8.1.1

Indicate whether each of the following method calls is valid. Describe why it isn't valid or, if it is valid, describe what it does. Assume `people` is an array of `Person` objects and `peopleList` is a `List` of `Person` objects.

a. `people.sort();`
b. `Arrays.sort(people, 0, people.length - 3);`
c. `Arrays.sort(peopleList, 0, peopleList.length - 3);`
d. `Collections.sort(people);`
e. `Collections.sort(peopleList, new ComparePerson());`
f. `Collections.sort(peopleList, 0, peopleList.size() – 3);`

a Not valid, the array does not have a sort method
b. Valid.
c. Not valid, Arrays.sort only accepts arrays as its argument.
d. Not valid, Collections.sort does not accept arrays.
e. Valid
f. Not valid, Collections.sort does not accept a range.

8.2.1

Show the progress of each pass of the selection sort for the following array. How many passes are needed? How many comparisons are performed? How many exchanges? Show the array after each pass.
40 35 80 75 60 90 70 75 50 22

Pass	Array After Pass	Comparisons	Exchanges
0	40 35 80 75 60 90 70 75 50 22	0	0
1	22 35 80 75 60 90 70 75 50 40	9	1
2	22 35 80 75 60 90 70 75 50 40	8	1
3	22 35 40 75 60 90 70 75 50 80	7	1
4	22 35 40 50 60 90 70 75 75 80	6	1
5	22 35 40 50 60 90 70 75 75 80	5	1
6	22 35 40 50 60 70 90 75 75 80	4	1
7	22 35 40 50 60 70 75 90 75 80	3	1
8	22 35 40 50 60 70 75 75 90 80	2	1
9	22 35 40 50 60 70 75 75 80 90	1	1
TOTAL		45	9

8.3.1

How many passes of bubble sort are needed to sort the following array of integers? How many comparisons are performed? How many exchanges? Show the array after each pass.
40 35 80 75 60 90 70 75 50 22

Pass	Array After Pass	Comparisons	Exchanges
0	40 35 80 75 60 90 70 75 50 22	0	0

1	35 40 75 60 80 70 75 50 22 90	9	7
2	35 40 60 75 70 75 50 22 80 90	8	5
3	35 40 60 70 75 50 22 75 80 90	7	3
4	35 40 60 70 50 22 75 75 80 90	6	2
5	35 40 60 50 22 70 75 75 80 90	5	2
6	35 40 50 22 60 70 75 75 80 90	4	2
7	35 40 22 50 60 70 75 75 80 90	3	1
8	35 22 40 50 60 70 75 75 80 90	2	1
9	22 35 40 50 60 70 75 75 80 90	1	1
TOTAL		45	24

8.4.1

Sort the following array using insertion sort. How many passes are needed? How many comparisons are performed? How many exchanges? Show the array after each pass.
40 35 80 75 60 90 70 75 50 22

Pass	Table After Pass	Compares	Exchanges
0	40 35 80 75 60 90 70 75 50 22	0	0
1	35 40 80 75 60 90 70 75 50 22	1	1
2	35 40 80 75 60 90 70 75 50 22	1	0
3	35 40 75 80 60 90 70 75 50 22	2	1
4	35 40 60 75 80 90 70 75 50 22	3	2
5	35 40 60 75 80 90 70 75 50 22	1	0
6	35 40 60 70 75 80 90 75 50 22	4	3
7	35 40 60 70 75 75 80 90 50 22	3	2
8	35 40 50 60 70 75 75 80 90 22	7	6
9	22 35 40 50 60 70 75 75 80 90	9	9
TOTAL		31	24

8.5.1

Complete Table 8.3 for *n* = 1024 and *n* = 2048.

n	n^2	$n \log n$
8	64	24
16	256	64
32	1024	160
64	4096	384
128	16384	896
256	65536	2048
512	262144	4608
1024	1048576	10240
2048	4194304	22528

