Hands-On: Sorting

This activity focuses on four common sorting algorithms and sample implementations. After completing this activity you should

- Understand the selection sort algorithm.
- Understand an implementation of selection sort in a Java method.
- Understand the insertion sort algorithm.
- Understand an implementation of insertion sort in a Java method.
- Understand the quicksort algorithm.
- Understand an implementation of quicksort in a Java method.
- Understand the merge sort algorithm.
- Understand an implementation of merge sort in a Java method.

You will need the following files to complete this activity.

- Sorts.java
- Sorts.selection_sort.jgrasp_canvas.xml
- Sorts.insertion_sort.jgrasp_canvas.xml
- Sorts.quicksort.jgrasp_canvas.xml
- Sorts.merge_sort.jgrasp_canvas.xml

Watching sorts work

Sorting was the subject of one of the first attempts at software visualization (see Ron Baecker's Sorting Out Sorting from 1981). The behavior of sorting algorithms lends itself well to a visual depiction, so canned animations of sorting remain popular. Below are three good examples.

- `http://www.sorting-algorithms.com/`
- `https://visualgo.net/en/sorting?slide=1`
- `https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html`

By using jGRASP and taking advantage of its *Viewer Canvas* you can get the advantages of algorithm animations in the context of *your own code*. The `Sorts.java` file contains implementations of all the sorting algorithms discussed in lecture. The XML files are jGRASP *Canvas* files - layout specification and other settings for pre-configured visualizations of each of the four main sorts.

The following sections can be done in any sequence. The instructions are written so that each section is self contained.

Selection sort

Selection sort is a conceptually simple sorting algorithm, but it is not scalable, with *O(N^2)* time complexity in the best, average, and worst cases. Here's a description from Wikipedia:

> The algorithm divides the input list into two parts: the sublist of items already sorted, which is built up from left to right at the front (left) of the list, and the sublist of items remaining to be sorted that occupy the rest of the list. Initially, the sorted sublist is empty and the unsorted sublist is the entire input list. The algorithm proceeds by finding the smallest (or largest, depending on sorting order) element in the unsorted sublist, exchanging (swapping) it with the leftmost unsorted element (putting it in sorted order), and moving the sublist boundaries one element to the right.

1. Open `Sorts.java` in jGRASP and compile it.
2. Click on the *Run in Canvas* button in the toolbar.
3. In the "Choose Canvas File" dialog that opens, select `Sorts.selection_sort.jgrasp_canvas.xml` and click *Ok*.
4. If necessary, resize the Canvas window that opens to make sure it fits on your screen appropriately.
5. Click on the step-over button in the Canvas window's toolbar until the statement `st.selectionSort(a);` is highlighted.
6. Click on the step-in button in the Canvas window's toolbar. Note that the viewers on the Canvas are populated with their current values.
7. Step-over each statement and observe its effect in the Canvas. Continue to do this until you get a good sense of the behavior of this method and the underlying algorithm.
8. Click on the statement `swap(a, i, min);` so that the cursor is somewhere on this line.
9. Predict what the value of `min` will be once execution reaches this statement.
10. Click on the *Run to Cursor* button in the Canvas toolbar and use the Canvas to confirm your prediction.
11. Continue to interact with the Canvas until you feel as though you have a solid understanding of selection sort.
12. Click on the "End" button in the jGRASP Run I/O controls at the bottom of the jGRASP desktop to stop the debugger from running.
13. Close the Canvas window.

Insertion sort

Insertion sort is a conceptually simple sorting algorithm with time complexity that varies according to how the input is arranged initially. If the input is either in reverse order or random order, insertion sort has *O(N^2)* time complexity and is thus not scalable to large data sets. But if the input is sorted or - more importantly - *almost* sorted, then insertion sort has *O(N)* time complexity and is very useful on large inputs. Here's a description from Wikipedia:

> Insertion sort iterates, consuming one input element each repetition, and growing a sorted output list. Each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.
>
> Sorting is typically done in-place, by iterating up the array, growing the sorted list behind it. At each array-position, it checks the value there against the largest value in the sorted list (which happens to be next to it, in the previous array-position checked). If larger, it leaves the element in place and moves to the next. If smaller, it finds the correct position within the sorted list, shifts all the larger values up to make a space, and inserts into that correct position.

