Sorting and Searching 9

Critics search for ages for the wrong word, which,
to give them credit, they eventually find.
—Peter Ustinov (1952)

- → **Java implementation of sorting algorithms**
- → **Selection and insertion sorts**
- → **Merge sort**
- → **Sequential search and binary search**

For each of the following sorting algorithms, assume that an array of n elements, `a[0]`, `a[1]`, ..., `a[n-1]`, is to be sorted in ascending order.

SORTS: SELECTION AND INSERTION SORTS

Selection Sort

This is a "search-and-swap" algorithm. Here's how it works.

Find the smallest element in the array and exchange it with `a[0]`, the first element. Now find the smallest element in the subarray `a[1]` ...`a[n-1]` and swap it with `a[1]`, the second element in the array. Continue this process until just the last two elements remain to be sorted, `a[n-2]` and `a[n-1]`. The smaller of these two elements is placed in `a[n-2]`; the larger, in `a[n-1]`; and the sort is complete.

Trace these steps with a small array of four elements. The unshaded part is the subarray still to be searched.

8	1	4	6	
1	8	4	6	after first pass
1	4	8	6	after second pass
1	4	6	8	after third pass

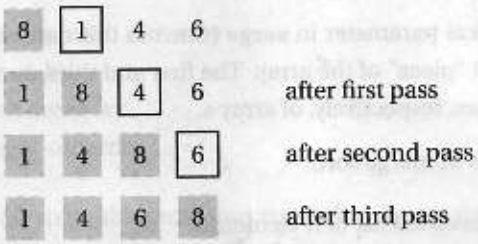
NOTE

1. For an array of n elements, the array is sorted after $n-1$ passes.
2. After the kth pass, the first k elements are in their final sorted position.

Insertion Sort

Think of the first element in the array, `a[0]`, as being sorted with respect to itself. The array can now be thought of as consisting of two parts, a sorted list followed by an unsorted list. The idea of insertion sort is to move elements from the unsorted list to the sorted list one at a time; as each item is moved, it is inserted into its correct position in the sorted list. In order to place the new item, some elements may need to be moved to the right to create a slot.

Here is the array of four elements. In each case, the boxed element is "it," the next element to be inserted into the sorted part of the list. The shaded area is the part of the list sorted so far.



NOTE

1. For an array of n elements, the array is sorted after $n - 1$ passes.
2. After the kth pass, `a[0]`, `a[1]`, ..., `a[k]` are sorted with respect to each other but not necessarily in their final sorted positions.
3. The worst case for insertion sort occurs if the array is initially sorted in reverse order, since this will lead to the maximum possible number of comparisons and moves.
4. The best case for insertion sort occurs if the array is already sorted in increasing order. In this case, each pass through the array will involve just one comparison, which will indicate that "it" is in its correct positionwith respect to the sorted list. Therefore, no elements will need to be moved,

Both insertion and selection sorts are inefficient for large n.

RECURSIVE SORTS: MERGE SORT AND QUICKSORT

Selection and insertion sorts are inefficient for large n, requiring approximately n passes through a list of n elements. More efficient algorithms can be devised using a "divide-and-conquer" approach, which is used in both the sorting algorithms that follow. Quicksort is not in the AP Java subset.

Merge Sort

Here is a recursive description of how merge sort works:

If there is more than one element in the array
    Break the array into two halves.
    Merge sort the left half.
    Merge sort the right half.
    Merge the two subarrays into a sorted array.

The main disadvantage of merge sort is that it uses a temporary array.

Merge sort uses a `merge` method to merge two sorted pieces of an array into a single sorted array. For example, suppose array `a[0] ... a[n-1]` is such that `a[0] ... a[k]` is sorted and `a[k+1] ... a[n-1]` is sorted, both parts in increasing order. Example:

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
2	5	8	9	1	6

In this case, `a[0] ...a[3]` and `a[4] ...a[5]` are the two sorted pieces. The method call `merge(a,0,3,5)` should produce the "merged" array:

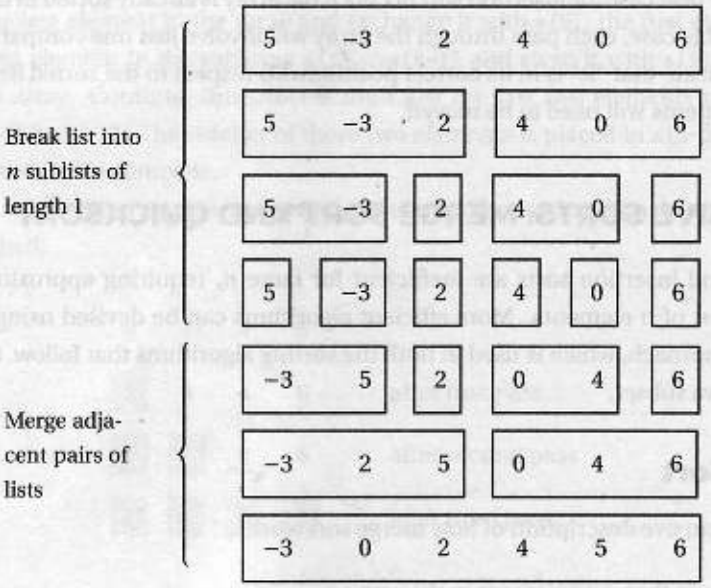
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
1	2	5	6	8	9

The middle numerical parameter in `merge` (the 3 in this case) represents the index of the last element in the first "piece" of the array. The first and third numerical parameters are the lowest and highest index, respectively, of array `a`.

Here's what happens in merge sort:

1. Start with an unsorted list of n elements.
2. The recursive calls break the list into n sublists, each of length 1. Note that these n arrays, each containing just one element, are sorted!
3. Recursively merge adjacent pairs of lists. There are then approximately $n/2$ lists of length 2; then, approximately $n/4$ lists of approximate length 4, and so on, until there is just one list of length n.

An example of merge sort follows:



Analysis of merge sort:

1. The major disadvantage of merge sort is that it needs a temporary array that is as large as the original array to be sorted. This could be a problem if space is a factor.
2. Merge sort is not affected by the initial ordering of the elements. Thus, best, worst, and average cases have similar run times.

