

# **ALGORITHM, REVIEW, SORTING**

**MEREWOOD, ROWAN**

# **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',  
  '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?

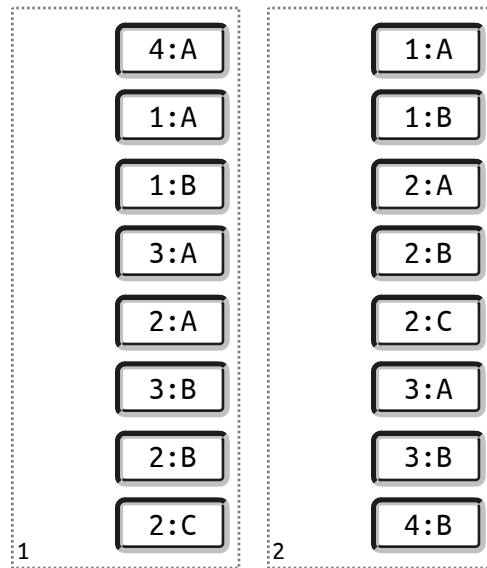
- Displaying lists to humans
- Categorising data
- Preparing data for merging
- Preparing data for searching

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

# COMPARING ALGORITHMS

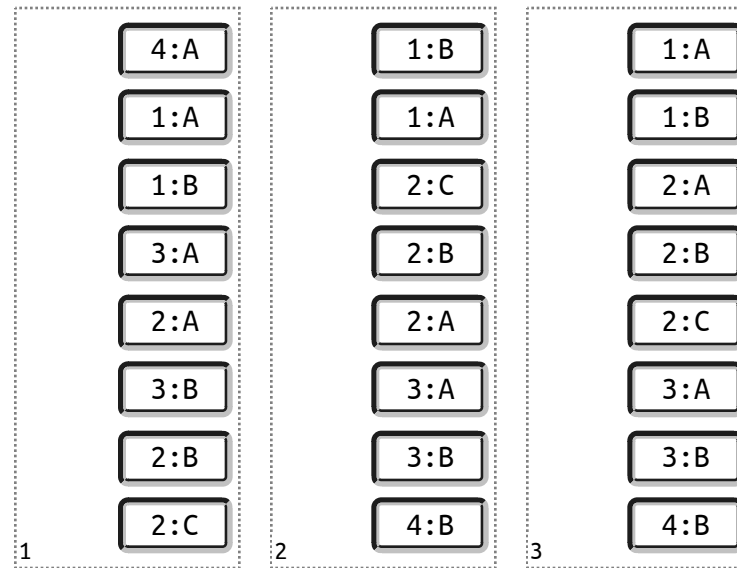
# STABILITY

## 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.)*

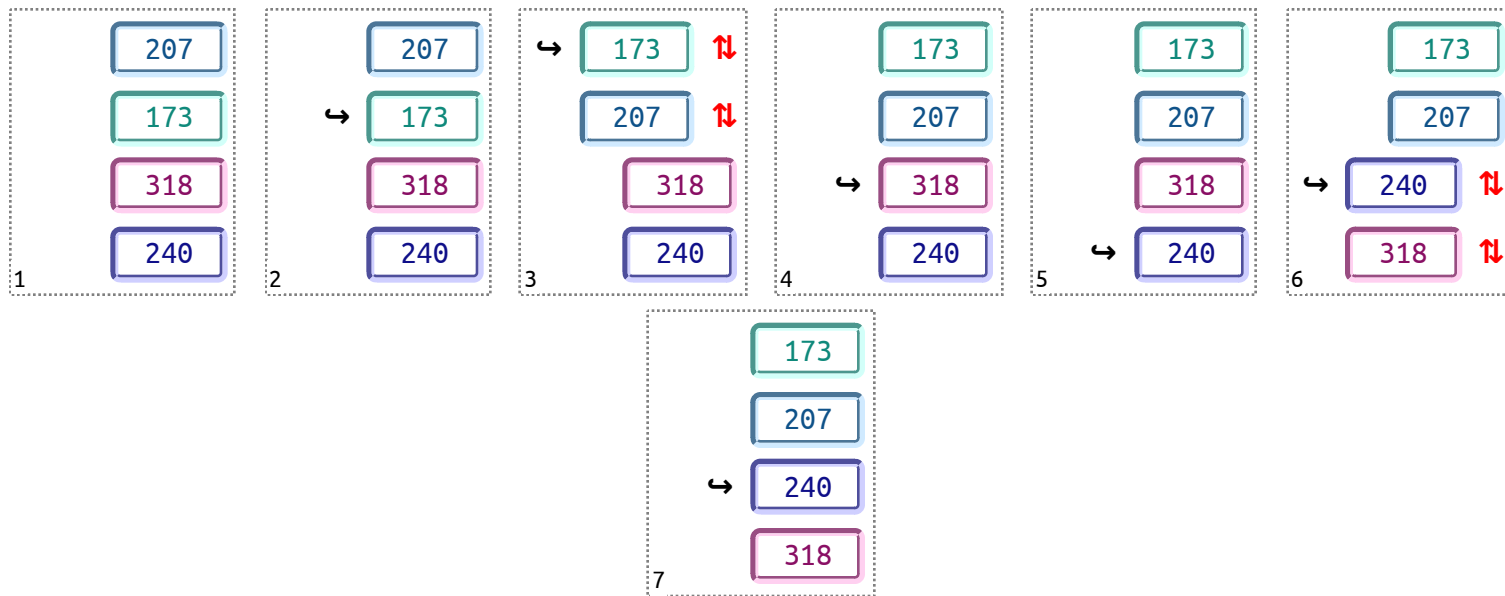
## CODE: COMPARE ELEMENTS

```
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



*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



*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--;  
    }  
}
```

