2 Algorithm Analysis

2.1 Big-O Notation

1. (a) $O(n)$
   (b) $O(n^2)$
   (c) $O(1)$
   (d) $O(n)$
   (e) $O(n^3)$
   (f) $O(n^2)$
   (g) $O(n^{3/2})$

2. (a) $O(n)$
   (b) $O(1)$
   (c) $O(n^2)$
   (d) $O(n^2)$
   (e) $O(n)$

3. It is constant, not depending on n.

4. The access time is constant, not depending on the length of the array.

5. $O(n)$, since the main loop runs $n - 1$ times.

6.

Best case	$O(1)$	If the target is found in slot 0.
Worst case	$O(n)$	If every item in the array needs to be looked at, the loop runs n times.
Average case	$O(n)$	On average, about half of the array elements need to be checked, and $O(n/2) = O(n)$.

7. All cases are $O(n)$ because every array item needs to be included. There is also no early exit from the loop.

8. All cases are $O(n)$ because every array item needs to be looked at to be sure the largest is found. There is also no early exit from the loop.

9. All cases are $O(n)$ because every array item needs to be looked at to be sure the smallest is found. There is also no early exit from the loop.

2.2 Sorting: Insertion Sort

1. The sort continues with:

```
3, 7, 14, 22 | 45, 12, 19, 42, 6
3, 7, 14, 22, 45 | 12, 19, 42, 6
3, 7, 12, 14, 22, 45 | 19, 42, 6
3, 7, 12, 14, 19, 22, 45 | 42, 6
3, 7, 12, 14, 19, 22, 42, 45 | 6
3, 6, 7, 12, 14, 19, 22, 42, 45
```

2. If the data are reverse-sorted, every item has to move as far as possible to the left during each insertion.

3. (a)
```
31 | 7, 2, 34, 10, 5, 40, 22
7, 31 | 2, 34, 10, 5, 40, 22
2, 7, 31 | 34, 10, 5, 40, 22
2, 7, 31, 34 | 10, 5, 40, 22
2, 7, 10, 31, 34 | 5, 40, 22
2, 5, 7, 10, 31, 34 | 40, 22
2, 5, 7, 10, 31, 34, 40 | 22
2, 5, 7, 10, 22, 31, 34, 40
```

(b)
```
29 | 12, 48, 41, 19, 6, 25, 33
12, 29 | 48, 41, 19, 6, 25, 33
12, 29, 48 | 41, 19, 6, 25, 33
12, 29, 41, 48 | 19, 6, 25, 33
12, 19, 29, 41, 48 | 6, 25, 33
6, 12, 19, 29, 41, 48 | 25, 33
6, 12, 19, 25, 29, 41, 48 | 33
6, 12, 19, 25, 29, 33, 41, 48
```

(c)
```
21 | 31, 39, 22, 18, 38, 25, 6
21, 31 | 39, 22, 18, 38, 25, 6
21, 31, 39 | 22, 18, 38, 25, 6
21, 22, 31, 39 | 18, 38, 25, 6
18, 21, 22, 31, 39 | 38, 25, 6
18, 21, 22, 31, 38, 39 | 25, 6
18, 21, 22, 25, 31, 38, 39 | 6
6, 18, 21, 22, 25, 31, 38, 39
```

(d)
```
24 | 43, 42, 33, 37, 31, 40, 8
24, 43 | 42, 33, 37, 31, 40, 8
24, 42, 43 | 33, 37, 31, 40, 8
24, 33, 42, 43 | 37, 31, 40, 8
24, 33, 37, 42, 43 | 31, 40, 8
24, 31, 33, 37, 42, 43 | 40, 8
24, 31, 33, 37, 40, 42, 43 | 8
8, 24, 31, 33, 37, 40, 42, 43
```

4. (a) Because the first item can already be considered as sorted when the algorithm starts.
   (b) Because of the conditional AND. The test `j >= 0` is done first, and if it fails, the second part of the AND with the array access is not done. (Because `j` is decreasing, the only danger is with negative `j`.)
   (c) At most, `j` takes the values `i - 1` down to 0, which corresponds to `i` iterations.

5. Best case is if the data is already sorted, in which case insertion sort is $O(n)$. When the data is sorted, the while loop only runs once for each `i`.

6. On average, the while loop will run approximately `i/2` times for each `i`. The total work is therefore approximately half the work of the worst case:

$$\frac{1}{2}\left(1 + 2 + \cdots + (n-1)\right) = \frac{1}{2}\frac{n(n-1)}{2}$$

   which is still $O(n^2)$.

7. Sample code for `main()` using `data` already in `ArrayFunctions`:

```
System.out.println("Display:");
display(data);
insertionSort(data);
System.out.println("Display after sort:");
display(data);
```

8.
```
data = new int[100];
randomFill(data, 10000);
System.out.println("Display:");
display(data);
insertionSort(data);
System.out.println("Display after sort:");
display(data);
```

9.
```
public static boolean isSorted(int[] data) {
   for (int i = 1; i < data.length; i++) {
      if (data[i] < data[i-1]) return false;
   }
   return true;
}
```

10.
```
public static void randomFill(int[] data) {
   Random gen = new Random();
   for (int i = 0; i < data.length; i++) {
      data[i] = gen.nextInt();
   }
}
```

11.
```
for (int n = 100; n < 200000; n *= 2) {
   data = new int[n];
   randomFill(data);
   long start = System.currentTimeMillis();
   insertionSort(data);
   long elapsed = System.currentTimeMillis() - start;
   System.out.println(n + ": " + elapsed);
   if (!isSorted(data)) System.out.println("Not sorted for n = " + n);
}
```