8.6.1

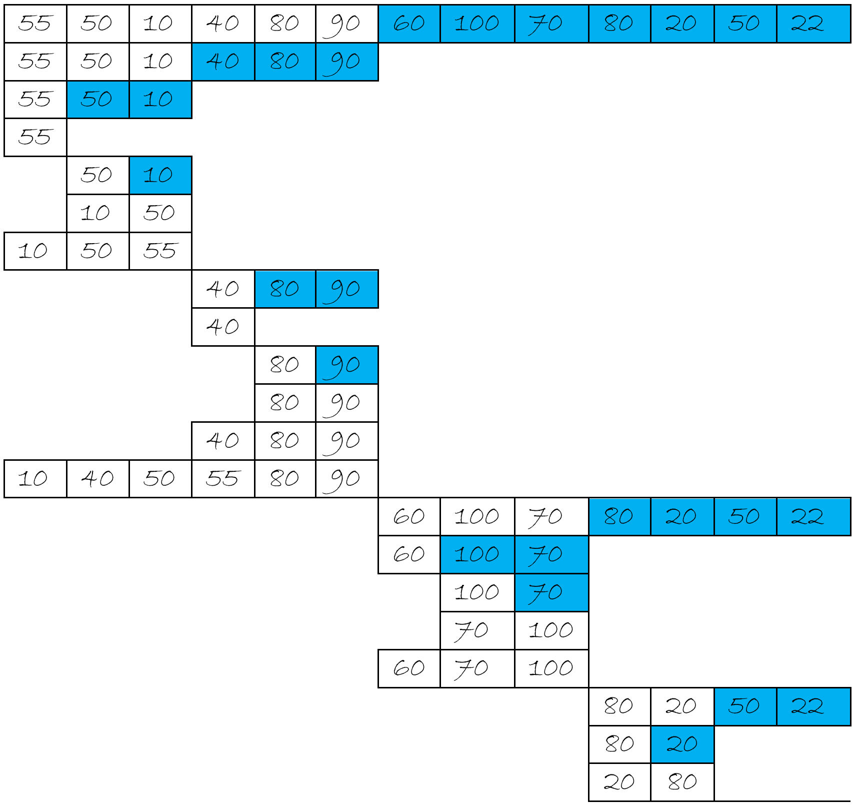
Trace the execution of Shell sort on the following array. Show the array after all sorts when the gap is 5, the gap is 2 and after the final sort when the gap is 1. List the number of comparisons and exchanges required when the gap is 5, the gap is 2 and when the gap is 1. Compare this with the number of comparisons and exchanges that would be required for a regular insertion sort.
40 35 80 75 60 90 70 65 50 22

Gap		Array After Sorting Gap	Compares	Exchanges
Original Array	0	40 35 80 75 60 90 70 75 50 22	0	0
5		40 35 75 50 22 90 70 80 75 60	5	3
2		22 35 40 50 70 60 75 80 75 90	13	6
1		22 35 40 50 60 70 75 75 80 90	11	2
TOTAL			29	11

Insertion sort took 31 compares and 24 exchanges to sort the same data.

8.7.1

Trace the execution of the merge sort on the following array, providing a figure similar to Figure 8.7.
55 50 10 40 80 90 60 100 70 80 20 50 22

