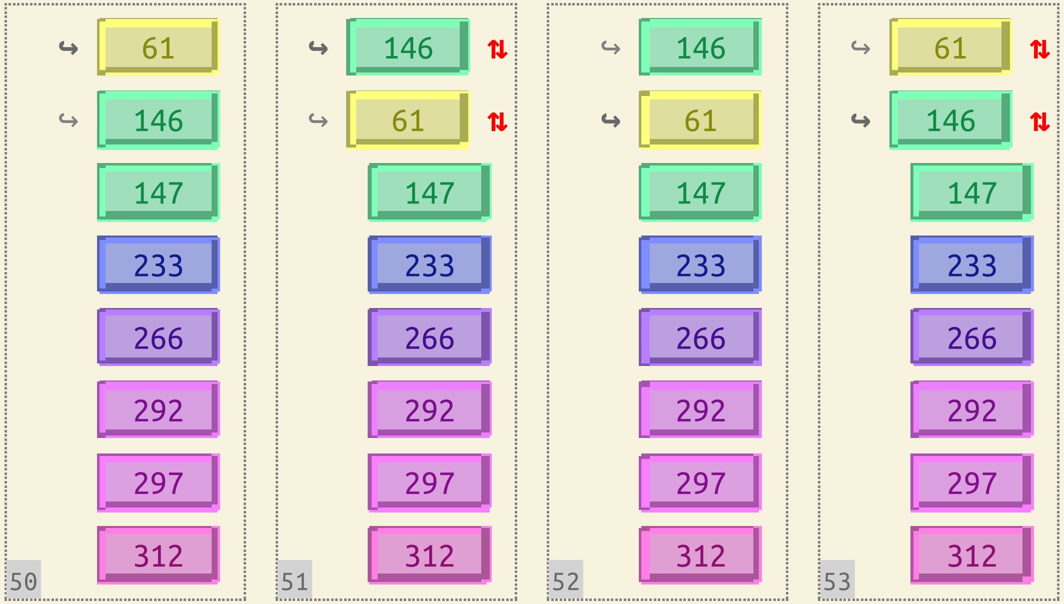
ITERATIONS




147 312 297 146 61 266 233 292
8
147 312 297 292 61 266 233 146
9
147 312 297 292 61 266 233 146
10
312 147 297 292 61 266 233 146
11
312 147 297 292 61 266 233 146
12
312 297 147 292 61 266 233 146
13
312 297 147 292 61 266 233 146
14
312 297 266 292 61 147 233 146
15
312 297 266 292 61 147 233 146
16
146 297 266 292 61 147 233 312
17
146 297 266 292 61 147 233 312
18
297 146 266 292 61 147 233 312
19
297 146 266 292 61 147 233 312
20
297 292 266 146 61 147 233 312
21

22	23	24	25	26	27	28	29	30	31	32	33	34	35
297	297	297	146	146	292	292	292	292	292	292	146	146	266
292	292	292	292	292	146	146	266	266	266	266	266	266	146
266	266	266	266	266	266	266	146	146	147	147	147	147	147
146	233	233	233	233	233	233	233	233	233	233	233	233	233
61	61	61	61	61	61	61	61	61	61	61	61	61	61
147	147	147	147	147	147	147	147	147	146	146	292	292	292
233	146	146	297	297	297	297	297	297	297	297	297	297	297
312	312	312	312	312	312	312	312	312	312	312	312	312	312


266 146 147 233 61 292 297 312
36
266 233 147 146 61 292 297 312
37
266 233 147 146 61 292 297 312
38
61 233 147 146 266 292 297 312
39
61 233 147 146 266 292 297 312
40
233 61 147 146 266 292 297 312
41
233 61 147 146 266 292 297 312
42
233 147 61 146 266 292 297 312
43
233 147 61 146 266 292 297 312
44
146 147 61 233 266 292 297 312
45
146 147 61 233 266 292 297 312
46
147 146 61 233 266 292 297 312
47
147 146 61 233 266 292 297 312
48
61 146 147 233 266 292 297 312
49


61
146
147
233
266
292
297
312
50
146
61
147
233
266
292
297
312
51
146
61
147
233
266
292
297
312
52
61
146
147
233
266
292
297
312
53

Sorted!

