11. Searching and Sorting (Ch10)

- Objectives - when we have completed this set of notes, you should be familiar with:
  - Linear Search
  - Binary Search
  - Selection Sort
  - Insertion Sort

Searching

- Finding a specific element in a group of items (*search pool*)
  - A student with a certain name in an array of Student objects
  - A bank account with a certain SS# in an online database
- Consideration:
  - What if the target is not present? (i.e., will not be found)
- Which search will be the most *efficient*?
- If class implements the Comparable interface, then two objects can be compared using the compareTo method (also a class must implement the Comparable interface in order for Collections.sort to work on objects of the class)

Linear Search

- You have likely used linear searches in COMP 1210 for your projects
- Examines each element in a group starting at the first
- Elements do not have to be in a specific order
  - Must consider how you will find the specific item
- Search ends when…
  - The element is found
  - The end of the list is reached
- NumberLinearSearch.java

Linear Search

- Easy to implement
- The search pool does not have to be sorted
- Why not always use linear search?
- If the target is not present, you must go to the end of the search pool
  - Not good if the size of the search pool is large
- What if your target is near the end of the list?
  - Each comparison takes a certain amount of time, which can add up to minutes, days, weeks, or longer on a large database

Binary Search

- An efficient alternative to linear search in some cases
- The search pool **must be sorted** first
- Starts at the middle of the candidate pool and then eliminates one half of the pool each time
- For 2^n items, at most n compares are required
  - $16 = 2^4$ so at most 4 compares
  - 4 billion is approx = 2^{32} so at most 32 compares

Binary Search

- Suppose that you are searching for the number 59 in the following array...

2	4	9	23	27	29	30	34	45	59	67	76	89	92	97

- Binary search starts at the middle.

2	4	9	23	27	29	30	34	45	59	67	76	89	92	97

- 59 > 34, so the entire left side can be eliminated (cuts the candidate pool in half)

2	4	9	23	27	29	30	34	45	59	67	76	89	92	97

Binary Search

- Start again with the second half

2	4	9	23	27	29	30	34	45	59	67	76	89	92	97

- 59 < 76, so the right half of the new candidate pool can be eliminated

2	4	9	23	27	29	30	34	45	59	67	76	89	92	97

- 59 is the middle of the new candidate pool and is returned

2	4	9	23	27	29	30	34	45	59	67	76	89	92	97

Binary Search

- Binary search would have taken 3 comparisons to find the element

2	4	9	23	27	29	30	34	45	59	67	76	89	92	97

- Linear search would have taken 10

2	4	9	23	27	29	30	34	45	59	67	76	89	92	97

- Imagine if there were 4,000,000,000 elements in the search pool … at most 32 compares
- NumberBinarySearchArray.java

Binary Search

- Why not always use binary search?
- Recall that it requires a sorted candidate pool
- Great if objects are pre-sorted, but sorting can also take a large number of comparisons
- You'll learn about many more sorting algorithms later in the CSSE program

Sorting

- We've said that sorting also takes comparisons
- When you invoke Collections.sort or Arrays.sort the code has been written for you, but it will still take time to perform the sort
- How are elements sorted?
- Which sort is 'better' given certain conditions?
- Let's look at two sorting algorithms:
  - Selection Sort
  - Insertion Sort

Selection Sort

- The approach of Selection Sort:
  - select a value and put it in its final place into the list
  - repeat for all other values
- In more detail:
  - find the smallest value in the list
  - switch it with the value in the first position
  - find the next smallest value in the list
  - switch it with the value in the second position
  - repeat until all values are in their proper places

Selection Sort

- An example: ⟷ Indicates that the two elements should be swapped

```
original:        3   9   6   1   2
smallest is 1:   1   9   6   3   2
smallest is 2:   1   2   6   3   9
smallest is 3:   1   2   3   6   9
smallest is 6:   1   2   3   6   9
```

- Each time, the smallest remaining value is found and exchanged with the element in the "next" position to be filled

Swapping

- The processing of the selection sort algorithm includes the *swapping* of two values

- Swapping requires three assignment statements and a temporary storage location:

```
temp = first;
first = second;
second = temp;
```

- See NumberSelectionSort.java
- See NumberSelectionSortExample.java

Insertion Sort

- The approach of Insertion Sort:
  - pick any item and insert it into its proper place in a sorted sublist
  - repeat until all items have been inserted
- In more detail:
  - consider the first item to be a sorted sublist (of one item)
  - insert the second item into the sorted sublist, shifting the first item as needed to make room to insert the new addition
  - insert the third item into the sorted sublist (of two items), shifting items as necessary
  - repeat until all values are inserted into their proper positions

Insertion Sort

- An example:

Indicates where the next element should be inserted

original:	3	9	6	1	2
insert 9:	3	9	6	1	2
insert 6:	3	6	9	1	2
insert 1:	1	3	6	9	2
insert 2:	1	2	3	6	9

- See NumberInsertionSort.java

Comparing Sorts

- The Selection and Insertion sort algorithms are similar in efficiency
- They both have outer loops that scan all elements, and inner loops that compare the value of the outer loop with almost all values in the list
- Approximately n^2 number of comparisons are made to sort a list of size n -> *order* n^2
- Other sorts can be more efficient: *order* $n \log_2 n$