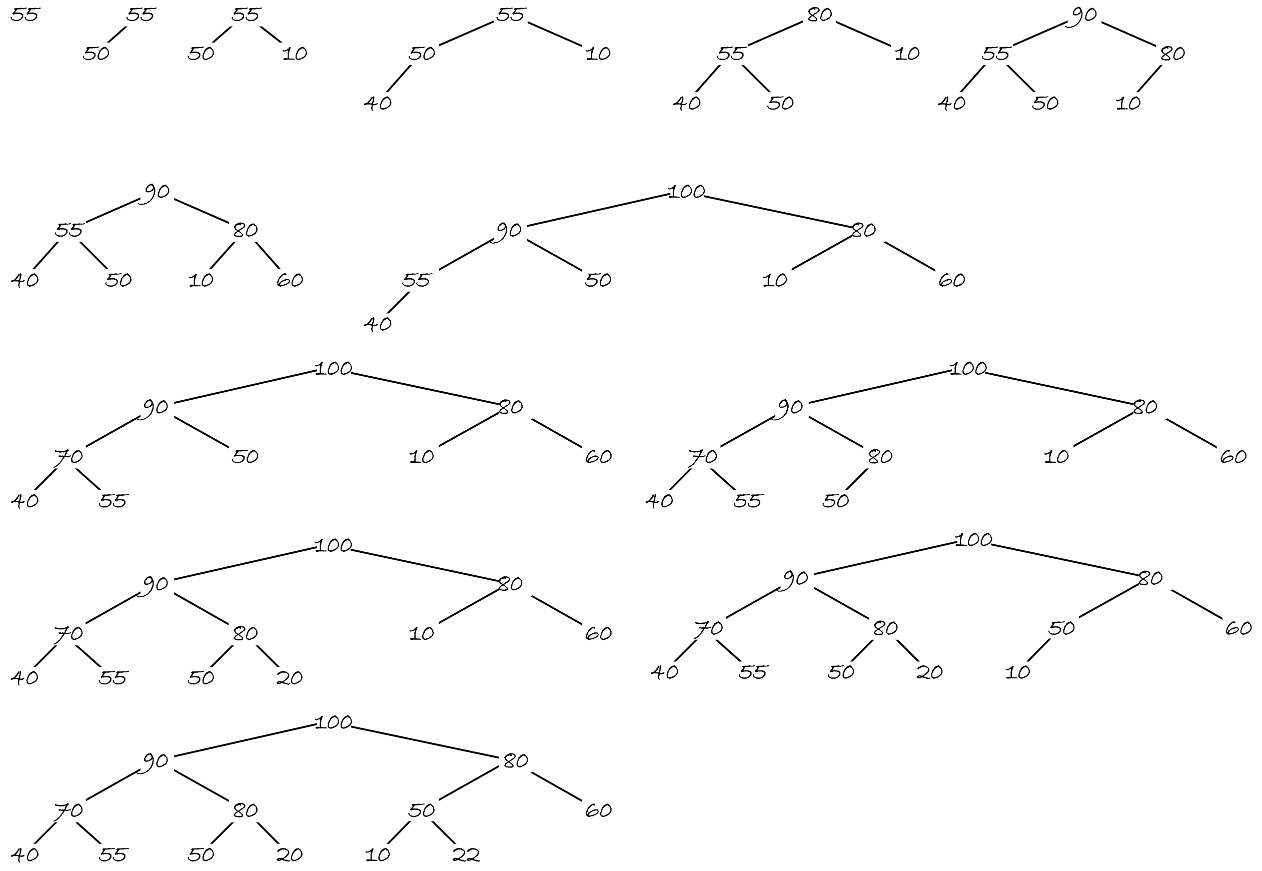


```
55  50  10  40  80  90  60  100 70  80  20  50  22
55  50  10  40  80  90
55  50  10
55
    50  10
    10  50
10  50  55
            40  80  90
            40
                80  90
                80  90
            40  80  90
10  40  50  55  80  90
                        60  100 70  80  20  50  22
                        60  100 70
                            100 70
                            70  100
                        60  70  100
                                    80  20  50  22
                                    80  20
                                    20  80
```

											55	22
											22	55
									20	22	55	80
						20	22	55	60	70	80	100
10	20	22	40	50	55	55	60	70	80	80	90	100

8.8.1

Build the heap from the numbers in the following list. How many exchanges were required? How many comparisons?
55 50 10 40 80 90 60 100 70 80 20 50 22



A total 22 comparisons and 10 exchanges were needed to build this heap from the numbers in the order presented.

