```
/*
* NAME: Megan Chu
* ID: A12814536
* LOGIN: cs12waot
*/
```

Problem 1. A = (1, 5, 3, 2, 0, 4, 6, 8)

j	Action	Array contents
0	Add 1 to I, I = 0, exchange A[0] with A[0]	1, 5, 3, 2, 0, 4, 6, 8
1	Add 1 to I, I = 1, exchange A[1] with A[1]	1, 5, 3, 2, 0, 4, 6, 8
2	Add 1 to I, I = 2, exchange A[2] with A[2]	1, 5, 3, 2, 0, 4, 6, 8
3	Add 1 to I, I = 3, exchange A[3] with A[3]	1, 5, 3, 2, 0, 4, 6, 8
4	Add 1 to I, I = 4, exchange A[4] with A[4]	1, 5, 3, 2, 0, 4, 6, 8
5	Add 1 to I, I = 5, exchange A[5] with A[5]	1, 5, 3, 2, 0, 4, 6, 8
6	Add 1 to I, I = 6, exchange A[6] with A[6]	1, 5, 3, 2, 0, 4, 6, 8

Last line says to exchange A[i + 1] with A[r], which is exchanging A[7] with A[7]. Then we return i + 1, which is returning 7.

The resulting array is the same as the given one: A = (1, 5, 3, 2, 0, 4, 6, 8)

Problem 2. To sort the numbers {5, 7, 1, 9, 45, 23, 7} in decreasing order using:

a. Selection sort

Each iteration of the outer loop	Resulting array
0	5, 7, 1, 9, 45, 23, 7
1	**1,** 5, 7, 9, 45, 23, 7
2	**1, 5,** 7, 9, 45, 23, 7
3	**1, 5, 7,** 7, 9, 45, 23
4	**1, 5, 7, 7,** 9, 45, 23
5	**1, 5, 7, 7, 9,** 23, 45
6	**1, 5, 7, 7, 9, 23, 45**

b. Insertion sort

Each iteration of the outer loop	Resulting array
0	5, 7, 1, 9, 45, 23, 7
1	**5, 7,** 1, 9, 45, 23, 7
2	**1, 5, 7,** 9, 45, 23, 7
3	**1, 5, 7, 9,** 45, 23, 7
4	**1, 5, 7, 9, 45,** 23, 7
5	**1, 5, 7, 9, 23, 45,** 7
6	**1, 5, 7, 7, 9, 23, 45**

c. Bubble sort

Each iteration of the outer loop	Resulting array
0	5, 7, 1, 9, 45, 23, 7
1	5, 1, 7, 9, 23, 7, **45**
2	1, 5, 7, 9, 7, **23, 45**
3	1, 5, 7, 7, **9, 23, 45**
4	**1, 5, 7, 7, 9, 23, 45**

Problem 3. To rearrange an array of n keys so that all the negative keys precede the nonnegative keys, we can loop through the array from the from the front to the end. We start with a variable storing the last index of the array. Once our loop reaches a nonnegative key, we swap that item with the last item of the array, and decrement the variable storing the last index. We continue this until the loop position equals the variable storing the last index. This would have runtime O(n) in the worst case where there are no nonnegative numbers in the list.

```
reArrange(A is an array)
{
      int end = A.length - 1;
      for(int i = 0; i < end; i++)
      {
            if(A[i] < 0)
            {
                  temp = A[i];
                  A[i] = A[end];
                  A[end] = temp;
                  end--;
            }
      }

}
```

Problem 4.
1. Without a sorted list, we would have to loop through the list with runtime O(n^2) in the worst case where m = n.

With a sorted list, we would only have to loop through the list one time, O(n), in the worst case where m = n. The best possible sorting time is O(n), using something like bucket sort, so the total runtime is O(n + n) or O(n). If we use something like quick sort or merge sort, we would have total runtime of O(nlogn + n) or O(nlogn). So it would be more efficient to have a sorted list in this case.

2. Without a sorted list, we would have to loop through the list as many times as there are unique integers. In the worst case this is O(n^2) if all the integers in the list are unique.

With a sorted list, we would only have to loop through the list one time, O(n) in any case. The best possible sorting time is O(n), so the total runtime is O(n + n) or O(n). On top of this, if we use a more average sort like merge or quicksort, we get a total running time of O(nlogn + n) or O(nlogn). It is more efficient to sort the list for this problem.

3. To compute the median of two arrays, you would basically have to reference all the elements in the array in their sorted order, except they are not in that order, so it would take much much longer to loop through the arrays to find the the elements in sorted order than actually sorting them. It would be more

efficient to sort both arrays and merge them together using Mergesort, and then access the middle of the array or average it. This has a runtime of O(nlogn) for MergeSort (where n is the length of the combined list), and constant running time for calculating the median of the sorted list.

4. Even if the array was sorted (requiring extra runtime), we would still have to loop through the entire array once to find each time the particular item is in the array. This is in the worst case where the item does not exist in the array, or is the last item in the array. Therefore it is faster to run through an unsorted array that does not require extra runtime to sort. Similarly to a sorted array, we only need to loop through the array once to find occurrences of a particular item. This takes runtime O(n).

5. Finding the minimum value in an unsorted array takes runtime O(n), looping through the array, and storing a smaller value than the current value if we find one. In a sorted array, we can just get the first value in the array, which is already the smallest, this is constant time. However, the time it takes to sort the array using an average runtime sort is O(nlogn), which is longer than finding the value in an unsorted array. Even with the fastest sort, it would still take O(n) time to find the minimum value, so sorting the array isn't worth it for this problem.