Searching and Sorting

- Searching algorithms with simple arrays
- Sorting algorithms with simple arrays
  - Selection Sort
  - Insertion Sort
  - Bubble Sort
  - Quick Sort
  - Merge Sort
- Introduction to Project 3
- Reading for this lecture: L&C 8.1-8.2

Searching

- Searching is the process of finding a target element within a group of items called the search pool
- The target may or may not be in the search pool
- We want to perform the search efficiently, minimizing the number of comparisons
- Let's look at two classic searching approaches: linear search and binary search
- We'll implement the searches with polymorphic `Comparable` objects

Linear Search

- A linear search begins at one end of the search pool and examines each element
- Eventually, either the item is found or the end of the search pool is encountered
- On average, it will search half of the pool
- This algorithm is O(n) for finding a single element or determining it is not found

Binary Search

- A *binary search* assumes the list of items is sorted
- Each pass eliminates a large part of the search pool with one comparison (“20 questions game”)
- A binary search examines the middle element of the list -- if the target is found, the search is over
- The process continues by comparing the target to the middle of the remaining *viable candidates*
- Eventually, the target is found or there are no remaining *viable candidates* (and the target has not been found)

Recursive Implementation

- When we studied binary search in CS110, we used a loop and narrowed the range of the portion of the array we were searching
- Now that we understand recursion, we can study a recursive implementation:

```
boolean found = binarySearch
    (array, 0, data.length - 1, element);


public boolean binarySearch
    (T[] data, int min, int max, T target)
```

Binary Search

- Recursive implementation:

```
{
 boolean found = false;
 int mid = (min + max) / 2;
 if (data[mid].compareTo(target) == 0)
   found = true;
 else if (data[mid].compareTo(target) > 0) {
   if (min <= mid - 1)
     found = binarySearch(data, min, mid-1, target);
 }
 else if (mid + 1 <= max)
   found = binarySearch(data, mid+1, max, target);
 return found;
}
```

Sorting

- *Sorting* is the process of arranging a list of items in a particular order
- The sorting process is based on specific value(s)
  - Sorting a list of test scores in ascending numeric order
  - Sorting a list of people alphabetically by last name
- Sequential sorts $O(n^2)$: Selection, Insertion, Bubble
- Logarithmic sorts O(nlogn): Quick, Merge
- Again, we'll implement the sorts with polymorphic `Comparable` objects
- Note: Text uses parameterized static methods, but the code of the methods is the same

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:

```
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
```

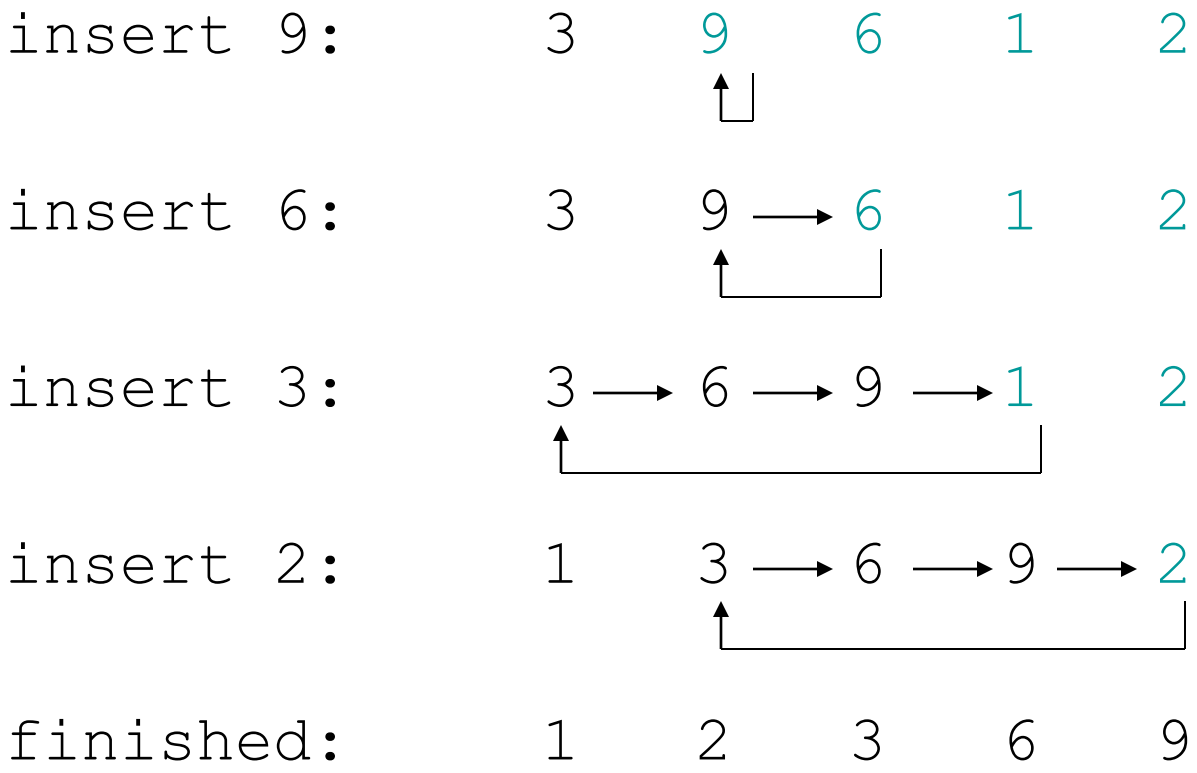
- Each time, the smallest remaining value is found and exchanged with the element in the "next" position to be filled