Quicksort

Optional topic

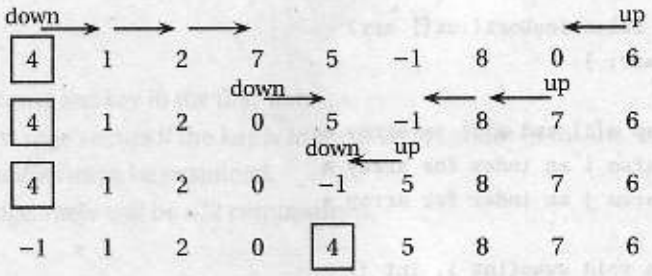
For large n, quicksort is, on average, the fastest known sorting algorithm. Here is a recursive description of how quicksort works:

If there are at least two elements in the array
    Partition the array.
    Quicksort the left subarray.
    Quicksort the right subarray.

The `partition` method splits the array into two subarrays as follows: a *pivot* element is chosen at random from the array (often just the first element) and placed so that all items to the left of the pivot are less than or equal to the pivot, whereas those to the right are greater than or equal to it.

For example, if the array is 4, 1, 2, 7, 5, −1, 8, 0, 6, and `a[0]` = 4 is the pivot, the `partition` method produces

Here's how the partitioning works: Let `a[0]`, 4 in this case, be the pivot. Markers `up` and `down` are initialized to index values 0 and $n-1$, as shown. Move the `up` marker until a value less than the pivot is found, or `down` equals `up`. Move the `down` marker until a value greater than the pivot is found, or `down` equals `up`. Swap `a[up]` and `a[down]`. Continue the process until `down` equals `up`. This is the pivot position. Swap `a[0]` and `a[pivotPosition]`.



Notice that the pivot element, 4, is in its final sorted position.

Analysis of quicksort:

1. For the fastest run time, the array should be partitioned into two parts of roughly the same size.

(continued)

The main disadvantage of quicksort is that its worst case behavior is very inefficient.

2. If the pivot happens to be the smallest or largest element in the array, the split is not much of a split—one of the subarrays is empty! If this happens repeatedly, quicksort degenerates into a slow, recursive version of selection sort and is very inefficient.
3. The worst case for quicksort occurs when the partitioning algorithm repeatedly divides the array into pieces of size 1 and $n-1$. An example is when the array is initially sorted in either order and the first or last element is chosen as the pivot. Some algorithms avoid this situation by initially shuffling up the given array (!) or selecting the pivot by examining several elements of the array (such as first, middle, and last) and then taking the median.

NOTE

For both quicksort and merge sort, when a subarray gets down to some small size m, it becomes faster to sort by straight insertion. The optimal value of m is machine-dependent, but it's approximately equal to 7.

SORTING ALGORITHMS IN JAVA

Unlike the container classes like `ArrayList`, whose elements must be objects, arrays can hold either objects or primitive types like `int` or `double`.

A common way of organizing code for sorting arrays is to create a sorter class with an array private instance variable. The class holds all the methods for a given type of sorting algorithm, and the constructor assigns the user's array to the private array variable.

➥ Example

Selection sort for an array of `int`.

```
/* A class that sorts an array of ints from
 * largest to smallest using selection sort. */

public class SelectionSort
{
    private int[] a;

    public SelectionSort(int[] arr)
    { a = arr; }

    /** Swap a[i] and a[j] in array a.
     *  @param i an index for array a
     *  @param j an index for array a
     */
    private void swap(int i, int j)
    {
        int temp = a[i];
        a[i] = a[j];
        a[j] = temp;
    }
```

```
    /** Sort array a from largest to smallest using selection sort.
     *  Precondition:  a is an array of ints.
     */
    public void selectionSort()
    {
        int maxPos, max;

        for (int i = 0; i < a.length - 1; i++)
        {
            //find max element in a[i+1] to a[a.length-1]
            max = a[i];
            maxPos = i;
            for (int j = i + 1; j < a.length; j++)
                if (max < a[j])
                {
                    max = a[j];
                    maxPos = j;
                }
            swap(i, maxPos); //swap a[i] and a[maxPos]
        }
    }
}
```

Note that in order to sort *objects*, there must be a `compareTo` method in the class, since you need to be able to compare elements.

SEQUENTIAL SEARCH

Assume that you are searching for a key in a list of n elements. A sequential search starts at the first element and compares the key to each element in turn until the key is found or there are no more elements to examine in the list. If the list is sorted, in ascending order, say, stop searching as soon as the key is less than the current list element. (If the key is less than the current element, it will be less than all subsequent elements.)

Analysis:

1. The best case has key in the first slot.
2. The worst case occurs if the key is in the last slot or not in the list. In the worst case, all n elements must be examined.
3. On average, there will be $n/2$ comparisons.

BINARY SEARCH

Binary search works only if the array is sorted on the search key.

If the elements are in a *sorted* array, a divide-and-conquer approach provides a much more efficient searching algorithm. The following recursive pseudo-code algorithm shows how the *binary search* works.

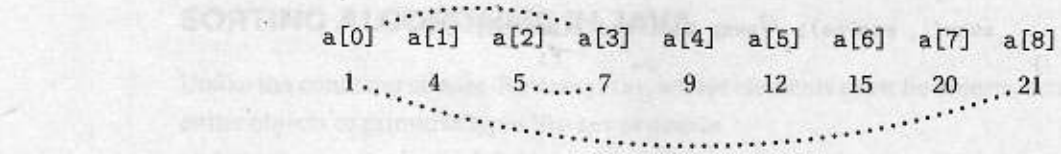
Assume that `a[low] ... a[high]` is sorted in ascending order and that a method `binSearch` returns the index of `key`. If `key` is not in the array, it returns -1.

```
if (low > high)   //Base case. No elements left in array.
    return -1;
else
{
    mid = (low + high)/2;
    if (key is equal to a[mid])   //found the key
        return mid;
    else if (key is less than a[mid])  //key in left half of array
        < binSearch for key in a[low] to a[mid-1] >
    else    //key in right half of array
        < binSearch for key in a[mid+1] to a[high] >
}
```

NOTE

When `low` and `high` cross, there are no more elements to examine, and `key` is not in the array.

