Searching and Sorting

- 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* or *search space*)
  - A student with a certain name in an array of Student objects
  - A bank account with a certain account number in an online database
- Considerations:
  - What if the target is not present? (i.e., will not be found)
  - Which search will be the most efficient?
- If a class implements the Comparable interface, then two objects can be compared using the compareTo method (also implementing the Comparable interface means Collections.sort or Arrays.sort will work on the list or array)

Linear Search

- You have used linear searches in your projects
- Examines each element in a group starting with the first element
- 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 – runs in *O*(n) time
- 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 seconds, minutes, or longer if the search space is extremely large

Binary Search

- A binary search can be an efficient alternative to a linear search
- However, the search pool **must be sorted**
- Starts at the middle of the candidate pool and either finds the target or eliminates one half of the pool for the next iteration
- For 2^n items, at most n iterations are required
  - $2^4 = 16$ so at most 4 iterations
  - $2^{32} = 4{,}294{,}967{,}296$ so at most 32 iterations to search a list of over 4 billion
  - Runs in $O(\log_2 n)$ time

Binary Search

- Suppose the target of the search is 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.

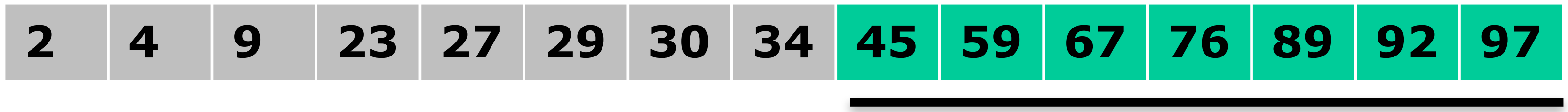
							↓							
2	4	9	23	27	29	30	34	45	59	67	76	89	92	97

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

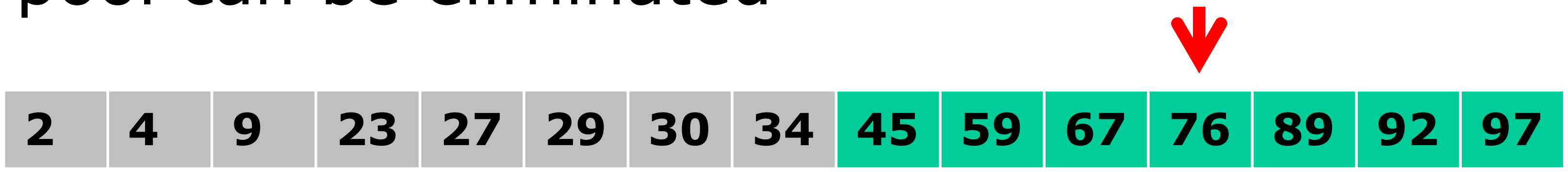
2	4	9	23	27	29	30	34	45	59	67	76	89	92	97

Binary Search

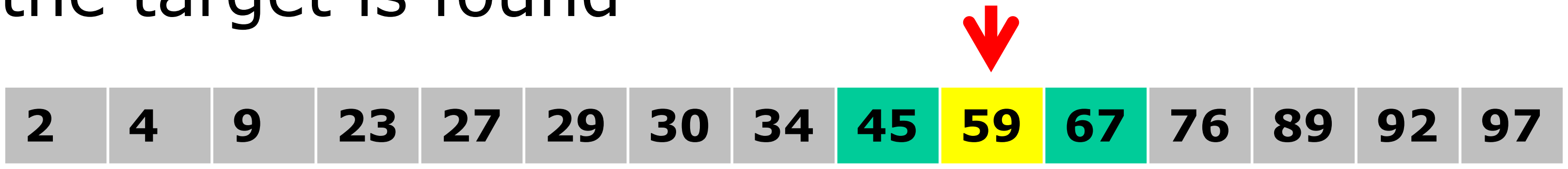
- Start again with the second half



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



- 59 is the middle of the new candidate pool and the target is found



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 iterations
- 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 iterations and/or comparisons
- You'll learn about many sorting algorithms later in your data structures and algorithms courses
- Next, we'll take a look at the details of two simple sorting algorithms