8.9.1

Trace the execution of quicksort on the following array, assuming that the first item in each subarray is the pivot value. Show the values of `first` and `last` for each recursive call and the array elements after returning from each call. Also, show the value of `pivot` during each call and the value returned through `pivIndex`. How many times is `sort` called, and how many times is `partition` called?
55 50 10 40 80 90 60 100 70 80 20 50 22

```
sort([55, 50, 10, 40, 80, 90, 60, 100, 70, 80, 20, 50, 22], 0, 12)
pivIndex: 6
 sort([20, 50, 10, 40, 22, 50, 55, 100, 70, 80, 60, 90, 80], 0, 5)
 pivIndex: 1
  sort([10, 20, 50, 40, 22, 50, 55, 100, 70, 80, 60, 90, 80], 0, 0)
  table: [10, 20, 50, 40, 22, 50, 55, 100, 70, 80, 60, 90, 80]
  sort([10, 20, 50, 40, 22, 50, 55, 100, 70, 80, 60, 90, 80], 2, 5)
  pivIndex: 5
   sort([10, 20, 50, 40, 22, 50, 55, 100, 70, 80, 60, 90, 80], 2, 4)
   pivIndex: 4
    sort([10, 20, 22, 40, 50, 50, 55, 100, 70, 80, 60, 90, 80], 2, 3)
    pivIndex: 2
     sort([10, 20, 22, 40, 50, 50, 55, 100, 70, 80, 60, 90, 80], 2, 1)
     table: [10, 20, 22, 40, 50, 50, 55, 100, 70, 80, 60, 90, 80]
     sort([10, 20, 22, 40, 50, 50, 55, 100, 70, 80, 60, 90, 80], 3, 3)
     table: [10, 20, 22, 40, 50, 50, 55, 100, 70, 80, 60, 90, 80]
    table: [10, 20, 22, 40, 50, 50, 55, 100, 70, 80, 60, 90, 80]
    sort([10, 20, 22, 40, 50, 50, 55, 100, 70, 80, 60, 90, 80], 5, 4)
    table: [10, 20, 22, 40, 50, 50, 55, 100, 70, 80, 60, 90, 80]
   table: [10, 20, 22, 40, 50, 50, 55, 100, 70, 80, 60, 90, 80]
   sort([10, 20, 22, 40, 50, 50, 55, 100, 70, 80, 60, 90, 80], 6, 5)
   table: [10, 20, 22, 40, 50, 50, 55, 100, 70, 80, 60, 90, 80]
  table: [10, 20, 22, 40, 50, 50, 55, 100, 70, 80, 60, 90, 80]
 table: [10, 20, 22, 40, 50, 50, 55, 100, 70, 80, 60, 90, 80]
 sort([10, 20, 22, 40, 50, 50, 55, 100, 70, 80, 60, 90, 80], 7, 12)
 pivIndex: 12
  sort([10, 20, 22, 40, 50, 50, 55, 80, 70, 80, 60, 90, 100], 7, 11)
  pivIndex: 10
   sort([10, 20, 22, 40, 50, 50, 55, 60, 70, 80, 80, 90, 100], 7, 9)
   pivIndex: 7
    sort([10, 20, 22, 40, 50, 50, 55, 60, 70, 80, 80, 90, 100], 7, 6)
    table: [10, 20, 22, 40, 50, 50, 55, 60, 70, 80, 80, 90, 100]
    sort([10, 20, 22, 40, 50, 50, 55, 60, 70, 80, 80, 90, 100], 8, 9)
    pivIndex: 8
     sort([10, 20, 22, 40, 50, 50, 55, 60, 70, 80, 80, 90, 100], 8, 7)
     table: [10, 20, 22, 40, 50, 50, 55, 60, 70, 80, 80, 90, 100]
     sort([10, 20, 22, 40, 50, 50, 55, 60, 70, 80, 80, 90, 100], 9, 9)
     table: [10, 20, 22, 40, 50, 50, 55, 60, 70, 80, 80, 90, 100]
    table: [10, 20, 22, 40, 50, 50, 55, 60, 70, 80, 80, 90, 100]
   table: [10, 20, 22, 40, 50, 50, 55, 60, 70, 80, 80, 90, 100]
   sort([10, 20, 22, 40, 50, 50, 55, 60, 70, 80, 80, 90, 100], 11, 11)
   table: [10, 20, 22, 40, 50, 50, 55, 60, 70, 80, 80, 90, 100]
  table: [10, 20, 22, 40, 50, 50, 55, 60, 70, 80, 80, 90, 100]
  sort([10, 20, 22, 40, 50, 50, 55, 60, 70, 80, 80, 90, 100], 13, 12)
  table: [10, 20, 22, 40, 50, 50, 55, 60, 70, 80, 80, 90, 100]
 table: [10, 20, 22, 40, 50, 50, 55, 60, 70, 80, 80, 90, 100]
```

sort is called 19 times, and partition is called 8 times

8.9.3

Explain why the condition `(down > first)` is not necessary in the loop that decrements down.

The pivot value is at table[first] the condition pivot.compareTo(table[down]) will be false when down == first, thus a separate test for down > first is not necessary.

8.10.1

Explain why method `verify` will always determine whether an array is sorted. Does `verify` work if an array contains duplicate values?

The verify method tests to see if table[i] <= table[i+1] us true for all values of i from 0 to table.length-2. This is the definition of a sorted array. It will work for an array containing duplicate elements since the comparison criteria is less than or equal to.