Example: suppose 5 is the key to be found in the following array:



First pass: `low` is 0, `high` is 8. `mid = (0+8)/2 = 4.` Check `a[4]`.
Second pass: `low` is 0, `high` is 3. `mid = (0+3)/2 = 1.` Check `a[1]`.
Third pass: `low` is 2, `high` is 3. `mid = (2+3)/2 = 2.` Check `a[2]`. Yes! Key is found.

Analysis of Binary Search

The number of comparisons for binary search in the worst case depends on whether n is a power of 2 or not.

1. In the best case, the key is found on the first try (i.e., `(low + high)/2` is the index of `key`).
2. In the worst case, the key is not in the list or is at either end of a sublist. Here the n elements must be divided by 2 until there is just one element, and then that last element must be tested. An easy way to find the number of comparisons in the worst case is to round n up to the next power of 2 and take the exponent. For example, in the array above, $n = 9$. Suppose 21 were the key. Round 9 up to 16, which equals 2^4. Thus you would need four comparisons to find it.
3. There's an interesting wrinkle when discussing the worst case of a binary search that uses the above algorithm. If n is an exact power of 2, the number of comparisons in the worst case equals the exponent plus one. For example, if $n = 32 = 2^5$, then the number of comparisons to find a key at one of the endpoints is $5 + 1 = 6$. Try it! Also, the number of comparisons to find a key that isn't in the list either will be 5 (the exponent), if the key value lies between `a[0]` and `a[n-1]`, or 6 (the exponent plus one), if the key value is less than `a[0]` or greater than `a[n-1]`.

   As a simple example, consider the array 3, 7, 9, 11, where `a[0]` is 3 and `a[3]` is 11. Notice that the number of elements, n, equals 4, which is 2^2.

   - If the key is 7, it will be found on the first comparison, since the `mid` index equals `(0+3)/2`, which is 1, and `a[1]` equals `a[mid]`. This represents the best case.
   - If the key is 3 or 11 (an endpoint of the array), the algorithm will need 3 comparisons to find the key, a worst case.

- If the key is 1 or 20, say (outside the range of values and not in the array), the algorithm will need 3 comparisons to find the key, a worst case.
- If the key is 8, say (not in the array, but inside the range of values), the algorithm will need just 2 comparisons to determine that the key is not in the array. This is therefore *not* a worst case situation.

Here is a general rule for calculating the maximum number of comparisons in different binary search situations:

If n, the number of elements, is not a power of 2, round n up to the nearest power of 2. The number of comparisons in the worst case is equal to the exponent. It represents the cases in which the key is at either of the endpoints of the array, or not in the array.

If n, the number of elements, is a power of 2, express n as a power of 2.

- Case 1: The key is at either of the endpoints of the array. These are worst cases in which the number of comparisons is equal to the exponent plus one.
- Case 2: The key is not in the array, and is also less than `a[0]` or greater than `a[n-1]`. These are worst cases in which the number of comparisons is equal to the exponent plus one.
- Case 3: The key is not in the array, and its value lies between `a[0]` and `a[n-1]`. Here the number of comparisons to determine that the key is not in the array is equal to the exponent. There will be one fewer comparison than in the worst case.

Chapter Summary

You should not memorize any sorting code. You must, however, be familiar with the mechanism used in each of the sorting algorithms. For example, you should be able to explain how the merge method of merge sort works, or how many elements are in their final sorted position after a certain number of passes through the selection sort loop. You should know the best and worst case situations for each of the sorting algorithms.

Be familiar with the sequential and binary search algorithms. You should know that a binary search is more efficient than a sequential search, and that a binary search can only be used for an array that is sorted on the search key.

MULTIPLE-CHOICE QUESTIONS ON SORTING AND SEARCHING

1. The decision to choose a particular sorting algorithm should be made based on

   I Run-time efficiency of the sort
   II Size of the array
   III Space efficiency of the algorithm

   (A) I only
   (B) II only
   (C) III only
   (D) I and II only
   (E) I, II, and III

2. The following code fragment does a sequential search to determine whether a given integer, `value`, is stored in an array `a[0] ... a[n-1]`.

```
int i = 0;
while (/* boolean expression */)
{
    i++;
}
if (i == n)
    return -1;     //value not found
else
    return i;      // value found at location i
```

   Which of the following should replace `/* boolean expression */` so that the algorithm works as intended?

   (A) `value != a[i]`
   (B) `i < n && value == a[i]`
   (C) `value != a[i] && i < n`
   (D) `i < n && value != a[i]`
   (E) `i < n || value != a[i]`

3. A feature of data that is used for a binary search but not necessarily used for a sequential search is

   (A) length of list.
   (B) type of data.
   (C) order of data.
   (D) smallest value in the list.
   (E) median value of the data.

4. Array `unsortedArr` contains an unsorted list of integers. Array `sortedArr` contains a list of integers sorted in increasing order. Which of the following operations is more efficient for `sortedArr` than `unsortedArr`? Assume the most efficient algorithms are used.

   I Inserting a new element
   II Searching for a given element
   III Computing the mean of the elements

   (A) I only
   (B) II only
   (C) III only
   (D) I and II only
   (E) I, II, and III

5. An algorithm for searching a large sorted array for a specific value x compares every third item in the array to x until it finds one that is greater than or equal to x. When a larger value is found, the algorithm compares x to the previous two items. If the array is sorted in increasing order, which of the following describes all cases when this algorithm uses fewer comparisons to find x than would a binary search?
   (A) It will never use fewer comparisons.
   (B) When x is in the middle position of the array
   (C) When x is very close to the beginning of the array
   (D) When x is very close to the end of the array
   (E) When x is not in the array

6. Assume that `a[0]` ... `a[N-1]` is an array of N positive integers and that the following assertion is true:

   `a[0]` > `a[k]` for all k such that $0 < k < N$

   Which of the following *must* be true?
   (A) The array is sorted in ascending order.
   (B) The array is sorted in descending order.
   (C) All values in the array are different.
   (D) `a[0]` holds the smallest value in the array.
   (E) `a[0]` holds the largest value in the array.

