## Sorting Algorithms... so far

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complexity</th>
<th>Copies</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bubble Sort</strong></td>
<td>$O(N^2)$</td>
<td>$O(N^2)$ comp.+ $O(N^2)$ swaps</td>
<td>- Slow, slow, slow</td>
</tr>
<tr>
<td><strong>Selection Sort</strong></td>
<td>$O(N^2)$</td>
<td>$O(N^2)$ comp.+ $O(N)$ swaps</td>
<td>- Intuitive but still slow</td>
</tr>
<tr>
<td><strong>Insertion Sort</strong></td>
<td>$O(N^2)$</td>
<td>$O(N^2)$ comp.+ $O(N^2)$ copies</td>
<td>- half #comp. than Bubble</td>
</tr>
<tr>
<td><strong>Enhanced Insertion Sort</strong></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><strong>List Insertion Sort</strong></td>
<td>$O(N^2)$</td>
<td>$O(N^2)$ comp.+ $O(N)$ copies</td>
<td>- Only 2N copies</td>
</tr>
<tr>
<td><strong>MergeSort</strong></td>
<td>$O(N\log N)$</td>
<td>$O(N)$ copies by $O(\log N)$ levels</td>
<td>- Divide &amp; Conquer + Recursive</td>
</tr>
<tr>
<td><strong>ShellSort</strong></td>
<td></td>
<td></td>
<td>Does not work 'in-place'</td>
</tr>
<tr>
<td><strong>QuickSort</strong></td>
<td></td>
<td></td>
<td>Does not work 'in place'</td>
</tr>
</tbody>
</table>
ShellSort- Intro

- Named after Donald L. Shell (discovered in 1959)
- It can be seen as a drastic improvement of the insertion sort algorithm
- It is much faster than simple sorting but not as fast as quicksort
- However, the worst case performance is not significantly worse than the average performance (this may not be the case for quicksort)
- It is sometimes recommended to experiment first with ShellSort before QuickSort
What is the main problem with Insertion sort?

- Suppose a key small items is on the far right: example worst case
  - Too many copies/shifts (using an array)
  - Too many comparisons (using an array with linear search, or linked-list)
  - For the example above, just for a single item you need N comparisons with all the other items on the left and N copies for insertion

How can we improve insertion sort?

- … using less copies/comparisons for a single item
ShellSort- N-Sorting

- Use widely space items (higher increment than 1)- Example with 4

- Once sorted use less widely space items and so on
- At each steps the list becomes ”more sorted” and we minimize the amount of work used by insertion sort
- The last stage uses an increment of 1, which is just insertion sort
ShellSort - Examples

- **Increment 5**
  
  3 0 1 8 7 2 5 4 9 6
  
  2 0 1 8 6 3 5 4 9 7

- **Increment 3**
  
  2 0 1 8 6 3 5 4 9 7
  
  2 0 1 5 4 3 7 6 9 8

- **Increment 1 (insertion sort)**
  
  2 0 1 5 4 3 7 6 9 8
  
  0 1 2 3 4 5 6 7 8 9
ShellSort - Examples

- For fun: https://www.youtube.com/watch?v=CmPA7zE8mx0

- To do: Test Java applet ShellSort.html
ShellSort- Why is it faster than insertion sort?

- When increment is large
  - The number of items to consider ”per sub-list” is small and they can move very long distance... this is quite effective
- When increments grow smaller
  - The number of items to consider per sub-list increases, but the items are already closer to their final sorted position
- Challenge: picking the right increment sequence
  - Shell originally suggested N/2 to start with and then divide in 2 for each pass. Example for N=100, increments of 50,25,12,6,3,1
  - However, it is very likely with this choice to run into worst-case behavior and end up with $O(N^2)$
  - The choice to divide by 2.2 actually improves the behavior (45,20,9,4,1)- provide a better mix with the items previously sorted
  - Most popular sequence was proposed by Knuth 40,13,4,1 starting from 1 it follows: $h=3*h+1$
public void shellSort(double[] array) {
    int n = array.length;
    int inner, outer;
    double temp;
    // find initial value of increment h (overhead for shellSort)
    int h = 1;
    while (h <= n / 3) h = h * 3 + 1;  // sequence (1,4,13,40,121,364,1093)

