### Sorting Algorithms... so far

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complexity</th>
<th>Copy Complexity</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble Sort</td>
<td>$O(N^2)$</td>
<td>$O(N^2)$ comp.+ $O(N^2)$ swaps</td>
<td>Slow, slow, slow</td>
</tr>
<tr>
<td>Selection Sort</td>
<td>$O(N^2)$</td>
<td>$O(N^2)$ comp.+ $O(N)$ swaps</td>
<td>Intuitive but still slow</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>$O(N^2)$</td>
<td>$O(N^2)$ comp.+ $O(N^2)$ copies</td>
<td>half #comp. than Bubble, O(N) total if data almost sorted</td>
</tr>
<tr>
<td>Enhanced Insertion Sort</td>
<td>$O(N^2)$</td>
<td>$O(N \log N)$ comp.+ $O(N^2)$ copies</td>
<td>Use binary search rather than linear search</td>
</tr>
<tr>
<td>List Insertion Sort</td>
<td>$O(N^2)$</td>
<td>$O(N^2)$ comp.+ $O(N)$ copies</td>
<td>Only 2N copies, Does not work 'in-place'</td>
</tr>
<tr>
<td>MergeSort</td>
<td>$O(N \log N)$</td>
<td>$O(N)$ copies by $O(\log N)$ levels</td>
<td>Divide &amp; Conquer + Recursive, Does not work 'in place'</td>
</tr>
<tr>
<td>ShellSort</td>
<td>$O(N(\log N)^2)$</td>
<td>In average</td>
<td>'Insertion sort' using increment, Worst case not far from average</td>
</tr>
<tr>
<td>QuickSort</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Partitioning

- Action of dividing into groups depending on key value

Partitioning is the underlying mechanism of quicksort

- Note: Partitionning is not stable. There may also be more items in one group than in the other
- It is a O(N) algorithm.
public int partition(int left, int right, double pivot) {
// here the array of double (for example) is a private variable of the class
int leftProbe = left-1; //becomes left after ++leftProbe
int rightProbe = right+1; //becomes right after --rightProbe

while (true) {
    // search smaller item from left- with extra test if it reaches the end
    while (leftProbe<right && array[++leftProbe]<pivot) { ; }
    // search larger item from right- with extra test if it reaches the end
    while (rightProbe>left && array[--rightProbe]>pivot) { ; }

    if (leftProbe >= rightProbe) // cross-over → end of partitioning process
        break;
    else // found misplaced items; swap
        swap(leftProbe, rightProbe);
}
return leftProbe;
}
QuickSort: Intro

- Most popular sorting algorithm
- In majority of situations, it is the fastest: $O(N\log N)$ and it works in-place (no extra copy needed)
- The Quicksort works by partitionining the array into two sub-arrays and then calling itself recursively to sort the left sub-array and then the right sub-array
- It is a divide a conquer algorithm (like MergeSort)
- Basic recursive code is short and simple
- However, sorting very large numbers of data items using recursive procedure may cause the memory stack to overflow. It is often recommended to remove the recursion and use iterations
public void recQuickSort(int left, int right) {
    // size 1- base step- sorted
    if (right - left <= 0) return;
    // size 2 or larger- recursive step
    else {
        // partition range using pivot at right and return pivotIndex
        int pivotIndex = partition(left, right, array[right]);
        // sort left side
        recQuickSort(left, pivotIndex - 1);
        // sort right side
        recQuickSort(pivotIndex + 1, right);
    }
}
You can pick the pivot more or less at random. For simplicity we can pick the item on the right end of the subarray to be partitioned.

In order to insert this pivot into its final sorted place it is convenient to:

1- reduce by one the size of the right subarray (up to 'right-1')
2- swap the pivot at 'right' with the 'partition index'
QuickSort: Choosing a pivot value?

- The partition method needs to be modified as follows (using underline):

```java
public int partition(int left, int right, double pivot) {
    // here the array of double (for example) is a private variable of the class
    int leftProbe = left-1;  // becomes left after ++leftProbe
    int rightProbe = right;  // becomes right-1 after --rightProbe

    while (true) {
        // search smaller item from left - No need of extra test
        while (array[++leftProbe]<pivot) { ; }
        // search larger item from right - with extra test if it reaches the end
        while (rightProbe>left && array[--rightProbe]>pivot) { ; }

        if (leftProbe >= rightProbe) // cross-over → end of partitioning process
            break;
        else  // found misplaced items; swap
            swap(leftProbe, rightProbe);
    }

    swap(leftProbe, right);  // insert pivot

    return leftProbe;  // return pivot location
}
```
quickSort - Examples