7. The following code is designed to set `index` to the location of the first occurrence of `key` in array `a` and to set `index` to −1 if `key` is not in `a`.

```
index = 0;
while (a[index] != key)
    index++;
if (a[index] != key)
    index = -1;
```

   In which case will this program *definitely* fail to perform the task described?
   (A) When `key` is the first element of the array
   (B) When `key` is the last element of the array
   (C) When `key` is not in the array
   (D) When `key` equals 0
   (E) When `key` equals `a[key]`

8. Consider the following class.

```
/** A class that sorts an array of Integer objects from
 *  largest to smallest using a selection sort.
 */
public class Sorter
{
    private Integer[] a;

    public Sorter(Integer[] arr)
    { a = arr; }

    /** Swap a[i] and a[j] in array a. */
    private void swap(int i, int j)
    { /* implementation not shown */ }

    /** Sort array a from largest to smallest using selection sort.
     *  Precondition: a is an array of Integer objects.
     */
    public void selectionSort()
    {
        for (int i = 0; i < a.length - 1; i++)
        {
            //find max element in a[i+1] to a[n-1]
            Integer max = a[i];
            int maxPos = i;
            for (int j = i + 1; j < a.length; j++)
                if (max.compareTo(a[j]) < 0) //max less than a[j]
                {
                    max = a[j];
                    maxPos = j;
                }
            swap(i, maxPos);    //swap a[i] and a[maxPos]
        }
    }
}
```

If an array of `Integer` contains the following elements, what would the array look like after the third pass of `selectionSort`, sorting from high to low?

89 42 −3 13 109 70 2

(A) 109 89 70 13 42 −3 2
(B) 109 89 70 42 13 2 −3
(C) 109 89 70 −3 2 13 42
(D) 89 42 13 −3 109 70 2
(E) 109 89 42 −3 13 70 2

9. Refer to method search.

```
/** @param v an initialized array of integers
 *  @param key the value to be found
 *  Postcondition:
 *   - Returned value k is such that -1 <= k <= v.length-1.
 *   - If k >= 0 then v[k] == key.
 *   - If k == -1, then key != any of the elements in v.
 */
public static int search(int[] v, int key)
{
    int index = 0;
    while (index < v.length && v[index] < key)
        index++;
    if (v[index] == key)
        return index;
    else
        return -1;
}
```

Assuming that the method works as intended, which of the following should be added to the precondition of search?

(A) v is sorted smallest to largest.
(B) v is sorted largest to smallest.
(C) v is unsorted.
(D) There is at least one occurrence of key in v.
(E) key occurs no more than once in v.

Questions 10–14 are based on the `binSearch` method and the private instance variable `a` for some class:

```
private int[] a;

/** Does binary search for key in array a[0]...a[a.length-1],
 *  sorted in ascending order.
 *  @param key the integer value to be found
 *  Postcondition:
 *   - index has been returned such that a[index]==key.
 *   - If key not in a, return -1.
 */
public int binSearch(int key)
{
    int low = 0;
    int high = a.length - 1;
    while (low <= high)
    {
        int mid = (low + high) / 2;
        if (a[mid] == key)
            return mid;
        else if (a[mid] < key)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1;
}
```

A binary search will be performed on the following list.

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
4	7	9	11	20	24	30	41

10. To find the key value 27, the search interval *after* the first pass through the `while` loop will be
    (A) a[0] ... a[7]
    (B) a[5] ... a[6]
    (C) a[4] ... a[7]
    (D) a[2] ... a[6]
    (E) a[6] ... a[7]

11. How many iterations will be required to determine that 27 is not in the list?
    (A) 1
    (B) 3
    (C) 4
    (D) 8
    (E) 16

12. What will be stored in y after executing the following?

```
int y = binSearch(4);
```

(A) 20
(B) 7
(C) 4
(D) 0
(E) -1

13. If the test for the `while` loop is changed to

```
while (low < high)
```

the `binSearch` method does not work as intended. Which value in the given list will not be found?

(A) 4
(B) 7
(C) 11
(D) 24
(E) 30

14. For `binSearch`, which of the following assertions will be true following every iteration of the `while` loop?

(A) `key = a[mid]` or `key` is not in `a`.
(B) `a[low] ≤ key ≤ a[high]`
(C) `low ≤ mid ≤ high`
(D) `key = a[mid]`, or `a[low] ≤ key ≤ a[high]`
(E) `key = a[mid]`, or `a[low] ≤ key ≤ a[high]`, or `key` is not in array `a`.

15. A large sorted array containing about 30,000 elements is to be searched for a value `key` using an iterative binary search algorithm. Assuming that `key` is in the array, which of the following is closest to the smallest number of iterations that will guarantee that `key` is found? Note: $10^3 \approx 2^{10}$.

(A) 15
(B) 30
(C) 100
(D) 300
(E) 3000

For Questions 16–19 refer to the `insertionSort` method and the private instance variable `a`, both in a `Sorter` class.

```
private Integer[] a;

/** Precondition:  a[0],a[1]...a[a.length-1] is an unsorted array
 *                 of Integer objects.
 *  Postcondition: Array a is sorted in descending order.
 */
public void insertionSort()
{
    for (int i = 1; i < a.length; i++)
    {
        Integer temp = a[i];
        int j = i - 1;
        while (j >= 0 && temp.compareTo(a[j]) > 0)
        {
            a[j+1] = a[j];
            j--;
        }
        a[j+1] = temp;
    }
}
```

16. An array of `Integer` is to be sorted biggest to smallest using the `insertionSort` method. If the array originally contains

    1 7 9 5 4 12

    what will it look like after the third pass of the `for` loop?

    (A) 9 7 1 5 4 12
    (B) 9 7 5 1 4 12
    (C) 12 9 7 1 5 4
    (D) 12 9 7 5 4 1
    (E) 9 7 12 5 4 1

17. When sorted biggest to smallest with `insertionSort`, which list will need the fewest changes of position for individual elements?

