13.1 – Linear Search

- A linear search simply examines each item in the search pool, one at a time, until either the target is found or until the pool is exhausted.
- This approach does not assume the items in the search pool are in any particular order.
- We just need to be able to examine each element in turn (in a linear fashion).
- It's fairly easy to understand, but not very efficient.

13.1 – The linearSearch Algorithm

```java
public static Comparable linearSearch (Comparable[] data, Comparable target) {
    Comparable result = null;
    int index = 0;
    while (result == null && index < data.length) {
        if (data[index].compareTo(target) == 0) {
            result = data[index];
        }
        index++;
    }
    return result;
}
```

13.1 – Binary Search

- If the search pool is sorted, then we can be more efficient than a linear search.
- A binary search eliminates large parts of the search pool with each comparison.
- Instead of starting the search at one end, we begin in the middle.
- If the target isn’t found, we know that if it is in the pool at all, it is in one half or the other.
- We can then jump to the middle of that half, and continue similarly.

13.1 – The binarySearch Algorithm

```java
public static Comparable binarySearch (Comparable[] data, Comparable target) {
    Comparable result = null;
    int first = 0, last = data.length-1, mid;
    while (result == null && first <= last) {
        mid = (first + last) / 2; // determine midpoint
        if (data[mid].compareTo(target) == 0) {
            result = data[mid];
        } else if (data[mid].compareTo(target) > 0) {
            last = mid - 1;
        } else {
            first = mid + 1;
        }
    }
    return result;
}
```
13.2 – Sorting

- Sorting is the process of arranging a group of items into a defined order based on particular criteria.
- Many sorting algorithms have been designed.
- **Sequential sorts** typically require $n^2$ comparisons to sort $n$ elements.
- **Logarithmic sorts** typically require about $n \log_2 n$ comparisons to sort $n$ elements.
- As with searching, we must be able to compare one element to another.

13.2 – SortPlayerList.java

```java
//********************************************************************
// SortPlayerList.java       Java Foundations
// Demonstrates a selection sort of Comparable objects.
//********************************************************************
public class SortPlayerList {
    public static void main (String[] args) {
        Contact[] players = new Contact[7];
        players[0] = new Contact("Rodger", "Federer", "610-555-7384");
        players[1] = new Contact("Andy", "Roddick", "215-555-3827");
        players[2] = new Contact("Maria", "Sharapova", "733-555-2969");
        players[3] = new Contact("Venus", "Williams", "663-555-3984");
        players[5] = new Contact("Eleni", "Daniilidou", "322-555-2284");
       Sorting.selectionSort(players);
        for (Comparable player : players)
            System.out.println (player);
    }
}
```
13.2 – Illustration of selection sort processing

Scan right starting with 3. 1 is the smallest, exchange 1 and 3.

Scan right starting with 9. 2 is the smallest, exchange 9 and 2.

Scan right starting with 6. 3 is the smallest, exchange 6 and 3.

Scan right starting with 6. 6 is the smallest, exchange 6 and 6.

13.2 – The selectionSort Algorithm

```java
public static void selectionSort (Comparable[] data)
{
    int min;
    for (int index = 0; index < data.length-1; index++)
    {
        min = index;
        for (int scan = index+1; scan < data.length; scan++)
            if (data[scan].compareTo(data[min]) < 0)
                min = scan;
        swap (data, min, index);
    }
}
```

13.2 – Insertion Sort

- Insertion sort orders a list of values by repetitively inserting a particular value into a sorted subset of the list
- More specifically
  - consider the first item to be a sorted sublist of length 1
  - insert the second item into the sorted sublist, shifting the first item if needed
  - insert the third item into the sorted sublist, shifting the other items as needed
  - repeat until all values have been inserted into their proper positions

13.2 – Illustration of insertion sort processing

3 is sorted. Shift nothing, insert 9.

3 and 9 are sorted. Shift 9 to the right, insert 6.

3, 6, and 9 are sorted. Shift 9, 6, and 3 to the right, insert 1.

