Solutions to Exercises in Chapter 3

3.1 To return the greatest integer in the array *a*[*left*…*right*]:

1. Set *greatest* to *a*[*left*].
2. For *p* = *left*+1, …, *right*, repeat:
   2.1. If *a*[*p*] > *greatest*, set *greatest* to *a*[*p*].
3. Terminate with answer *greatest*.

To return the position of the greatest integer in the array *a*[*left*…*right*]:

1. Set *greatest* to *a*[*left*], and set *pos* to *left*.
2. For *p* = *left*+1, …, *right*, repeat:
   2.1. If *a*[*p*] < *greatest*, set *greatest* to *a*[*p*], and set *pos* to *p*.
3. Terminate with answer *pos*.

To sum the integers in the array *a*[*left*…*right*]:

1. Set *sum* to 0.
2. For *p* = *left*, …, *right*, repeat:
   2.1. Increment *sum* by *a*[*p*].
3. Terminate with answer *sum*.

To count the number of odd and even integers in the array *a*[*left*…*right*]:

1. Set *odd* to 0, and set *even* to 0.
2. For *p* = *left*, …, *right*, repeat:
   2.1. If *a*[*p*] is odd, increment *odd*.
   2.2. If *a*[*p*] is even, increment *even*.
3. Terminate with answers *odd* and *even*.

To reverse the order of the integers in the array *a*[*left*…*right*]:

1. Set *l* to *left*, and set *r* to *right*.
2. While *l* < *r*, repeat:
   2.1. Swap *a*[*l*] with *a*[*r*].
   2.2. Increment *l*, and decrement *r*.
3. Terminate with answer *true*.

3.2 To test whether the array *a*[*left*…*right*] is sorted in ascending order:

1. For *p* = *left*+1, …, *right*, repeat:
   1.1. If *a*[*p*–1] is greater than *a*[*p*], terminate with answer *false*.
2. Terminate.

The number of comparisons is between 1 and *n*–1, i.e., *n*/2 on average.

Method to test whether the array `a[left…right]` is sorted in ascending order:

```
static boolean isSorted (Comparable[] a,
                int left, int right) {
  for (int p = left+1; p <= right; p++) {
    if (a[p-1].compareTo(a[p]) > 0)
      return false;
  }
  return true;
}
```

3.3 To test whether the character array *a*[*left*…*right*] is a palindrome:

1. Set l to *left*, and set r to *right*.
2. While $l < r$, repeat:
   2.1. If $a[l] \neq a[r]$, terminate with answer *false*.
   2.2. Increment l, and decrement r.
3. Terminate with answer *true*.

The number of comparisons is between 1 and $n/2$, i.e., about $n/4$ on average. Therefore the algorithm's time complexity is $O(n)$. Its space complexity is $O(1)$.

Method to test whether the character array `a[left…right]` is a palindrome:

```
static boolean isPalindrome (char[] a,
                int left, int right) {
  int l = left, r = right;
  while (l < r) {
    if (a[l] != a[r])  return false;
    l++;  r--;
  }
  return true;
}
```

3.4 To test whether the character array $a[left…right]$ is a palindrome, ignoring spaces and punctuation:

1. Set l to *left*, and set r to *right*.
2. While $l < r$, repeat:
   2.1. If $a[l]$ is a space or punctuation, increment l.
   2.2. If $a[r]$ is a space or punctuation, decrement r.
   2.3. Otherwise (if neither $a[l]$ nor $a[r]$ is a space or punctuation):
      2.3.1. If $a[l] \neq a[r]$, terminate with answer *false*.
      2.3.2. Increment l and decrement r.
3. Terminate with answer *true*.

3.5 To be able to insert a value of any Java primitive type T into an array of type `T[]`, we need eight different methods (since Java has eight primitive types). To be able to insert an object into an object array, we need one further method. In total we need nine methods.

3.6 If step 1 of Algorithm 3.6 were implemented by a for statement that scanned from left to right copying each component, the leftmost component would be copied into all other components.