    (A) 5, 1, 2, 3, 4, 9
    (B) 9, 5, 1, 4, 3, 2
    (C) 9, 4, 2, 5, 1, 3
    (D) 9, 3, 5, 1, 4, 2
    (E) 3, 2, 1, 9, 5, 4

18. When sorted biggest to smallest with `insertionSort`, which list will need the greatest number of changes in position?

    (A) 5, 1, 2, 3, 4, 7, 6, 9
    (B) 9, 5, 1, 4, 3, 2, 1, 0
    (C) 9, 4, 6, 2, 1, 5, 1, 3
    (D) 9, 6, 9, 5, 6, 7, 2, 0
    (E) 3, 2, 1, 0, 9, 6, 5, 4

19. While typing the `insertionSort` method, a programmer by mistake enters

```
while (temp.compareTo( a[j]) > 0)
```

instead of

```
while (j >= 0 && temp.compareTo( a[j]) > 0)
```

Despite this mistake, the method works as intended the first time the programmer enters an array to be sorted in descending order. Which of the following could explain this?

I The first element in the array was the largest element in the array.
II The array was already sorted in descending order.
III The first element was less than or equal to all the other elements in the array.

(A) I only
(B) II only
(C) III only
(D) I and II only
(E) II and III only

20. The elements in a long list of integers are roughly sorted in decreasing order. No more than 5 percent of the elements are out of order. Which of the following is a valid reason for using an insertion sort rather than a selection sort to sort this list into decreasing order?

I There will be fewer comparisons of elements for insertion sort.
II There will be fewer changes of position of elements for insertion sort.
III There will be less space required for insertion sort.

(A) I only
(B) II only
(C) III only
(D) I and II only
(E) I, II, and III

Optional topic

21. The code shown sorts array `a[0] ... a[a.length-1]` in descending order.

```
public static void sort(String[] a)
{
    for (int i = 0; i < a.length - 1; i++)
        for (int j = 0; j < a.length - i - 1; j++)
            if (a[j].compareTo(a[j+1]) < 0)
                swap(a, j, j + 1);  //swap a[j] and a[j+1]
}
```

This is an example of

(A) selection sort.
(B) insertion sort.
(C) merge sort.
(D) quicksort.
(E) none of the above.

22. Which of the following is a valid reason why merge sort is a better sorting algorithm than insertion sort for sorting long, randomly ordered lists?

I Merge sort requires less code than insertion sort.
II Merge sort requires less storage space than insertion sort.
III Merge sort runs faster than insertion sort.

(A) I only
(B) II only
(C) III only
(D) I and II only
(E) II and III only

23. A large array of lowercase characters is to be searched for the pattern "pqrs." The first step in a very efficient searching algorithm is to look at characters with index

(A) 0, 1, 2, ... until a "p" is encountered.
(B) 0, 1, 2, ... until any letter in "p" ... "s" is encountered.
(C) 3, 7, 11, ... until an "s" is encountered.
(D) 3, 7, 11, ... until any letter in "p" ... "s" is encountered.
(E) 3, 7, 11, ... until any letter other than "p" ... "s" is encountered.

24. The array `names[0]`, `names[1]`, ..., `names[9999]` is a list of 10,000 name strings. The list is to be searched to determine the location of some name `X` in the list. Which of the following preconditions is necessary for a binary search?

(A) There are no duplicate names in the list.
(B) The number of names N in the list is large.
(C) The list is in alphabetical order.
(D) Name `X` is definitely in the list.
(E) Name `X` occurs near the middle of the list.

25. Consider the following method:

```
/** Precondition: a[0],a[1]...a[n-1] contain integers. */
public static int someMethod(int[] a, int n, int value)
{
    if (n == 0)
        return -1;
    else
    {
        if (a[n-1] == value)
            return n - 1;
        else
            return someMethod(a, n - 1, value);
    }
}
```

The method shown is an example of

(A) insertion sort.
(B) merge sort.
(C) selection sort.
(D) binary search.
(E) sequential search.

Optional topic

26. The `partition` method for quicksort partitions a list as follows:

(i) A pivot element is selected from the array.
(ii) The elements of the list are rearranged such that all elements to the left of the pivot are less than or equal to it; all elements to the right of the pivot are greater than or equal to it.

Partitioning the array requires which of the following?

(A) A recursive algorithm
(B) A temporary array
(C) An external file for the array
(D) A swap algorithm for interchanging array elements.
(E) A merge method for merging two sorted lists

27. Assume that merge sort will be used to sort an array `arr` of `n` integers into increasing order. What is the purpose of the `merge` method in the merge sort algorithm?

(A) Partition `arr` into two parts of roughly equal length, then merge these parts.
(B) Use a recursive algorithm to sort `arr` into increasing order.
(C) Divide `arr` into `n` subarrays, each with one element.
(D) Merge two sorted parts of `arr` into a single sorted array.
(E) Merge two sorted arrays into a temporary array that is sorted.

28. A binary search is to be performed on an array with 600 elements. In the *worst* case, which of the following best approximates the number of iterations of the algorithm?

(A) 6
(B) 10
(C) 100
(D) 300
(E) 600

29. A worst case situation for insertion sort would be

I A list in correct sorted order.
II A list sorted in reverse order.
III A list in random order.

(A) I only
(B) II only
(C) III only
(D) I and II only
(E) II and III only

30. Consider a binary search algorithm to search an ordered list of numbers. Which of the following choices is closest to the maximum number of times that such an algorithm will execute its main comparison loop when searching a list of 1 million numbers?

(A) 6
(B) 20
(C) 100
(D) 120
(E) 1000

31. Consider these three tasks:

I A sequential search of an array of n names

II A binary search of an array of n names in alphabetical order

III An insertion sort into alphabetical order of an array of n names that are initially in random order

For large n, which of the following lists these tasks in order (from least to greatest) of their average case run times?