HEAP SORT *(CONT.)*

SUMMARY

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

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

SEARCHING ALGORITHMS

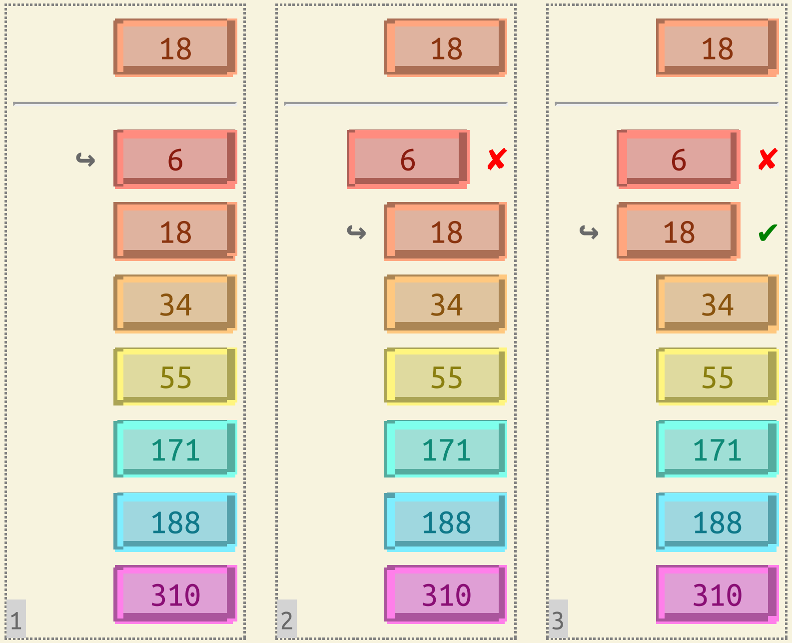
SEQUENTIAL SEARCH

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...

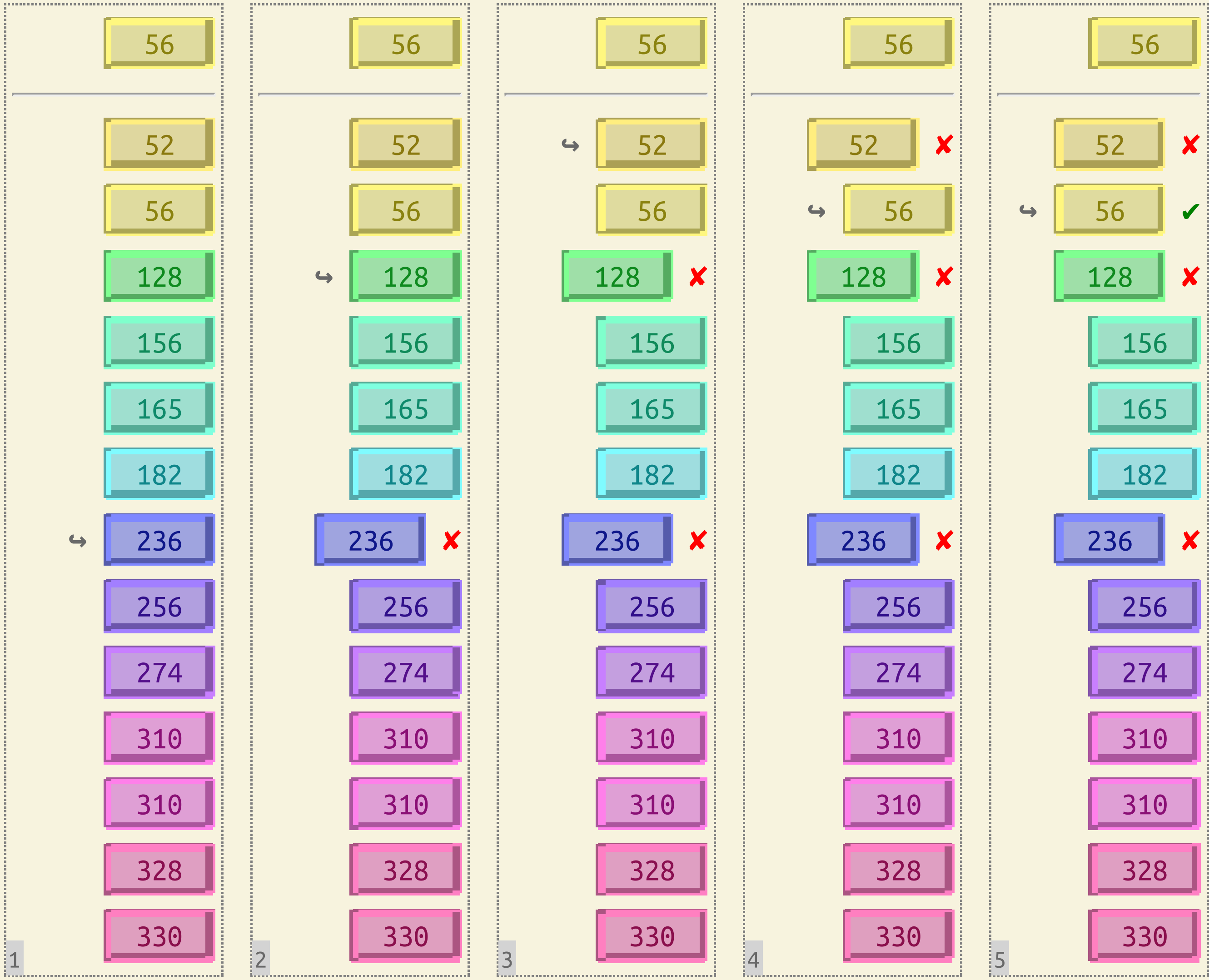
BINARY SEARCH

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


56
52 56 128 156 165 182 236 256 274 310 310 328 330
1
56
52 56 128 156 165 182 236 256 274 310 310 328 330
2
56
52 56 128 156 165 182 236 256 274 310 310 328 330
3
56
52 56 128 156 165 182 236 256 274 310 310 328 330
4
56
52 56 128 156 165 182 236 256 274 310 310 328 330
5

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