1, 3, 6, and 9 are sorted. Shift 9, 6, and 3 to the right, insert 2.
13.2 – The insertionSort Algorithm

```java
public static void insertionSort (Comparable[] data) {
    for (int index = 1; index < data.length; index++) {
        Comparable key = data[index];
        int position = index;
        // Shift larger values to the right
        while (position > 0 && data[position-1].compareTo(key) > 0) {
            data[position] = data[position-1];
            position--;
        }
        data[position] = key;
    }
}
```

13.2 – Bubble Sort

- Bubble sort orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary.
- More specifically:
  - scan the list, exchanging adjacent elements if they are not in relative order; this bubbles the highest value to the top
  - scan the list again, bubbling up the second highest value
  - repeat until all elements have been placed in their proper order

```java
public static void bubbleSort (Comparable[] data) {
    int position, scan;
    for (position = data.length - 1; position >= 0; position--) {
        for (scan = 0; scan <= position - 1; scan++)
            if (data[scan].compareTo(data[scan+1]) > 0) {
                swap (data, scan, scan+1);
            }
    }
}
```

13.2 – Quick Sort

- Quick sort orders a list of values by partitioning the list around one element, then sorting each partition:
  - choose one element in the list to be the partition element
  - organize the elements so that all elements less than the partition element are to the left and all greater are to the right
  - apply the quick sort algorithm (recursively) to both partitions
- It would be nice if the partition element divided the list roughly in half
- QS has two methods
  - quickSort – performs the recursive algorithm
  - partition – rearranges the elements into two partitions

```java
public static void quickSort (Comparable[] data) {
    // Quick sort
}
```
13.2 – The quickSort Algorithm

```java
public static void quickSort (Comparable[] data, int min, int max) {
    int pivot;
    if (min < max) {
        pivot = partition (data, min, max); // make partitions
        quickSort(data, min, pivot-1); // sort left partition
        quickSort(data, pivot+1, max); // sort right partition
    }
}
```

13.2 – The partition Algorithm

```java
private static int partition (Comparable[] data, int min, int max) {
    // Use first element as the partition value
    Comparable partitionValue = data[min];
    int left = min;
    int right = max;
    while (left < right) {
        // Search for an element that is > the partition element
        while (data[left].compareTo(partitionValue) <= 0 && left < right)
            left++;
        // Search for an element that is < the partition element
        while (data[right].compareTo(partitionValue) > 0)
            right--;
        if (left < right) {
            swap(data, left, right);
        }
    }
    // Move the partition element to its final position
    swap(data, min, right);
    return right; // That will become the pivot
}
```

13.2 – Merge Sort

- **Merge sort** orders a list of values by recursively dividing the list in half until each sub-list has one element, then recombining.
- More specifically:
  - divide the list into two roughly equal parts
  - recursively divide each part in half, continuing until a part contains only one element
  - merge the two parts into one sorted list
  - continue to merge parts as the recursion unfolds

13.2 – The decomposition of merge sort
13.2 – The merge portion of merge sort

13.2 – The mergeSort Algorithm

//-----------------------------------------------------------------
// Sorts the specified array of objects using the merge sort algorithm.
//-----------------------------------------------------------------
public static void mergeSort (Comparable[] data, int min, int max)
{
if (min < max)
{
int mid = (min + max) / 2;
mergeSort (data, min, mid);
mergeSort (data, mid+1, max);
merge (data, min, mid, max);
}
}

13.2 – The merge Algorithm

//-----------------------------------------------------------------
// Sorts the specified array of objects using the merge sort algorithm.
//-----------------------------------------------------------------
public static void merge (Comparable[] data, int first, int mid, int last)
{
Comparable[] temp = new Comparable[data.length];
int first1 = first, last1 = mid; // endpoints of first subarray
int first2 = mid+1, last2 = last; // endpoints of second subarray
int index = first1; // next index open in temp array
// Copy smaller item from each subarray into temp until one
// of the subarrays is exhausted
while (first1 <= last1 && first2 <= last2)
{
if (data[first1].compareTo(data[first2]) < 0)
{
temp[index] = data[first1];
first1++;
}
else
{
temp[index] = data[first2];
first2++;
}
index++;
}
// Copy remaining elements from first subarray, if any
while (first1 <= last1)
{
temp[index] = data[first1];
first1++;
index++;
}
// Copy remaining elements from second subarray, if any
while (first2 <= last2)
{
temp[index] = data[first2];
first2++;
index++;
}
// Copy merged data into original array
for (index = first; index <= last; index++)
data[index] = temp[index];

(more...)