12.
```
data = new int[100];
randomFill(data, 10000);
System.out.println("Sum: " + sum(data));
System.out.println("Max: " + max(data));
System.out.println("Min: " + min(data));
System.out.println("Display:");
display(data);
```

13. (a) a) 31, 7, 2, 34, 10, 5, 40, 22
2 | 7, 31, 34, 10, 5, 40, 22
2, 5 | 31, 34, 10, 7, 40, 22
2, 5, 7 | 34, 10, 31, 40, 22
2, 5, 7, 10 | 34, 31, 40, 22
2, 5, 7, 10, 22 | 31, 40, 34
2, 5, 7, 10, 22, 31 | 40, 34
2, 5, 7, 10, 22, 31, 34, 40

b) 29, 12, 48, 41, 19, 6, 25, 33
6 | 12, 48, 41, 19, 29, 25, 33
6, 12 | 48, 41, 19, 29, 25, 33
6, 12, 19 | 41, 48, 29, 25, 33
6, 12, 19, 25 | 48, 29, 41, 33
6, 12, 19, 25, 29 | 48, 41, 33
6, 12, 19, 25, 29, 33 | 41, 48
6, 12, 19, 25, 29, 33, 41, 48

c) 21, 31, 39, 22, 18, 38, 25, 6
6 | 31, 39, 22, 18, 38, 25, 21
6, 18 | 39, 22, 31, 38, 25, 21
6, 18, 21 | 22, 31, 38, 25, 39
6, 18, 21, 22 | 31, 38, 25, 39
6, 18, 21, 22, 25 | 38, 31, 39
6, 18, 21, 22, 25, 31 | 38, 39
6, 18, 21, 22, 25, 31, 38, 39

d) 24, 43, 42, 33, 37, 31, 40, 8
8 | 43, 42, 33, 37, 31, 40, 24
8, 24 | 42, 33, 37, 31, 40, 43
8, 24, 31 | 33, 37, 42, 40, 43
8, 24, 31, 33 | 37, 42, 40, 43
8, 24, 31, 33, 37 | 42, 40, 43
8, 24, 31, 33, 37, 40 | 42, 43
8, 24, 31, 33, 37, 40, 42, 43

(b)
```
public static void selectionSort(int[] data) {
   for (int i = 0; i < data.length - 1; i++) {
      int nextSmallestIndex = i;
      for (int j = i + 1; j < data.length; j++) {
         if (data[j] < data[nextSmallestIndex]) nextSmallestIndex = j;
      }
      if (i != nextSmallestIndex) {
         int temp = data[i];
         data[i] = data[nextSmallestIndex];
         data[nextSmallestIndex] = temp;
      }
   }
}
```

(c) $O(n^2)$, using a similar sum as for insertion sort.

(d) See solution for Exercise 12.

(e) Insertion sort is more sensitive, since its while loop may run once or it may run the maximum number of times. Selection sort's internal loop is a for-loop, so it always runs the same number of times. The only thing that changes is the number of swaps that are necessary.

2.3 Searching: Binary Search

1. (a)

left	right	mid	
0	7	3	
4	7	5	Returns 5

(b)

left	right	mid	
0	7	3	
0	2	1	
0	0	0	Returns 0

(c)

left	right	mid	
0	7	3	
4	7	5	
4	4	4	
4	3		Returns −1

(d)

left	right	mid	
0	7	3	
4	7	5	
6	7	6	
7	7	7	
8	7		Returns −1

(e)

left	right	mid	
0	7	3	
0	2	1	
0	0	0	
0	−1		Returns −1

(f)

left	right	mid	
0	7	3	
4	7	5	
6	7	6	Returns 6

2. (a)

left	right	mid	
0	9	4	
0	3	1	
2	3	2	
3	3	3	
3	2		Returns −1

(b)

left	right	mid	
0	9	4	
0	3	1	
2	3	2	Returns 2

(c)

left	right	mid	
0	9	4	
5	9	7	
8	9	8	Returns 8

(d)

left	right	mid	
0	9	4	
5	9	7	
8	9	8	
9	9	9	
9	8		Returns −1

(e)

left	right	mid	
0	9	4	
5	9	7	
8	9	8	
9	9	9	Returns 9

(f)

left	right	mid	
0	9	4	
0	3	1	
0	0	0	
0	−1		Returns −1

3. Best case for searching is one step, finding the element in the first spot checked. Generally, this is due to luck rather than the algorithm and not representative of the algorithm's performance.

4. Linear search is $O(n)$. Insertion sort in the worst case is $O(n^2)$ and binary search is $O(\log n)$, so the combination is $O(n^2 + \log n) = O(n^2)$. Therefore, linear search is a better choice.

5. ```
   data = {3, 6, 7, 12, 14, 19, 22, 42, 45};
   System.out.println("Binary search for data[2]: " + binarySearch(data, 7));
   System.out.println("Binary search for 1000: " + binarySearch(data, 1000));
   System.out.println("Binary search for data[8]: " + binarySearch(data, 45));
   ```

6. Example code for `main()` is below. If the item at slot 17 happens to have a duplicate entry, a different index may be returned.

   ```
   data = new int[100];
   randomFill(data, 10000);
   insertionSort(data);
   System.out.println("Binary search for data[17]: " + binarySearch(data, data[17]));
   ```
   ```

7. With those changes, the binary search can enter infinite loops. Infinite loops are possible both for items in the array and not in the array.