1. Open `Sorts.java` in jGRASP and compile it.

2. Click on the *Run in Canvas* button in the toolbar.

3. In the "Choose Canvas File" dialog that opens, select `Sorts.insertion_sort.jgrasp_canvas.xml` and click *Ok*.

4. If necessary, resize the Canvas window that opens to make sure it fits on your screen appropriately.

5. Click on the step-over button ⇩ in the Canvas window's toolbar until the statement `st.insertionSort(a);` is highlighted.

6. Click on the step-in button ↳ in the Canvas window's toolbar. Note that the viewers on the Canvas are populated with their current values.

7. Step-over ⇩ each statement and observe its effect in the Canvas. Continue to do this until you get a good sense of the behavior of this method.

8. Make sure you can predict what value `j` will cause the inner loop to stop on any given iteration of the outer loop.

9. Continue to interact with the Canvas until you feel as though you have a solid understanding of insertion sort.

10. Click on the “End” button in the jGRASP Run I/O controls at the bottom of the jGRASP desktop to stop the debugger from running.

11. Close the Canvas window.

Quicksort

The *quicksort* algorithm isn’t as conceptually simple as insertion sort or selection sort, but it is far more efficient. Although it has a worst-case time complexity that is *O(N^2)*, quicksort’s average and best case time complexity is *O(N log N)*. The worst-case can typically be avoided, so quicksort is one of the most commonly used sorting algorithms, even for large data sets. Here’s a description from Wikipedia.

> Quicksort is a divide and conquer algorithm. Quicksort first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quicksort can then recursively sort the sub-arrays.
>
> The steps are:
>
> 1. Pick an element, called a pivot, from the array.
> 2. Partitioning: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
> 3. Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

1. Open `Sorts.java` in jGRASP and compile it.

2. Click on the *Run in Canvas* button in the toolbar.

3. In the “Choose Canvas File” dialog that opens, select `Sorts.quicksort.jgrasp_canvas.xml` and click *Ok*.

4. If necessary, resize the Canvas window that opens to make sure it fits on your screen appropriately.

5. Click on the step-over button in the Canvas window’s toolbar until the statement `st.quicksort(a);` is highlighted.

6. Click on the step-in button in the Canvas window’s toolbar. Note that the viewers on the Canvas are populated with their current values.

7. Step-over the `shuffle(a);` statement and pause when the `qsort(a, 0, a.length - 1);` statement is highlighted.

8. Step-in to the call to `qsort` .
9. Step-over the `if` statement and pause when the call to `partitionOnPivot` is highlighted.
    - This method will *partition* the array `a` around the given *pivot* value, which is designated in this call as `a[lo]` .
    - Note that shuffling the array before calling `qsort` doesn't guarantee that `a[lo]` is a *good* pivot value. So, here are some important questions to think about.
        - What makes a *good* pivot value?
        - What makes a *bad* pivot value?
        - What would be the *best* pivot value for a given array?
        - If the shuffling doesn't guarantee good pivot values, what does it do?
    - Now look at the Canvas to see what the pivot value will be and predict what the array will look like after the partitioning is performed.
10. Step-over the call to `partitionOnPivot` and confirm your prediction with the Canvas. **Note that the pivot value is now in its final sorted position.**
11. At this point the debugger is stopped on the first of two recursive calls to `qsort` . This first call will recursively apply this process to the left half of the array; that is, it will sort `a[lo]` through `a[j-1]` .
12. Step-over the first call to `qsort` and see the effect reflected in the Canvas.
13. The second recursive call to `qsort` will sort the right half of the array; that is, it will sort `a[j+1]` through `a[hi]` . (Remember that `a[j]` holds the pivot and it's already in the correct sorted position.)
14. Step-over the second call to `qsort` and see the effect reflected in the Canvas.
15. Continue to interact with the Canvas until you feel as though you have a solid understanding of quicksort.
    - We will revisit this algorithm at the point in the course where we more fully cover recursion. But, if you're interested, you can step-in to the recursive calls and watch the entire sorting process occur step by step.
16. Click on the "End" button in the jGRASP Run I/O controls at the bottom of the jGRASP desktop to stop the debugger from running.
17. Close the Canvas window.

Partition

The actual work of rearranging elements of the array in quicksort is accomplished by *partitioning* array elements around a *pivot* value. It's important that you understand this process in detail. There are different strategies for partitioning, and different strategies for selecting the pivot value. For this course both strategies are simple and straightforward.

