

Algorithm,  
Review,  
Sorting

Merewood, Rowan

# Comparing Algorithms

Big **O** notation for:

Running Time – best / average / worst

Memory Usage

**1** →  $n$  →  $n \log n$  →  $n^2$

Also: stability and adaptability

# Insertion Sort

Loop through the whole array

Compare the current element to the previous one  
If it's in order, carry on

If it's not:

Extract it from the array

Loop back through the array  
to find its correct position

# Insertion Sort

```
function insertionSort(array $a) {  
    for ($i = 1; $i < count($a); $i++) {  
        if ($a[$i] > $a[$i-1]) {  
            continue;  
        }  
  
        $tmp = $a[$i];  
        $j = $i - 1;  
  
        while($j >= 0 && $a[$j] > $tmp) {  
            $a[$j+1] = $a[$j];  
            $j--;  
        }  
        $a[$j+1] = $tmp;  
    }  
  
    return $a;  
}
```

# Insertion Sort

Running time

Best:  $O(n)$     Ave. / Worst:  $O(n^2)$

Memory usage:  $O(n)$

Adaptive, Stable, In-place and On-line

# Bubble Sort

Loop through the whole array

Compare the current element to the next

If they're not in order, swap them round

Repeat until no swaps are needed

# Bubble Sort

```
function bubbleSort(array $a) {  
    for ($i = count($a); $i > 0; $i--) {  
        $swapped = false;  
        for ($j = 0; $j < $i-1; $j++) {  
            if ($a[$j] > $a[$j+1]) {  
                $tmp = $a[$j];  
                $a[$j] = $a[$j+1];  
                $a[$j+1] = $tmp;  
                $swapped = true;  
            }  
        }  
        if (!$swapped) {  
            return $a;  
        }  
    }  
}
```

# Bubble Sort

Running time:

Best:  $O(n)$       Ave. / Worst:  $O(n^2)$

Memory usage:  $O(n)$

*"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"*

- Donald Knuth

More writes, more swaps, etc...

# Quick Sort

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

```
function quickSort(array &$a, $left, $right) {  
    $pivot = $a[ceil(($left + $right) / 2)];  
    $i = $left;  
    $j = $right;  
    while ($i <= $j) {  
        while ($a[$i] < $pivot) $i++;  
        while ($a[$j] > $pivot) $j--;  
        if ($i <= $j) {  
            $tmp = $a[$i];  
            $a[$i++] = $a[$j];  
            $a[$j--] = $tmp;  
        }  
    }  
    if ($left < $j) quickSort($a, $left, $j);  
    if ($i < $right) quickSort($a, $i, $right);  
    return $a;  
}
```

# Quick Sort

Running time:

Best/Ave:  $O(n \log n)$       Worst:  $O(n^2)$

Memory usage:  $O(n)$  [well sort of...]

Easily parallelized

Used by PHP's `sort()`

Optimisations on: picking a pivot point,  
storage options and recursion

# Heap Sort

A heap is a specific type of binary tree

In a heap, the parent node is always greater than the child

If you convert your array to a heap, the root of the heap will be the greatest value

Extract that value, then convert the rest of the array to a heap again

# Heap Sort

```
function heapSort(array $a) {  
    $size = count($a);  
    for ($i = round(($size / 2))-1; $i >= 0; $i--)  
        siftDown($a, $i, $size);  
  
    for ($i = $size-1; $i >= 1; $i--) {  
        $tmp = $a[0];  
        $a[0] = $a[$i];  
        $a[$i] = $tmp;  
        siftDown($a, 0, $i-1);  
    }  
    return $a;  
}
```

# Heap Sort

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

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

# Heap Sort

Running time:  $O(n \log n)$  ← every time

```
$h = new SplMinHeap();
```

```
foreach ($unsorted as $val) $h->insert($val);
```

```
$h->top();
```

```
while($h->valid()) {  
    echo $h->current()."\n";  
    $h->next();  
}
```

# Counting Sort

Find the highest and lowest values in the array  
and calculate the range

Create another 0-filled array  
the size of the range

Populate that with a count of the instances of  
each value in the array

Now loop through the values in the range again  
Populating a final array with the  
Relevant number of instances of the value

# Counting Sort

```
function countingSort(array $a) {  
    $size = count($a);  
    $min = $max = $a[0];  
    for($i = 1; $i < $size; $i++) {  
        if ($a[$i] < $min)  
            $min = $a[$i];  
        elseif ($a[$i] > $max)  
            $max = $a[$i];  
    }  
    $range = $max - $min + 1;  
    $count = array();  
    for($i = 0; $i < $range; $i++)  
        $count[$i] = 0;  
    for($i = 0; $i < $size; $i++)  
        $count[ $a[$i] - $min ]++;  
    $z = 0;  
    for($i = $min; $i <= $max; $i++)  
        for($j = 0; $j < $count[ $i - $min ]; $j++)  
            $a[$z++] = $i;  
    return $a;  
}
```

# Counting Sort

Running time:  $O(n+c)$

However, if  $c$  is large this is silly

Especially on memory usage!

Sorting a list of exam grades

# Algorithm Ages

Insertion Sort ← optimised in 1959 (Shell Sort)

Counting Sort ← invented in 1954 (Harold H. Seward)

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



Sorted.