(A) II I III
(B) I II III
(C) II III I
(D) III I II
(E) III II I

Optional topic

Questions 32 and 33 are based on the `Sort` interface and `MergeSort` and `QuickSort` classes shown below.

```
public interface Sort
{
    void sort();
}

public class MergeSort implements Sort
{
    private String[] a;

    public MergeSort(String[] arr)
    { a = arr; }

    /** Merge a[lb] to a[mi] and a[mi+1] to a[ub].
     *  Precondition: a[lb] to a[mi] and a[mi+1] to a[ub] both sorted
     *                in increasing order.
     */
    private void merge(int lb, int mi, int ub)
    { /* Implementation not shown. */ }

    /** Sort a[first]..a[last] in increasing order using merge sort.
     *  Precondition: a is an array of String objects.
     */
    private void sort(int first, int last)
    {
        int mid;

        if (first != last)
        {
            mid = (first + last) / 2;
            sort(first, mid);
            sort(mid + 1, last);
            merge(first, mid, last);
        }
    }

    /** Sort array a from smallest to largest using merge sort.
     *  Precondition: a is an array of String objects.
     */
    public void sort()
    {
        sort(0, a.length - 1);
    }
}
```

(continued)

```
public class QuickSort implements Sort
{
    private String[] a;

    public QuickSort(String[] arr)
    { a = arr; }

    /** Swap a[i] and a[j] in array a. */
    private void swap(int i, int j)
    { /* Implementation not shown. */ }

    /** @return the index pivPos such that a[first] to a[last]
     *  is partitioned: a[first..pivPos] <= a[pivPos] and
     *  a[pivPos..last] >= a[pivPos]
     */
    private int partition(int first, int last)
    { /* Implementation not shown. */ }

    /** Sort a[first]..a[last] in increasing order using quicksort.
     *  Precondition: a is an array of String objects.
     */
    private void sort(int first, int last)
    {
        if (first < last)
        {
            int pivPos = partition(first, last);
            sort(first, pivPos - 1);
            sort(pivPos + 1, last);
        }
    }

    /** Sort array a in increasing order. */
    public void sort()
    {
        sort(0, a.length - 1);
    }
}
```

(continued)

32. Notice that the `MergeSort` and `QuickSort` classes both have a private helper method that implements the recursive sort routine. For this example, which of the following is a valid reason for having a helper method?

I The helper method hides the implementation details of the sorting algorithm from the user.

II A method with additional parameters is needed to implement the recursion.

III Providing a helper method increases the run-time efficiency of the sorting algorithm.

(A) I only
(B) II only
(C) III only
(D) I and II only
(E) I, II, and III

33. A piece of code to test the `QuickSort` and `MergeSort` classes is as follows:

(continued)

```
//Create an array of String values
String[] strArray = makeArray(strList);
writeList(strArray);
/* more code */
```

where `makeArray` creates an array of `String` from a list `strList`. Which of the following replacements for `/* more code */` is reasonable code to test `QuickSort` and `MergeSort`? You can assume `writeList` correctly writes out an array of `String`.

(A)
```
Sort q = new QuickSort(strArray);
Sort m = new MergeSort(strArray);
q.sort();
writeList(strArray);
m.sort();
writeList(strArray);
```

(B)
```
QuickSort q = new Sort(strArray);
MergeSort m = new Sort(strArray);
q.sort();
writeList(strArray);
m.sort();
writeList(strArray);
```

(C)
```
Sort q = new QuickSort(strArray);
Sort m = new MergeSort(strArray);
String[] copyArray = makeArray(strList);
q.sort(0, strArray.length - 1);
writeList(strArray);
m.sort(0, copyArray.length - 1);
writeList(copyArray);
```

(D)
```
QuickSort q = new Sort(strArray);
String[] copyArray = makeArray(strList);
MergeSort m = new Sort(strArray);
q.sort();
writeList(strArray);
m.sort();
writeList(copyArray);
```

(E)
```
Sort q = new QuickSort(strArray);
String[] copyArray = makeArray(strList);
Sort m = new MergeSort(copyArray);
q.sort();
writeList(strArray);
m.sort();
writeList(copyArray);
```

Questions 34–36 refer to the Hi-Lo game described below.

Consider the problem of writing a Hi-Lo game in which a user thinks of an integer from 1 to 100 inclusive and the computer tries to guess that number. Each time the computer makes a guess, the user makes one of three responses:

- "lower" (i.e., the number is lower than the computer's guess)
- "higher" (i.e., the number is higher than the computer's guess)
- "you got it in < ***however many*** > tries!"

34. Suppose the game is programmed so that the computer uses a binary search strategy for making its guesses. What is the maximum number of guesses the computer could make before guessing the user's number?
    (A) 50
    (B) 25
    (C) 10
    (D) 7
    (E) 6

35. Suppose the computer used a *sequential search* strategy for guessing the user's number. What is the maximum number of guesses the computer could make before guessing the user's number?
    (A) 100
    (B) 99
    (C) 50
    (D) 25
    (E) 10

36. Using a sequential search strategy, how many guesses *on average* would the computer need to guess the number?
    (A) 100
    (B) Between 51 and 99
    (C) 50
    (D) 25
    (E) Fewer than 25

ANSWER KEY

1. **E**
2. **D**
3. **C**
4. **B**
5. **C**
6. **E**
7. **C**
8. **A**
9. **A**
10. **C**
11. **B**
12. **D**
13. **A**
14. **E**
15. **A**
16. **B**
17. **B**
18. **A**
19. **D**
20. **A**
21. **E**
22. **C**
23. **D**
24. **C**
25. **E**
26. **D**
27. **D**
28. **B**
29. **B**
30. **B**
31. **A**
32. **D**
33. **E**
34. **D**
35. **A**
36. **C**

ANSWERS EXPLAINED

1. **(E)** The time and space requirements of sorting algorithms are affected by all three of the given factors, so all must be considered when choosing a particular sorting algorithm.

2. **(D)** Choice B doesn't make sense: The loop will be exited as soon as a value is found that does *not* equal `a[i]`. Eliminate choice A because, if `value` is not in the array, `a[i]` will eventually go out of bounds. You need the `i < n` part of the boolean expression to avoid this. The test `i < n`, however, must precede `value != a[i]` so that if `i < n` fails, the expression will be evaluated as false, the test will be short-circuited, and an out-of-range error will be avoided. Choice C does not avoid this error. Choice E is wrong because both parts of the expression must be true in order to continue the search.