    while (h > 0) {
        // decreasing h until 1 - use inverse formula (h-1)/3
        for (outer = h; outer < n; outer++) {
            temp = array[outer];
            inner = outer;
            while (inner > h - 1 && array[inner - h] >= temp) {
                array[inner] = array[inner - h];
                inner = inner - h;
                array[inner] = temp;
            }
            h = (h - 1) / 3;  // decrease h
        }
    }
}
ShellSort - Efficiency

- Theoretical analysis is difficult except in special cases
- Based on experiments there are various estimates

<table>
<thead>
<tr>
<th>O() Value + Type of Sort</th>
<th>10 Items</th>
<th>100 Items</th>
<th>1,000 Items</th>
<th>10,000 Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N^2$ Insertion, etc.</td>
<td>100</td>
<td>10,000</td>
<td>1,000,000</td>
<td>100,000,000</td>
</tr>
<tr>
<td>$N^{3/2}$ Shellsort</td>
<td>32</td>
<td>1,000</td>
<td>32,000</td>
<td>1,000,000</td>
</tr>
<tr>
<td>$N^*(\log N)^2$ Shellsort</td>
<td>10</td>
<td>400</td>
<td>9,000</td>
<td>160,000</td>
</tr>
<tr>
<td>$N^{5/4}$ Shellsort</td>
<td>18</td>
<td>316</td>
<td>5,600</td>
<td>100,000</td>
</tr>
<tr>
<td>$N^{7/6}$ Shellsort</td>
<td>14</td>
<td>215</td>
<td>3,200</td>
<td>46,000</td>
</tr>
<tr>
<td>$N^*\log N$ Quicksort, etc.</td>
<td>10</td>
<td>200</td>
<td>3,000</td>
<td>40,000</td>
</tr>
</tbody>
</table>

- Average is $O(N(\log N)^2)$
Notion of Partitioning

- Action of dividing into groups depending on key value (a 'custom' divide)

- **Example:**
  
  You may want to divide employee records into two groups:
  - Employees who live within 15 miles of the office
  - Those who live further away

- After being partitionned the data is not sorted yet; however it is more sorted that it was before

- Partitioning is the underlying mechanism of quicksort

- Note: Partitionning is not stable. Each group is not in the same order as it was originally
Notion of Partitioning

- **To do**: Test Java applet Partition.html

**Remark**: The partitioning process does not necessarily divide the list in half, that depends on the pivot/key value. There may be more items in one group than in the other.
Notion of Partitioning- Algorithm

- It uses two probes/pointers at the opposite ends of the array
  - The probe from the left moves towards the right
  - The probe from the right moves toward the left
- They are both incremented/decremented
  - Left probe stops when it encounters an item larger than pivot
  - Right probe stops when it encounters an item smaller than pivot
- Both items are swapped
- The process stops when the probes cross
- Extra-tests necessary to check special case (if all data are below or above the pivot, one of the probe would go all the way beyond the limit of the array)
- The partitioning algorithm is $O(N)$ (but fewer swaps than comparisons)
private static void partition(double[] array, double pivot) {
    int left = -1;
    int right = array.length;

    while (true) {
        while (left < array.length && array[++left] < pivot) { ; } // search from left
        while (right > 0 && array[--right] > pivot) { ; } // search from right

        if (left >= right) { // cross-over indicates end of partitioning process
            break;
        }
        else { // found misplaced items; swap
            double temp = array[left];
            array[left] = array[right];
            array[right] = temp;
        }
    }
}

Note: array[++i] means i=i+1 and then array[i]
This order is reversed for array[i++]
Logistic

Project 1- average is 64.5

Change of rules for projects- New deadline for preliminary version (bonus point)