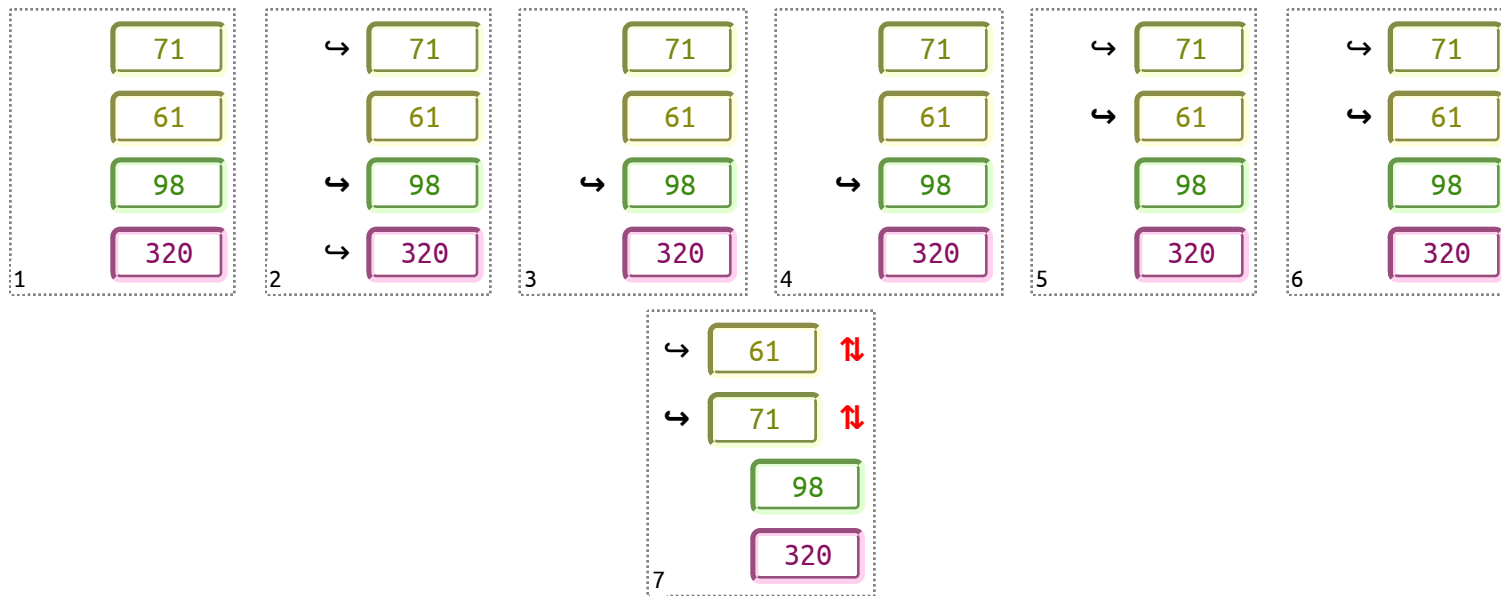
# QUICK SORT *(CONT.)*

## 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



*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;  
    }  
}
```

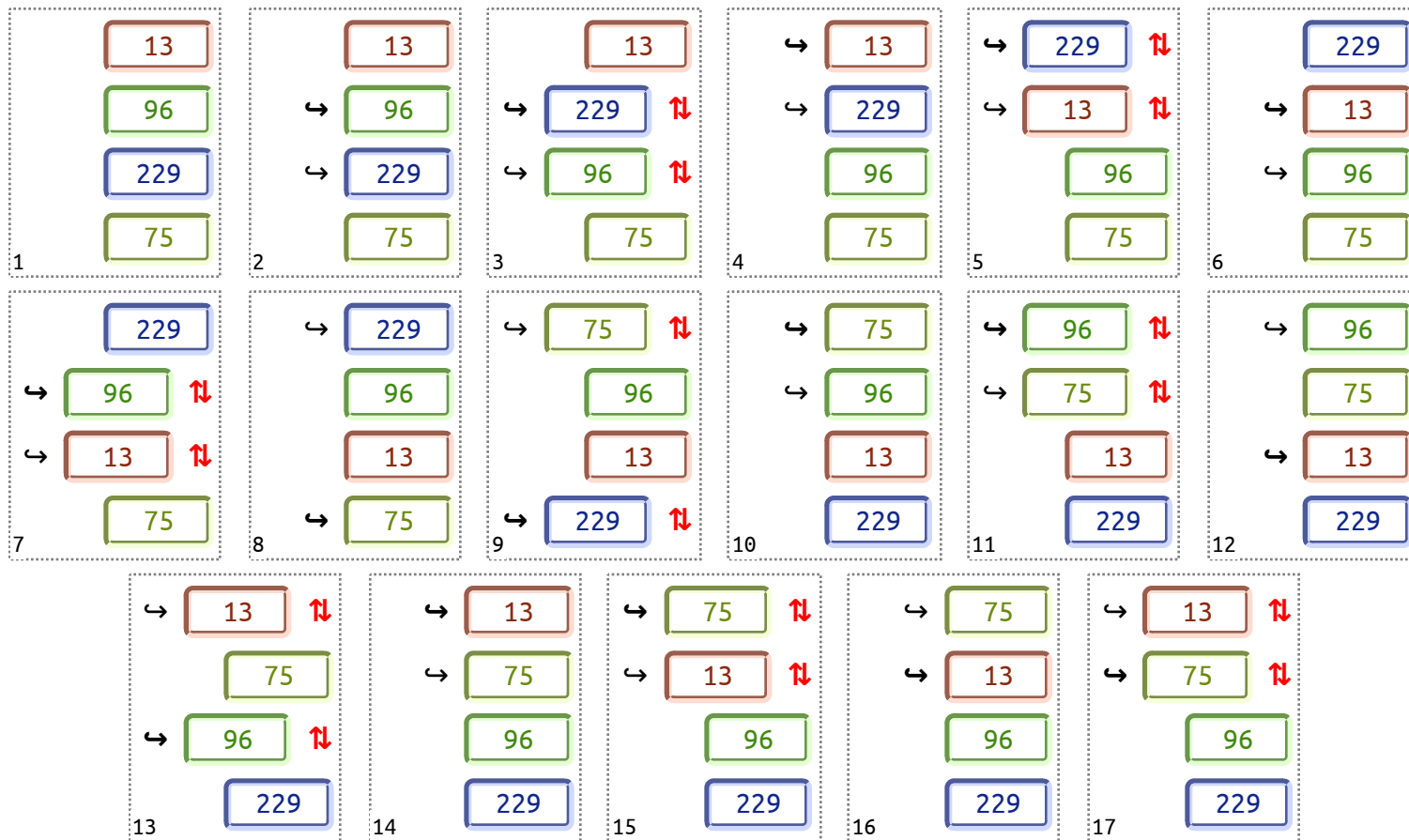
# HEAP SORT *(CONT.)*

## 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





*Sorted!*

# HEAP SORT *(CONT.)*

## SUMMARY

- Best / average / worst case:  $O(n \log n)$
- Implemented by `Sp1MinHeap()`

```
$h = new Sp1MinHeap();  
foreach ($unsorted as $val) $h->insert($val);  
$h->top();  
while($h->valid()) {  
    echo $h->current()."\n";  
    $h->next();  
}
```

# 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*

# COMBINING ALGORITHMS

Set a threshold in the quick sort algorithm so it swaps to insertion sort for smaller arrays

# THANK YOU!

[HTTPS://JOIND.IN/7790](https://joind.in/7790)