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:



Bubble Sort

- Bubble sort algorithm sorts the values by repeatedly comparing neighboring elements
- It swaps their position if they are not in order
- Each pass through the algorithm moves the largest value to its final position
- A pass may also reposition other elements
- May be more efficient if looping is stopped when no changes were made on last pass

Bubble Sort

- An example of one pass:

```
Original:     9   6   8  12   3   1   7
Swap 9 & 6:   6   9   8  12   3   1   7
Swap 9 & 8:   6   8   9  12   3   1   7
No swap:      6   8   9  12   3   1   7
Swap 12 & 3:  6   8   9   3  12   1   7
Swap 12 & 1:  6   8   9   3   1  12   7
Swap 12 & 7:  6   8   9   3   1   7  12
```

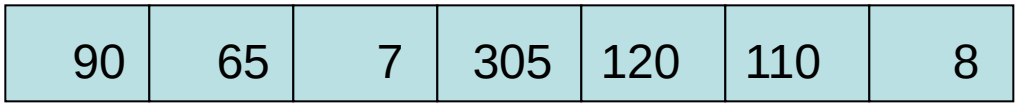
- Each pass moves largest to last position
- Each pass can iterate one less time than last

Quick Sort

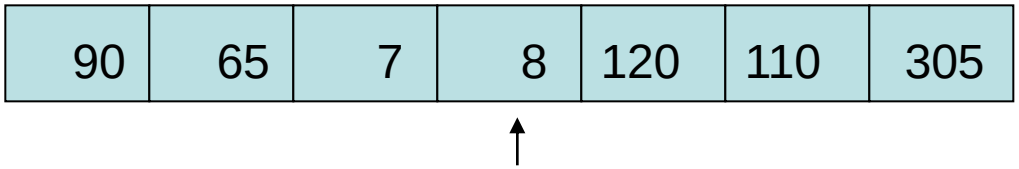
- The quick sort algorithm is a “divide and conquer” algorithm
- It compares the data values to a partition element while partitioning into two sub-lists
- Then, it recursively sorts the sub-lists on each side of the partition element
- The recursion base case is a list containing only 1 element (which is inherently sorted)
- A simplistic choice of the partition element is the first element, but that may not be best

Quick Sort

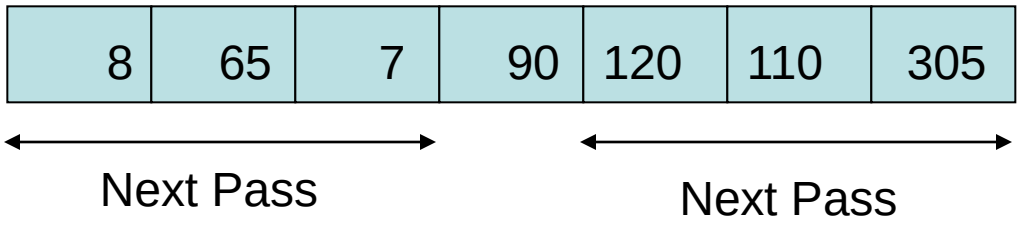
Initial Data (Select first element as the partition element)



Move all data below partition element value to the left
Move all data above partition element value to the right



Swap first element with element at the split point
Partition element is now in its correct position



Quick Sort

- If the list is already (or nearly) sorted, the first element is a poor choice as partition element
- The two partitioned sub-lists are lopsided
  - One may contain only one element
  - The other may contain all the other elements
- In this case, the quick sort is not so quick
- A better choice might be the middle element of the list, but note that there is still an initial order for the elements that is “pathological”

Merge Sort

- Merge sort is another “divide and conquer” algorithm with a different division strategy
  - Cut the array of data in half
  - Sort the left half (recursively calling itself)
  - Sort the right half (recursively calling itself)
  - Merge the two sorted halves of the array
- The merge process for two arrays that are already sorted is only O(n) and we perform O(log n) merges