Step | Recursion Level
--- | ---
1 | 1
2, 8 | 2
3, 7, 9, 12 | 3
4, 10 | 4
5, 6, 11 | 5
quickSort - Examples

call to quickSort: left=0 right=9 array:47 38 86 13 56 52 41 71 33 16
partitioned at pivot 16

call to quickSort: left=0 right=0 array:13 16 86 47 56 52 41 71 33 38

call to quickSort: left=2 right=9 array:13 16 86 47 56 52 41 71 33 38
partitioned at pivot 38

call to quickSort: left=2 right=2 array:13 16 33 38 56 52 41 71 86 47

call to quickSort: left=4 right=9 array:13 16 33 38 56 52 41 71 86 47
partitioned at pivot 47

call to quickSort: left=4 right=4 array:13 16 33 38 41 47 56 71 86 52

call to quickSort: left=6 right=9 array:13 16 33 38 41 47 56 71 86 52
partitioned at pivot 52

call to quickSort: left=6 right=5 array:13 16 33 38 41 47 52 71 86 56

call to quickSort: left=7 right=9 array:13 16 33 38 41 47 52 71 86 56
partitioned at pivot 56

call to quickSort: left=7 right=6 array:13 16 33 38 41 47 52 56 86 71

call to quickSort: left=8 right=9 array:13 16 33 38 41 47 52 56 86 71
partitioned at pivot 71

call to quickSort: left=8 right=7 array:13 16 33 38 41 47 52 56 71 86

call to quickSort: left=9 right=9 array:13 16 33 38 41 47 52 56 71 86
quickSort- Examples

- For fun: https://www.youtube.com/watch?v=ywWBy6J5gz8

They are using a different algorithm for partitioning- pivot starts on the left and only one probe is used to find the pivot correct position- Recursion is however similar.

- To do: Test Java applet quickSort.html
QuickSort: difficulties

- If we assume an inversely sorted array the algorithm becomes very slow...$O(N^2)$...

- The problem comes from our choice for the pivot, which will always produce a subarray of size 1 and another of size $N-1$... so $N$ recursive levels at the end.

- Ideally the pivot should be the median of the items being sorted so the array can be divided by roughly 2 at each recursive level (providing then $\log_2(N)$ levels)

- If the data is truly random, the choice of the right item as pivot is a good one, but when the data is sorted or inversely sorted, this choice of pivot is really bad.

- Can we improve on our approach for selecting the pivot?
  - Yes using the median-of-three partitioning
QuickSort: Median-of-Three partitioning

- Computing the median value directly and pick the correct pivot is not practical (more time consuming than sorting itself).
- A good compromise consists of finding the median of the first, last and middle elements (called the median-of-three approach).

Actually, it is also convenient to directly sort these 3 elements (p347 textbook)
QuickSort: improvements

- 1- Median-three-partitioning in action
  - **To do**: Test Java applet quickSort2.html

- 2- Using an insertion sort procedure for small partitions
  - Hybrid scheme- do Quicksort $O(N \log N)$ until the array is almost sorted and then use InsertionSort which is then $O(N)$
  - We need to decide on a good cut-off (size of the small partition)- Knuth recommends 9
public void recQuickSort(int left, int right) {
    // size <10- base step- insertion sort -
    if (right-left+1<10) {
        insertionSort(left, right); // this step can be commented if you want
    } // to insert sort the entire array after quickSort is done
    // recursive step
    else{
        // partition range using median-of-three pivot and return pivotIndex
        int pivotIndex = partition(left, right, medianOf3(left, right));
        // sort left side
        recQuickSort(left, pivotIndex-1);
        // sort right side
        recQuickSort(pivotIndex+1, right);
    }
}
### Sorting Algorithms... the big picture

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complexity</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble Sort</td>
<td>$O(N^2)$</td>
<td>$O(N^2)$ comp.+ $O(N^2)$ swaps</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Slow, slow, slow</td>
</tr>
<tr>
<td>Selection Sort</td>
<td>$O(N^2)$</td>
<td>$O(N^2)$ comp.+ $O(N)$ swaps</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Intuitive but still slow</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>$O(N^2)$</td>
<td>$O(N^2)$ comp.+ $O(N^2)$ copies</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• half #comp. than Bubble</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• $O(N)$ total if data almost sorted</td>
</tr>
<tr>
<td>Enhanced Insertion Sort</td>
<td>$O(N^2)$</td>
<td>$O(N\log N)$ comp.+ $O(N^2)$ copies</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Use binary search rather than linear search</td>
</tr>
<tr>
<td>List Insertion Sort</td>
<td>$O(N^2)$</td>
<td>$O(N^2)$ comp.+ $O(N)$ copies</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Only 2N copies</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Does not work 'in-place'</td>
</tr>
<tr>
<td>MergeSort</td>
<td>$O(N\log N)$</td>
<td>$O(N)$ copies by $O(\log N)$ levels</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Divide &amp; Conquer + Recursive</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Does not work 'in place'</td>
</tr>
<tr>
<td>ShellSort</td>
<td>$O(N(\log N)^2)$</td>
<td>In average</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 'Insertion sort' using increment</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Worst case not far from average</td>
</tr>
<tr>
<td>QuickSort</td>
<td>$O(N\log N)$</td>
<td>Comp. $\geq$ swaps</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Divide and Conquer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Uses partitioning recursively</td>
</tr>
</tbody>
</table>