3.7 The exact number of comparisons performed by Algorithm 3.18 depends on what step 2.1 does. If the subarray $a[l…r]$ has an even number of components, step 2.1 cannot set *mid* to a value exactly midway between l and r; instead it must choose a value either (i) slightly closer to l than to r, or (ii) slightly closer to r than to l.

Target	banana	grape	plum	lychees	strawberry
No. of comparisons (i)	2	3	4	4	4
No. of comparisons (ii)	3	2	3	4	3

3.9 In linear search of a sorted array, it would be advantageous to search from right to left in circumstances when it is known that the target lies closer to the right than to the left of the array.

3.10 To delete *val* from the unsorted array $a[left…right]$:

1. For p = *left*, …, *right*, repeat:
   1.1. If *val* is equal to $a[p]$:
      1.1.1. Copy $a[p+1…right]$ into $a[p…right–1]$.
      1.1.2. Make $a[right]$ unoccupied.
      1.1.3. Terminate.
2. Terminate.

To delete *val* from the sorted array *a*[*left*…*right*]:

1. For *p* = *left*, …, *right*, repeat:
   1.1. If *val* is equal to *a*[*p*]:
        1.1.1. Copy *a*[*p*+1…*right*] into *a*[*p*…*right*–1].
        1.1.2. Make *a*[*right*] unoccupied.
        1.1.3. Terminate.
   1.2. If *val* is less than *a*[*p*], terminate.
2. Terminate.

To insert *val* in the sorted array *a*[*left*…*right*]:

1. For *p* = *right*, …, *left*, repeat:
   1.1. If *val* is less than *a*[*p*]:
        1.1.1. Copy *a*[*p*] into *a*[*p*+1].
   1.2. If *val* is greater than or equal to *a*[*p*]:
        1.2.1. Copy *val* into *a*[*p*+1].
        1.2.2. Terminate.
2. Copy *val* into *a*[*left*].
3. Terminate.

(*Note:* This algorithm overwrites *a*[*right*+1], assuming that it exists.)

To find the least component of the unsorted array *a*[*left*…*right*]:

1. Set *least* to *a*[*left*].
2. For *p* = *left*+1, …, *right*, repeat:
   1.1. If *a*[*p*] is less than *least*, set *least* to *a*[*p*].
2. Terminate with answer *least*.

All these algorithms have time complexity $O(n)$.

3.11 The directory should be sorted by name, allowing the most frequently-called method `searchByName` to be implemented using binary search. Methods:

```
static String searchByName
              (DirectoryEntry[] dir,
               String targetName) {
  int l = 0, r = dir.length - 1;
  while (l <= r) {
    int m = (l + r)/2;
    int comp =
         targetName.compareTo(dir[m].name);
    if (comp == 0)
      return dir[m].number;
    else if (comp < 0)
      r = m - 1;
    else  // comp > 0
      l = m + 1;
  }
  return null;
}
```

```
static String[] searchByNumber
               (DirectoryEntry[] dir,
                String targetNumber) {
   String[] names1 = new String[dir.length];
   int count = 0;
   for (int p = 0; p < dir.length; p++) {
      if (targetNumber.equals(dir[p].number))
         names1[count++] = dir[p].name;
   }
   if (count == 0)
      return null;
   else {
      String[] names2 = new String[count];
      System.arraycopy(names1, 0, names2, 0,
            count);
      return names2;
   }
}
```