Merge Sort

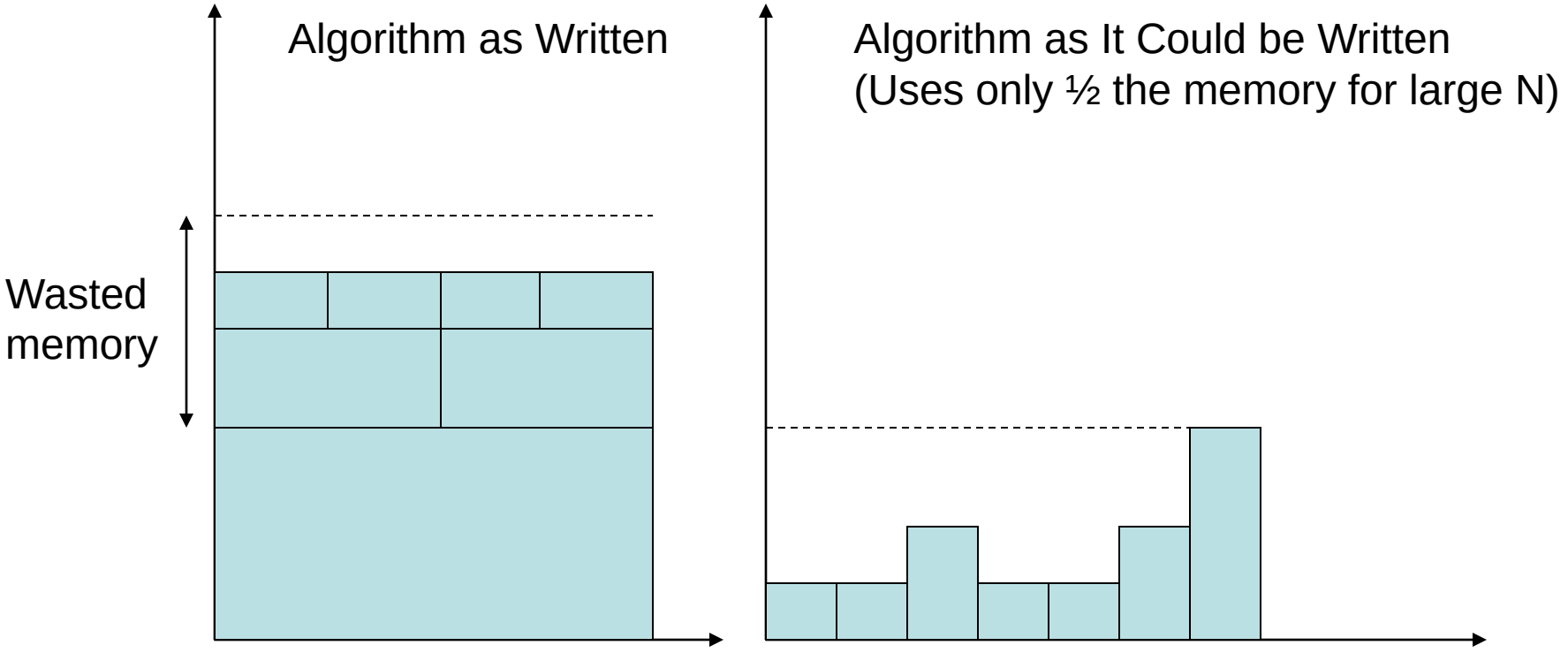
- All comparisons are in the merge process
- Copy all elements into a workspace array
- Copy them back into the original array
  - If there are elements remaining in both halves, compare first elements of each and copy one
  - If there are no elements remaining in left half, copy remainder of right half
  - If there are no elements remaining in right half, copy remainder of left half

Faster Sorts / Memory Usage

- Note: the two faster sorts use more memory
- All of the sort algorithms use the data array, a temp location to do a 3-way swap, and a few variables to control the loop / recursion
- What additional memory is used in the Quick Sort algorithm?
- What additional memory is used in the Merge Sort algorithm?

Text's Merge Sort Memory Usage

The text's code allocates memory for the `T []` before the recursive calls and only needs to use that memory after the recursive calls. This uses more memory than would be required with different design.



Introduction to Project 3

- In Project 3, you will experiment with and observe the performance of various sorting algorithms applied to data in different states of organization
- Some of the data will be organized randomly, some will be already sorted (or almost sorted), and some will be sorted in exactly in the opposite order of the desired order
- You will learn that different sort algorithms have different performance in these different situations

Introduction to Project 3

- You will be provided the L&C example code for the class “SortingAndSearching” which contains multiple sorting methods each of which uses a different algorithm
- You will also be provided a skeleton of the main class that reads a file and sets up its data in an array for sorting
- You will be provided with some test data files that present different situations for sorting

Introduction to Project 3

- “Instrument” the code of the L&C class so that you can determine how many comparisons each performs in the process of sorting a data file
  - Add/modify attributes and/or methods as needed
- Add code in the main class to sort the data files using each algorithm and record the results
- Modify some of the sort algorithms and study comparison with the unmodified versions
- Write a report that compares your results to the Big-O notation behavior that you expected for each algorithm and explain all your observations