Question 1

i).

A recursive function is one that references themselves. A function that may call itself during its execution. This can lead to an infinite loop unless the recursive function is well defined. A well defined recursive function has a “base case,” and must “move towards” the base case (reduce). Basically the recursive function cannot be infinite.

```
L1   // curr should be size-1 for the first call
L2   int find1(int arrA[], int size, int curr)
L3   {
L4     if (size == 1) {
L5           return (curr);
L6       }
L7     else if ( arrA[curr] < arrA[size - 2] ) {
L8           return (find1 (arrA, size - 1, size - 2));
L9       }
L10      else {
L11          return (find1 (arrA, size - 1, curr));
L12      }
L13  }
L14
L15  int find2(int arrA[], int size)
L16  {
L17    int curr = 0;
L18    for (int i = 1; i < size; i++) {
L19          if (arrA[i] > arrA[curr]) {
L20              curr = i;
L21          }
L22      }
L23    return (curr);
L24  }
```

FOR find1()

Base case: At the base case the function will not call itself again.

The base case check is at line 4. The base case return is on line 5. When size is 1 the function will return a value and no more recursive calls are made.

Reduction: Every call of the function will reduce it towards the base case.

This happens at Line 8 and line 11 because size is reduced by 1 when the recursive function is called again. This means eventually size will equal 1, making line 4 true, ending the recursive function.

ii).

find1()

Line	Cost	numTimes	Cost*numTimes
L4	1	n	n
L5	1	1	1
L7	1	n - 1	n - 1
L8 or L11	1	n - 1	n - 1
		Total:	f(n) = 3n - 1

L4: Always checked. True n - 1 times and false 1 time.

L5: Only happens once.

L7: Always checked, besides the last function call when L4 is true.

L8 and L11: One of them will run because of the “else”.

find2()

Line	Cost	numTimes	Cost*numTimes
L17	1	1	1
L18	1	n	n
L19	1	n - 1	n - 1
L20	1	n - 1	n - 1
L23	1	1	1
		Total:	f(n) = 3n

size = n.

L17: Happens once.

L18: True n - 1 times. Then false 1 time. So it runs n times.

L19: Checked n - 1 times.

L20: Worst case scenario, arrA[i] is always bigger than arrA[curr]. (sorted in ascending order). Therefore, L19 is always true, therefore L20 happens the same number of times as L19. So L20 runs n - 1 times.

L23: Happens once.

Question 2

Imagine we split the range of elements given in arrA into two sub arrays (lower bound -> mid AND mid+1 -> upper bound). (This is done in the declarations made in the for loop in L20). We are going to populate a new array, arrC, one element of arrC at a time using the 2 sub-arrays of arrA. A counter declared as "k" in the code given will keep track of the population of arrC. The left-most unused (not already placed into arrC) element for the left sub-array of arrA is compared to the left-most unused element for the right sub-array of arrA - the element with the lower value is placed into arrC. A counter is incremented to keep track of the left-most unused element for each of the sub-arrays - "i++" in L22 and "j++" in L24.

First loop…

arrA

1	2	4	6	3	5	7	8

^ last of left sub-array

Compare the red values.

arrC

1							

The lower value goes into arrC. And the counter for its sub-array increments. (i++).

… after last loop (i <= mid is no longer true so the loop is exited).

arrA

1	2	4	6	3	5	7	8

^ last of left sub-array

arrC

1	2	3	4	5	6		

When one of the sub-arrays reaches its end, L20 will return false. This sub-array has been fully used, so the rest of the other sub-array must be placed into arrC (this is because the rest of the values in the other sub-array will all be bigger than the values of the sub-array that is at its end). Either L27 will run or L29 will run.

The rest of the sub-array is placed into arrC. In this example L28 is true 2 times. and L29 happens twice.

arrA

1	2	4	6	3	5	7	8

^ last of left sub-array

arrC

1	2	3	4	5	6	7	8

Finally, place the elements of arrC into arrA, overwriting the unsorted section of arrA with the sorted array that has been calculated and stored in arrC. (L30 and L31).

arrA

1	2	4	6	3	5	7	8

document continues on next page...