Sorting

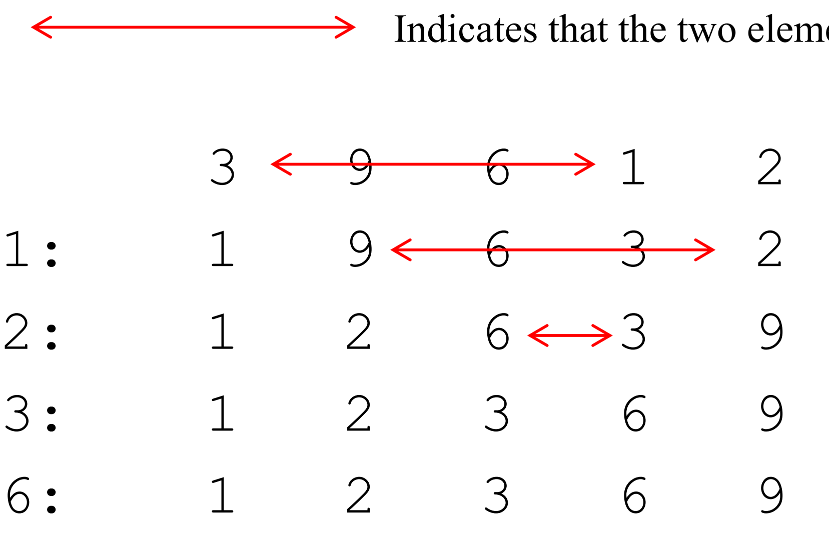
- Sorting is the process of placing items in order based on one or more sort keys.
- When you invoke Collections.sort or Arrays.sort, the code has been written for you, but it will still take time to perform the sort
- How are elements sorted?
- Which sort is “better” given certain conditions?
- Let’s 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 in 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 Sorting.java
- See NumberSelectionSortExample.java

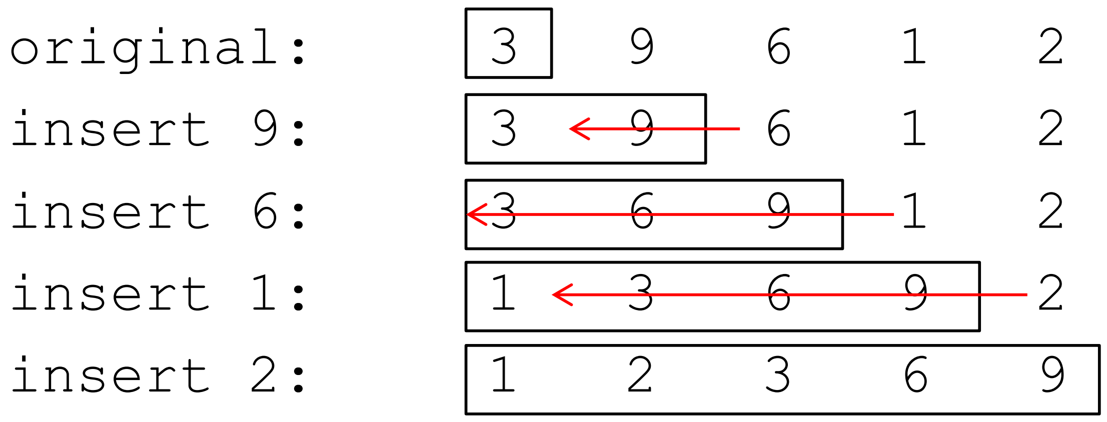
Insertion Sort

- 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



- See NumberInsertionSort.java Sorting.java

Comparing Sorts

- The algorithms for Selection sort and Insertion sort are similar in efficiency
- They both have outer loops that scan essentially all elements, and inner loops that compare the value at the outer loop index with almost all values in the list
- Approximately n^2 iterations or comparisons are required to sort a list of size n, which translates to $O(n^2)$ time – worst case
- Other sorting algorithms are more efficient: e.g., Heap sort: $O(n \log_2 n)$ time – worst case

Polymorphism

- The Searching.java and Sorting.java classes provide a nice introduction to these topics
  - linearSearch and binarySearch methods
  - selectionSort and insertionSort methods
- These methods demonstrate the use and power of polymorphism
  - equals method
  - compareTo method
- Bottom line – Polymorphism is truly an important concept in OOD/OOP