3.13 To compute the union of *s1*[*left1*…*right1*] and *s2*[*left2*…*right2*] in *s3*[*left3*…]:

1. Set *i* to *left1*, set *j* to *left2*, and set *k* to *left*.
2. While $i \le right1$ and $j \le right2$, repeat:
   2.1. If *s1*[*i*] is equal to *s2*[*j*]:
      2.1.1. Copy *s1*[*i*] into *s3*[*k*].
      2.1.2. Increment *i*, *j*, and *k*.
   2.2. Otherwise, if *s1*[*i*] is less than *s2*[*j*]:
      2.2.1. Copy *s1*[*j*] into *s3*[*k*].
      2.2.2. Increment *i* and *k*.
   2.3. Otherwise, if *s1*[*i*] is greater than *s2*[*j*]:
      2.3.1. Copy *s2*[*j*] into *s3*[*k*].
      2.3.2. Increment *j* and *k*.
3. While $i \le right1$, repeat:
   3.1. Copy *s1*[*i*] into *s3*[*k*].
   3.2. Increment *i* and *k*.
4. While $j \le right2$, repeat:
   4.1. Copy *s2*[*j*] into *s3*[*k*].
   4.2. Increment *j* and *k*.
5. Terminate.

To compute the intersection of *s1*[*left1*…*right1*] and *s2*[*left2*…*right2*] in *s3*[*left3*…]:

1. Set *i* to *left1*, set *j* to *left2*, and set *k* to *left*.
2. While $i \le right1$ and $j \le right2$, repeat:
   2.1. If *s1*[*i*] is equal to *s2*[*j*]:
      2.1.1. Copy *s1*[*i*] into *s3*[*k*].
      2.1.2. Increment *i*, *j*, and *k*.
   2.2. Otherwise, if *s1*[*i*] is less than *s2*[*j*]:
      2.2.1. Increment *i*.
   2.3. Otherwise, if *s1*[*i*] is greater than *s2*[*j*]:
      2.3.1. Increment *j*.
3. Terminate.

3.15 To read values from the unsorted file *f* into a sorted array *a*[0…] (version 1):

1. Set *m* to 0.
2. While not at end of file *f*, repeat:
   2.1. Read value *val* from *f*.
   2.2. Copy *val* into *a*[*m*].
   2.3. Increment *m*.
3. Sort *a*[0…*m*–1].
4. Terminate.

Step 2 performs 0 comparisons. If step 3 uses selection sort, it performs about $n^2/2$ comparisons. Version 1 therefore performs about $n^2/2$ comparisons.

To read values from the unsorted file *f* into a sorted array *a*[0…] (version 2):

1. Set *m* to 0.
2. While not at end of file *f*, repeat:
   2.1. Read value *val* from *f*.
   2.2. Insert *val* in the sorted array *a*[0…*m*–1].
   2.3. Increment *m*.
3. Terminate.

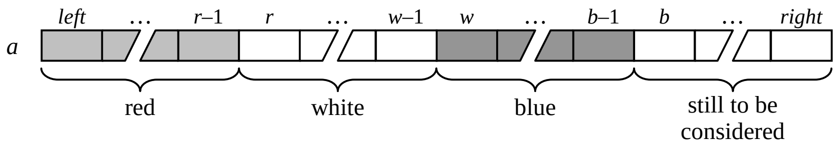
Step 2.2 would use the sorted-array insertion algorithm of Exercise 3.10 (above). This performs about $m/2$ comparisons. Since *m* ranges from 0 to *n*–1, the total number of comparisons is $0 + 1/2 + ... + (n–1)/2 = n(n–1)/4 \approx n^2/4$.

Both versions have time complexity $O(n^2)$, but version 2 is about twice as fast as version 1.

3.16 To sort an array of colors *a*[*left*…*right*] into the order red–white–blue:

1. Set *r* to *left*, set *w* to *left*, and set *b* to *left*.
2. While $b \le right$, repeat:
   2.1. If *a*[*b*] is blue:
      2.1.1. Increment *b*.
   2.2. If *a*[*b*] is white:
      2.2.1. If $b > w$, swap *a*[*b*] with *a*[*w*].
      2.2.2. Increment *w* and *b*.
   2.3. If *a*[*b*] is red:
      2.3.1. If $b > r$, swap *a*[*b*] with *a*[*r*].
      2.3.2. If $b > w$, swap *a*[*b*] with *a*[*w*].
      2.3.3. Increment *r*, *w*, and *b*.
3. Terminate

The loop invariant is:



This algorithm performs 1 color comparison and at most 4 copies per iteration, i.e., *n* color comparisons and at most 4*n* copies in total. Its time complexity is $O(n)$.

