Name:_____

1. Recall the iterative (i.e., using a loop) *binary search* on an array sorted in *ascending order*.

```
//*************************************************************
// The binarySearch function performs a binary search on an     *
// integer array. array, which has a maximum of size elements,  *
// is searched for the number stored in value. If the number is *
// found, its array subscript is returned. Otherwise, -1 is     *
// returned indicating the value was not in the array.          *
//*************************************************************
int binarySearch(int array[], int size, int value) {
   int first = 0,              // First array element
       last = size - 1,        // Last array element
       middle,                 // Mid point of search
       position = -1;          // Position of search value
   bool found = false;         // Flag

   while (!found && first <= last) {
      middle = (first + last) / 2;     // Calculate mid point
      if (array[middle] == value) {    // If value is found at mid
         found = true;
         position = middle;
      } else if (array[middle] > value) {  // If value is in lower half
         last = middle - 1;
      } else {
         first = middle + 1;           // If value is in upper half
      } // end if
   } // end while
   return position;
} // end binarySearch
```

a) Trace the `binarySearch` code using the following actual parameters by showing the changes to first, last, middle, position, and found.

	0	1	2	3	4	5	6	(MAX-1)
array:	2	3	4	5	7	8	9	

size: 7 value: 7

first	last	middle	position	found
0	6		-1	false

b) How might we think of this search problem recursively, i.e., solving a searching problem by splitting the problem into one or more simplier search problems?

c) What additional parameter(s) would be needed to specify the problem recursively? (needed to specify the smaller problem size)

d) What base case(s) are trivial enough that the answer is obvious?

e) Write the code for recursive binarySearch.

Name:_____

2. So far, we have only looked at simple sorts. Recall that all simple sorts consist of nested loops:

- an outer loop that keeps track of the dividing line between the sorted and unsorted part
- an inner loop that grows the size of the sorted part by one

Usually, the number of inner loop iterations is something like = (n-1) + (n-2) + (n-3) + ... + 3 + 2 + 1 = n * (n-1)/2. which is "big-oh" of n^2 (i.e., $O(n^2)$).

Next, we'll look at a couple of more "advanced" sorting algorithms that are recursive: Merge sort, and Quick Sort. In general, a problem can be solved recursively if it can be broken down into smaller problems that are identical in structure to the overall problem.

a) What determines the "size" of a sorting problem?

b) How might we break the original problem down into smaller problems that are identical? Are there any additional parameters that might be needed? (remember recursive binary search needed extra parameters)

c) What base case(s) (i.e., trival, non-recursive case(s)) might we encounter with recursive sorts?

d) Consider why a recursive sort might be more efficient. Assume that I had a simple n^2 sorting algorithm with n = 100, then there is roughly 100^2 or 10,000 amount of work. Suppose I split the problem down into two smaller problems of size 50.

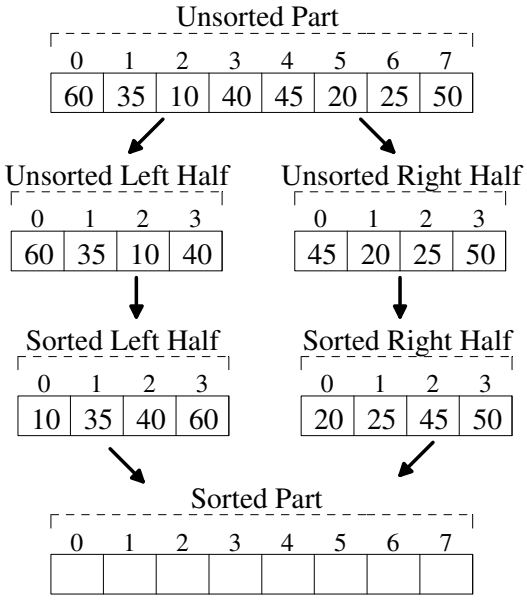
- If I run the n^2 algorithm on both smaller problems of size 50, then what would be the approximate amount of work?

- If I further solve the problems of size 50 by splitting each of them into two problems of size 25, then what would be the approximate amount of work?

Name:_____

3. The general idea merge sort is as follows. Assume "n" items to sort.

- Split the unsorted part in half to get two smaller sorting problems of about equal size = n/2
- Solve both smaller problem recursively using merge sort
- "Merge" the solution to the smaller problems together to solve the original sorting problem of size n



a) Fill in the sorted part in the above.

b) Describe how you filled in the sorted part in the above example?

c) Complete the code that merges the two sorted lists back into a larger list.

```
void merge(int sorted[], int & sortedCount, int leftHalf[], int leftCount,
                                            int rightHalf[], int rightCount) {
```