Question 3

a)

	Count sort	**Merge sort**	**Quick sort**
Recursive calls		9 998	9 002
Swaps		61 808	34 636
Comparisons		55 296	84 507
Time taken (ms)	1	5	4

```
Count sort...
Before the sort, run time = 70 clock ticks
Before the sort, run time = 71 clock ticks
Time taken to sort: 1 miliseconds
```

b)

	Count sort	**Merge sort**	**Quicksort**
Advantages	Fastest.	Works for general data.	Works for general data.
Disadvantages	Only works for integer data.	REQUIRES SPACE for the temporary array (arrC). Slowest.	A bad pivot choice makes quicksort perform worse.
Time taken (ms)	**1**	**5**	**4**

Merge sort

```
Merge sort...
Swaps = 61808
Comparisons = 55296
Recursive calls = 9998

Before the sort, run time = 64 clock ticks
Before the sort, run time = 69 clock ticks
Time taken to sort: 5 miliseconds
```

Recursive call counter increments in mergeSort() function. Line 398 and line 400.

```
389 //mergeSort to sort values in an integer array arrA[]
390 // lb = 0 and ub = size - 1 for the first call
391 void mergeSort(int arrA[], int lb, int ub) {
392 
393     int mid;
394 
395     if (lb < ub) {
396         mid = (int)((lb + ub) / 2);
397         mergeSort(arrA, lb, mid);
398         reCalls++;
399         mergeSort(arrA, mid + 1, ub);
400         reCalls++;
401         merge(arrA, lb, mid, ub);
402     }
403 }
```

Swap counter increments in write-back section of code in merge() function. Line 454.

```
449     //write back from arrC to arrA so correct values are in place for next merge
450     i = lb;
451     k = 0;
452     while (i <= ub) {
453         arrA[i] = arrC[k];
454         swaps++;
455         i++;
456         k++;
457     }
458 }
```

Comparisons counter increments in merge() function. Line 428.

```
419     while (i <= mid && j <= ub) {
420         if (arrA[i] <= arrA[j]) {
421             arrC[k] = arrA[i];
422             i++;
423         }
424         else {
425             arrC[k] = arrA[j];
426             j++;
427         }
428         comparisons++;
429         k++;
430     } //end while
```

Quicksort

```
Quicksort...
Swaps = 34636
Comparisons = 84507
Recursive calls = 9002

Before the sort, run time = 70 clock ticks
Before the sort, run time = 74 clock ticks
Time taken to sort: 4 miliseconds
```

Recursive calls counter increments in quicksort() function. Line 286 and line 288.

```
275 //QUICKSORT
276
277 void quickSort(int arrA[], int startval, int endval) {
278
279     if ((endval - startval) < 1) {
280         return;
281     }
282     else {
283         int k = partition2(arrA, startval, endval);
284         //now sort the two sub-arrays
285         quickSort(arrA, startval, k - 1);  //left partition
286         reCalls++;
287         quickSort(arrA, k + 1, endval);   //right partition
288         reCalls++;
289     }
290 }
```

Swap counter increments in partition2() function. Line 367 and line 373.

Comparisons counter increments in partition2() function. Line 370.


```
275 //QUICKSORT
276
277 void quickSort(int arrA[], int startval, int endval) {
278
279     if ((endval - startval) < 1) {
280         return;
281     }
282     else {
283         int k = partition2(arrA, startval, endval);
284         //now sort the two sub-arrays
285         quickSort(arrA, startval, k - 1);  //left partition
286         reCalls++;
287         quickSort(arrA, k + 1, endval);   //right partition
288         reCalls++;
289     }
290 }
```

Question 4

Plagiarism Declaration: “I am aware of what plagiarism is and include this here to confirm that this work is my own.”