1. Open `Sorts.java` in jGRASP and compile it.
2. Click on the *Run in Canvas* button in the toolbar.
3. In the "Choose Canvas File" dialog that opens, select `Sorts.quicksort.jgrasp_canvas.xml` and click *Ok*.
4. If necessary, resize the Canvas window that opens to make sure it fits on your screen appropriately.
5. Click on the step-over button in the Canvas window's toolbar until the statement `st.partitionOnPivot(a, 0, a.length - 1, 5);` is highlighted.
6. Step-in to the call to `partitionOnPivot` .
7. Step-over the statements in `partitionOnPivot` and observe their behavior in the Canvas window.
8. After you have observed the method's execution to completion, change the last parameter to `partitionOnPivot` to a different value and repeat all the steps above.
9. Continue to experiment with different choices of the pivot value until you are confident that you understand both the purpose of this method and the manner in which it accomplishes this purpose.
10. Click on the "End" button in the jGRASP Run I/O controls at the bottom of the jGRASP desktop to stop the debugger from running.
11. Close the Canvas window.

Merge sort

The *merge sort* algorithm isn't as conceptually simple as insertion sort or selection sort, but it is far more efficient. Merge sort's time complexity is *O(N log N)* in all cases, and it is therefore a very scalable algorithm. In fact, merge sort is one of the most commonly used sorting algorithms, even for large data sets. Here's a description from Wikipedia:

> Conceptually, a merge sort works as follows:
>
> 1. Divide the unsorted list into n sublists, each containing 1 element (a list of 1 element is considered sorted).

> 2. Repeatedly merge sublists to produce new sorted sublists until there is only 1 sublist remaining. This will be the sorted list.

1. Open `Sorts.java` in jGRASP and compile it.
2. Click on the *Run in Canvas* button in the toolbar.
3. In the "Choose Canvas File" dialog that opens, select `Sorts.merge_sort.jgrasp_canvas.xml` and click *Ok*.
4. If necessary, resize the Canvas window that opens to make sure it fits on your screen appropriately.
5. Click on the step-over button in the Canvas window's toolbar until the statement `st.mergesort(a);` is highlighted.
6. Click on the step-in button in the Canvas window's toolbar. Note that the viewers on the Canvas are populated with their current values.
7. Step-over the statements of `mergesort` until `msort(a, 0, a.length - 1);` .{java} is highlighted.
8. Step-in to the call to `msort` .
9. Step-over the statements of `msort` and observe their effects in the Canvas window.
10. Continue to interact with the Canvas until you feel as though you have a solid understanding of merge sort.
    - We will revisit this algorithm at the point in the course where we more fully cover recursion. But, if you're interested, you can step-in to the recursive calls and watch the entire sorting process occur step by step.
11. Click on the "End" button in the jGRASP Run I/O controls at the bottom of the jGRASP desktop to stop the debugger from running.
12. Close the Canvas window.

Merge

The actual work of rearranging elements of the array in merge sort is accomplished by *merging* elements from two sorted halves of an array into another array. It's important that you understand this process in detail. Note that merging typically requires a lot of extra memory - a second array of the same size as the original.

1. Open `Sorts.java` in jGRASP and compile it.
2. Click on the *Run in Canvas* button in the toolbar.

3. In the “Choose Canvas File” dialog that opens, select `Sorts.merge_sort.jgrasp_canvas.xml` and click *Ok*.
4. If necessary, resize the Canvas window that opens to make sure it fits on your screen appropriately.
5. Click on the step-over button in the Canvas window’s toolbar until the statement `st.merge(a, lo, mid, hi);` is highlighted.
    - Note that `a` has been arranged so that `a[lo]..a[mid]` is sorted and `a[mid + 1]..a[hi]` is also sorted. These are the two sorted halves that need to be merged into one sorted whole.
6. Step-in to the call to `merge` .
7. Step-over the statements in `partitionOnPivot` and observe their behavior in the Canvas window.
8. After you have observed the method’s execution to completion, change the values in the array parameter to `merge` and repeat all the steps above.
9. Continue to experiment with different arrangements of the values in the array until you are confident that you understand both the purpose of this method and the manner in which it accomplishes this purpose.
10. Click on the “End” button in the jGRASP Run I/O controls at the bottom of the jGRASP desktop to stop the debugger from running.
11. Close the Canvas window.

Submission

The submission page for this activity asks you to apply your understanding of these sorting algorithms to a problem and then submit it for a grade.