3. **(C)** The binary search algorithm depends on the array being sorted. Sequential search has no ordering requirement. Both depend on choice A, the length of the list, while the other choices are irrelevant to both algorithms.

4. **(B)** Inserting a new element is quick and easy in an unsorted array—just add it to the end of the list. Computing the mean involves finding the sum of the elements and dividing by n, the number of elements. The execution time is the same whether the list is sorted or not. Operation II, searching, is inefficient for an unsorted list, since a sequential search must be used. In `sortedArr`, the efficient binary search algorithm, which involves fewer comparisons, could be used. In fact, in a sorted list, even a sequential search would be more efficient than for an unsorted list: If the search item were not in the list, the search could stop as soon as the list elements were greater than the search item.

5. **(C)** Suppose the array has 1000 elements and x is somewhere in the first 8 slots. The algorithm described will find x using no more than five comparisons. A binary search, by contrast, will chop the array in half and do a comparison six times before examining elements in the first 15 slots of the array (array size after each chop: 500, 250, 125, 62, 31, 15).

6. **(E)** The assertion states that the first element is greater than all the other elements in the array. This eliminates choices A and D. Choices B and C are incorrect because you have no information about the relative sizes of elements `a[1]...a[N-1]`.

7. **(C)** When `key` is not in the array, `index` will eventually be large enough that `a[index]` will cause an `ArrayIndexOutOfBoundsException`. In choices A and B, the algorithm will find `key` without error. Choice D won't fail if 0 is in the array. Choice E will work if `a[key]` is not out of range.

8. **(A)**

After 1st pass:	109	42	−3	13	89	70	2
After 2nd pass:	109	89	−3	13	42	70	2
After 3rd pass:	109	89	70	13	42	−3	2

9. **(A)** The algorithm uses the fact that array `v` is sorted smallest to largest. The `while` loop terminates—which means that the search stops—as soon as `v[index] >= key`.

10. **(C)** The first pass uses the interval `a[0]...a[7]`. Since `mid` = (0 + 7)/2 = 3, `low` gets adjusted to `mid` + 1 = 4, and the second pass uses the interval `a[4]...a[7]`.

11. **(B)** First pass: compare 27 with `a[3]`, since `low` = 0 `high` = 7 `mid` = (0 + 7)/2 = 3. Second pass: compare 27 with `a[5]`, since `low` = 4 `high` = 7 `mid` = (4 + 7)/2 = 5. Third pass: compare 27 with `a[6]`, since `low` = 6 `high` = 7 `mid` = (6 + 7)/2 = 6. The fourth pass doesn't happen, since `low` = 6, `high` = 5, and therefore the test `(low <= high)` fails. Using the general rule for finding the number of iterations when `key` is not in the list: If n is the number of elements, round n up to the nearest power of 2, which is 8 in this case. Note that $8 = 2^3$. Since 27 lies between 4 and 41, there will be 3 iterations of the "divide-and-compare" loop.

12. **(D)** The method returns the index of the `key` parameter, 4. Since `a[0]` contains 4, `binSearch(4)` will return 0.

13. **(A)** Try 4. Here are the values for `low`, `high`, and `mid` when searching for 4:

First pass:	`low` = 0,	`high` = 7,	`mid` = 3
Second pass:	`low` = 0,	`high` = 2,	`mid` = 1

After this pass, `high` gets adjusted to `mid` −1, which is 0. Now `low` equals `high`, and the test for the `while` loop fails. The method returns −1, indicating that 4 wasn't found.

14. **(E)** When the loop is exited, either `key` = `a[mid]` (and `mid` has been returned) or `key` has not been found, in which case either `a[low]` ≤ `key` ≤ `a[high]` or `key` is not in the array. The correct assertion must account for all three possibilities.

15. **(A)** $30{,}000 = 1000 \times 30 \approx 2^{10} \times 2^5 = 2^{15}$. Since a successful binary search in the worst case requires $\log_2 n$ iterations, 15 iterations will guarantee that `key` is found. (Note that $30{,}000 < 2^{10} \times 2^5 = 32{,}768$.) Shortcut: $30{,}000 < 2^{15}$. Therefore, the maximum (worst case) number of comparisons that guarantees the key is found is equal to the exponent, 15.

16. **(B)** Start with the second element in the array.

After 1st pass:	7	1	9	5	4	12
After 2nd pass:	9	7	1	5	4	12
After 3rd pass:	9	7	5	1	4	12

17. **(B)** An insertion sort compares `a[1]` and `a[0]`. If they are not in the correct order, `a[0]` is moved and `a[1]` is inserted in its correct position. `a[2]` is then inserted in its correct position, and `a[0]` and `a[1]` are moved if necessary, and so on. Since B has only one element out of order, it will require the fewest changes.

18. **(A)** This list is almost sorted in reverse order, which is the worst case for insertion sort, requiring the greatest number of comparisons and moves.

19. **(D)** `j >= 0` is a stopping condition that prevents an element that is larger than all those to the left of it from going off the left end of the array. If no error occurred, it means that the largest element in the array was `a[0]`, which was true in situations I and II. Omitting the `j >= 0` test will cause a run-time (out-of-range) error whenever `temp` is bigger than all elements to the left of it (i.e., the insertion point is 0).

20. **(A)** Look at a small array that is almost sorted:

    10 8 9 6 2

    For insertion sort you need four passes through this array.
    The first pass compares 8 and 10—one comparison, no moves.
    The second pass compares 9 and 8, then 9 and 10. The array becomes
    10 9 8 6 2—two comparisons, two moves.
    The third and fourth passes compare 6 and 8, and 2 and 6—no moves.
    In summary, there are approximately one or two comparisons per pass and no more than two moves per pass.
    For selection sort, there are four passes too.
    The first pass finds the biggest element in the array and swaps it into the first position. The array is still 10 8 9 6 2—four comparisons. There are two moves if your algorithm makes the swap in this case, otherwise no moves.
    The second pass finds the biggest element from `a[1]` to `a[4]` and swaps it into the second position: 10 9 8 6 2—three comparisons, two moves.
    For the third pass there are two comparisons, and one for the fourth. There are zero or two moves each time.
    Summary: 4+3+2+1 total comparisons and a possible two moves per pass.
    Notice that reason I is valid. Selection sort makes the same number of comparisons irrespective of the state of the array. Insertion sort does far fewer comparisons if the array is almost sorted. Reason II is invalid. There are roughly the same number of data movements for insertion and selection. Insertion may even have more changes, depending on how far from their insertion points the unsorted elements are. Reason III is wrong because insertion and selection sorts have the same space requirements.

