1. One approach is to look for a statement that is executed at least as many times as other statements in the code. In part **a)**, we can use the statement `B[i] = B[i] + A[j]` in the nested *for* loop. For each value of $i = 0, 1, \ldots n-1$, the inner *for* loop runs for i executions, and the statement in the inner loop is executed i times. The total number of executions is $0 + 1 + 2 + \cdots + (n-1)$, which we know is $\Theta(n^2)$. (It is exactly $n(n-1)/2$.) The execution of other statements in the code segment add other constant multiples of n^2 or lower order terms, which does not change the Big-Oh analysis. So the run time for part **a)** is $\Theta(n^2)$.

For part **b)**, the statement `C[i]=A[i]*B[i]` is executed n times and nothing else is executed more often, so the run time for the *for* loop is $\Theta(n)$. (The first statement, which calls *new int[n]*, is more problematic. That statement does not actually execute in constant time in all cases. In fact, it can trigger a garbage collection that can take an undetermined amount of time. In almost all cases, however, it will not take longer than $\Theta(n)$, which does not change the analysis.)

2. **a)** This algorithm checks whether the array contains any duplicate elements. That is, it returns *true* if $A[i] = A[j]$ for some i and j with $j > i$. (Note that it is essential that the inner *for* loop start with $j = i + 1$ to avoid comparing $A[i]$ to itself, which would always return a *true* result.)

b) In the worst case, A does not contain any duplicates, which means that the test in the *if* statement is always *false*, and the code does not end until the *for* loops have run completely. In that case, the comparison is done $(n-1) + (n-2) + \cdots + 2 + 1$ times, and the run time is $\Theta(n^2)$. The best case occurs when $A[0]$ equals $A[1]$, so the code ends after doing just one comparison, so that the run time is $\Theta(1)$.

c) After the array has been sorted, any pair of duplicate elements will be next to each other in the array, so we only have to compare each element to the element that immediately follows it in the array. Using mergeSort to do the sorting, the algorithm is

```
mergeSort(A);
for (int i = 0; i < n-1; i++) {
  if (A[i] == A[i+1])
    return false;
}
return true;
```

The run time for *mergeSort* is $\Theta(n * \log(n))$, and the time for the *for* loop is $O(n)$. Since n is a lower-order term compared to $n * \log(n)$, the total run time is $\Theta(n * \log(n))$ (in all cases).

3. There are many algorithms that will work (including the one that I discussed in class that uses only $n + \log_2(n)$ comparisons). Perhaps the most natural one is:

```
first = A[0]; // this will be the largest element
second = A[1]; // this will be the second-largest element
if (A[1] > A[0]) { // they are in the wrong order, so swap them
    temp = A[0];
    A[0] = A[1];
    A[1] = temp;
}
```

```
for ( i = 2; i < N; i++ ) {
    if (A[i] > second) { // A[i] becomes first or second largest
        if (A[i] < first) { // A[i] is not the first largest
            second = A[i];
        }
        else { // A[i] is first largest
            second = first; // The previous first largest moves down
            first = A[i];
        }
    }
}
```

This algorithm clearly has run time $\Theta(n)$, but the question that was asked is how many comparisons does it make. There is 1 comparison at the start and at least $N-2$ comparisons in the *for* loop. Depending on how often the first comparison in the loop succeeds or fails, there can be up to $N-2$ additional comparisons. So, the total number of comparisons is in the range $N-1$ to $2*N-3$, with an average of about $\frac{3}{2}N$. (The best case is when $A[0]$ and $A[1]$ are already the first and second elements, in which case there are exactly $N-1$ comparisons. The worst case is when the array is in increasing order, in which case there are exactly $2N-3$ comparisons.)

(Note: It is not necessary to modify the array in order to efficiently solve this problem. It is certainly not necessary to sort the array completely.)

4. If we can find an index p such that $A[p] >= X$, then, since the array is sorted, we know that N, the index of X in the array, must be less than or equal to p. Once we have p, we can use *binarySearch*(A,0,p) to find N. (Use a version of binary search that returns the index of the value that is being looked for, not just true or false.) An algorithm that does this is:

```
p = 1;
while (A[p] < X)
  p = 2*p;
return binarySearch(A,0,k);
```

When the while loop ends, p is the first power of 2 such that $N \leq p$. If $p = 2^k$, then k is the number of times that the *while* loop has run, and k is approximately $\log_2(N)$. (It is exactly $\lceil \log_2(N) \rceil$.) So the run time for the *while* loop is $\Theta(\log(N))$. We know that binary search on the subarray from 0 to 2^k also takes k steps, so the run time for the binary search is also $\Theta(\log(N))$.

5. **a)** When applying the Master Theorem to this recurrence relation, b = 3 and a = 9, so $\log_b(a) = \log_3(9) = \log_3(3^2) = 2$. The exponent on the "extra work" function, n^2, is also 2, so this is Case 2 of the Master Theorem, and we get $T(n) = \Theta(n^{\log_b(a)} \log(n)) = \Theta(n^2 \log(n))$.

b) When applying the Master Theorem to this recurrence relation, b = 3 and a = 9, so $\log_b(a) = \log_3(9) = \log_3(3^2) = 2$. The exponent on the "extra work" function, n^3, is 3, which is greater than $\log_b(a)$, so this is Case 3 of the Master Theorem, and we get that $T(n)$ is the same as the extra work function, $T(n) = \Theta(n^3)$.

c) When applying the Master Theorem to this recurrence relation, b = 4 and a = 5, so $\log_b(a) = \log_4(5)$, which is about 1.161. The exponent on the "extra work" function, n, is 1, which is less than $\log_b(a)$, so this is Case 1 of the Master Theorem, and we get that $T(n) = \Theta(n^{\log_b(a)}) = \Theta(n^{1.161})$.