3.17 Let $n_1 = right1 – left1 + 1$, $n_2 = right2 – left2 + 1$, and $n = n_1 + n_2$.

To copy all values from unsorted arrays *a1*[*left1*…*right1*] and *a2*[*left2*…*right2*] into sorted array *a3*[*left3*…] (version 1):

1. Concatenate *a1*[*left1*…*right1*] and *a2*[*left2*…*right2*] into *a3*[*left3*…].
2. Sort *a3*[*left3*…].
3. Terminate.

Step 1 performs 0 comparisons. If step 2 uses (say) selection sort, it performs about $n^2/2$ comparisons. The total number of comparisons is therefore about $n^2/2 = (n_1 + n_2)^2/2 = n_1^2/2 + n_2^2/2 + n_1n_2$.

To copy all values from unsorted arrays *a1*[*left1*…*right1*] and *a2*[*left2*…*right2*] into sorted array *a3*[*left3*…] (version 2):

1. Sort *a1*[*left1*…*right1*].
2. Sort *a2*[*left2*…*right2*].
3. Merge *a1*[*left1*…*right1*] and *a2*[*left2*…*right2*] into *a3*[*left3*…].
4. Terminate.

If steps 1 and 2 use selection sort, they perform about $n_1^2/2$ and $n_2^2/2$ comparisons, respectively. Step 3 performs about $n = n_1 + n_2$ comparisons. The total number of comparisons is therefore about $n_1^2/2 + n_2^2/2 + n_1 + n_2$.

For all but small values of n_1 and n_2, $n_1 + n_2 < n_1 n_2$. Therefore version 2 is faster than version 1.

3.18 To find the position of the leftmost subarray of *a*[0…*n*–1] that matches *b*[0…*m*–1] (assuming that $m \leq n$):

1. For *p* = 0, …, *n*–*m*, repeat:
   1.1. If *a*[*p*…*p*+*m*–1] matches *b*[0…*m*–1], terminate with answer *p*.
2. Terminate with answer *none*.

To determine whether *a*[*p*…*p*+*m*–1] matches *b*[0…*m*–1]:

1. For *d* = 0, …, *m*–1, repeat:
   1.1. If *a*[*p*+*d*] is unequal to *b*[*d*], terminate with answer *false*.
2. Terminate with answer *true*.

The auxiliary algorithm performs between 1 and *m* comparisons, i.e., (*m*+1)/2 comparisons on average. The main algorithm performs between 1 and *n* iterations, i.e., (*n*+1)/2 iterations on average. Therefore it performs (*m*+1)(*n*+1)/4 comparisons on average. Its time complexity is $O(mn)$.

3.20 Bubble sort method:

```
static void bubbleSort (Comparable[] a,
            int left, int right) {
  for (int i = 0; i <= right-left-1; i++) {
    for (int j = left+1; j <= right-i; j++) {
      int comp = a[j-1].compareTo(a[j]);
      if (comp > 0) {
        Comparable temp = a[j-1];
        a[j-1] = a[j];  a[j] = temp;
      }
    }
  }
}
```

The bubble-sort algorithm performs about $n^2/2$ comparisons and (on average) about $n^2/2$ copies (counting a swap as 2 copies). Thus it is slower than either selection sort or insertion sort, although all three have time complexity is $O(n^2)$.

3.21 Shell sort method:

```java
static void shellSort (Comparable[] a,
            int left, int right) {
  int gap = right - left + 1;
  do {
    gap = gap / 2;
    if (gap % 2 == 0)  gap++;
    for (int i = gap; i <= right; i++) {
      Comparable current = a[i];
      int j = i - gap;
      while (j > left &&
          current.compareTo(a[j]) < 0) {
        a[j+gap] = a[j];
        j -= gap;
      }
      a[j+gap] = current;
    }
  } while (gap != 1);
}
```