Optional topic

21. **(E)** In the first pass through the outer `for` loop, the smallest element makes its way to the end of the array. In the second pass, the next smallest element moves to the second last slot, and so on. This is different from the sorts in choices A through D; in fact, it is a bubble sort.

22. **(C)** Reject reason I. Merge sort requires both a `merge` and a `mergeSort` method—*more* code than the relatively short and simple code for insertion sort. Reject reason II. The `merge` algorithm uses a temporary array, which means *more* storage space than insertion sort. Reason III is correct. For long lists, the "divide-and-conquer" approach of merge sort gives it a faster run time than insertion sort.

23. **(D)** Since the search is for a four-letter sequence, the idea in this algorithm is that if

you examine every fourth slot, you'll find a letter in the required sequence very quickly. When you find one of these letters, you can then examine adjacent slots to check if you have the required sequence. This method will, on average, result in fewer comparisons than the strictly sequential search algorithm in choice A. Choice B is wrong. If you encounter a "q," "r," or "s" without a "p" first, you can't have found "pqrs." Choice C is wrong because you may miss the sequence completely. Choice E doesn't make sense.

24. **(C)** The main precondition for a binary search is that the list is ordered.

25. **(E)** This algorithm is just a recursive implementation of a sequential search. It starts by testing if the last element in the array, `a[n-1]`, is equal to `value`. If so, it returns the index `n - 1`. Otherwise, it calls itself with `n` replaced by `n - 1`. The net effect is that it examines `a[n-1]`, `a[n-2]`, ... The base case, `if (n == 0)`, occurs when there are no elements left to examine. In this case, the method returns −1, signifying that `value` was not in the array.

Optional topic

26. **(D)** The `partition` algorithm performs a series of swaps until the pivot element is swapped into its final sorted position (see p. 343). No temporary arrays or external files are used, nor is a recursive algorithm invoked. The `merge` method is used for merge sort, not quicksort.

27. **(D)** Recall the merge sort algorithm:

    Divide `arr` into two parts.
    Merge sort the left side.
    Merge sort the right side.
    Merge the two sides into a single sorted array.

    The `merge` method is used for the last step of the algorithm. It does not do any sorting or partitioning of the array, which eliminates choices A, B, and C. Choice E is wrong because `merge` starts with a *single* array that has two sorted parts.

28. **(B)** Round 600 up to the next power of 2, which is $1024 = 2^{10}$. For the worst case, the array will be split in half $\log_2 1024 = 10$ times. Recall the shortcut: $600 < 2^{10}$, so the worst case equals the exponent, 10.

29. **(B)** If the list is sorted in reverse order, each pass through the array will involve the maximum possible number of comparisons and the maximum possible number of element movements if an insertion sort is used.

30. **(B)** 1 million $= 10^6 = (10^3)^2 \approx (2^{10})^2 = 2^{20}$. Thus, there will be on the order of 20 comparisons.

31. **(A)** A binary search, on average, has a smaller run time than a sequential search. All of the sorting algorithms have greater run times than a sequential search. This is because a sequential search looks at each element once. A sorting algorithm, however, processes *other* elements in the array for each element it looks at.

Optional topic

32. **(D)** Reason I is valid—it's always desirable to hide implementation details from users of a method. Reason II is valid too—since `QuickSort` and `MergeSort` implement the `Sort` interface, they must have a `sort` method with no parameters. But parameters are needed to make the recursion work. Therefore each `sort` requires a helper method with parameters. Reason III is invalid in this particular example of helper methods. There are many examples in which a helper method enhances efficiency (e.g., Example 2 on p. 313), but the `sort` example is not one of them.

(continued)

33. **(E)** Since `Sort` is an interface, you can't create an instance of it. This eliminates choices B and D. The `sort` methods alter the contents of `strArray`. Thus invoking `q.sort()` followed by `m.sort()` means that `m.sort` will always operate on a sorted array, assuming quicksort works correctly! In order to test both quicksort and merge sort on unsorted arrays, you need to make a copy of the original array or create a different array. Eliminate choice A (and B again!), which does neither of these. Choice C is wrong because it calls the *private* `sort` methods of the classes. The `Sort` interface has just a single *public* method, `sort`, with no arguments. The two classes shown must provide an implementation for this `sort` method, and it is this method that must be invoked in the client program.

34. **(D)** The computer should find the number in no more than seven tries. This is because the guessing interval is halved on each successive try:

(1) $100 \div 2 = 50$ numbers left to try
(2) $50 \div 2 = 25$ numbers left to try
(3) $25 \div 2 = 13$ numbers left to try
(4) $13 \div 2 = 7$ numbers left to try
(5) $7 \div 2 = 4$ numbers left to try
(6) $4 \div 2 = 2$ numbers left to try
(7) $2 \div 2 = 1$ number left to try

Seven iterations of the loop leaves just 1 number left to try! Don't forget the shortcut. The algorithm is a binary search of 100 possible elements. Rounding 100 up to the next power of 2 gives $128 = 2^7$. The exponent, 7, is the number of guesses in the worst case.

35. **(A)** The maximum number of guesses is 100. A sequential search means that the computer starts at the first possible number, namely 1, and tries each successive number until it gets to 100. If the user's number is 100, the computer will take 100 guesses to reach it.

36. **(C)** On average the computer will make 50 guesses. The user is equally likely to pick any number between 1 and 100. Half the time it will be less than 50; half the time, greater than 50. So on the average, the distance of the number from 1 is 50.