Ryan Vollmer
CS 325
HW 2

1)

a) $T(n) = T(n-2) + n$
$= T(n-4) + n + n$
$= T(n-6) + n + n + n$
$= T(1) + n + ... + n$
$= n((n-1) / 4) + T(1)$
$= (1/4)n^2 - (1/4)n + T(1)$
$= \theta(n^2)$

b) $T(n) = 3T(n - 1) + 1$
$= 3 * [3 * T(n-2) + 1] + 1 = 3^2 * T(n-2) + 3 + 1$
$= 3^2 * [3 * T(n-3) + 1] + 3 + 1 = 3^3 * T(n-3) + 3^2 + 3 + 1$
$= 3^k * T(n-k) + 1 + 3 + 3^2 + ... + 3^{(n-k-1)}$
$= 3^{(n-1)} * T(1) + 1 + 3 + 3^2 + ... 3^{(n-2)}$
$= 1 + 3 + 3^2 + ... 3^{(n-1)}$
$= \theta(3^n)$

c) $T(n) = 2T(n/8) + 4n^2$
by master theorem:
$a=2$, $b=8$, $f(n)=4n^2$, $c=2$
$\log_b(a) = \log_8(2) = 1/3 < c$
$T(n) = \theta(n^2)$

2)

a) it becomes sorted because:
1) If it is 1 element in A it is sorted
2) If it contains 2 elements it becomes sorted in in the swap statement
3) If an array is more than 2 elements it eventually gets broken down to be 2 elements, which then get sorted due to the following:
a) the first recursive call breaks all elements down by 2/3 the original size, until they become a sorted pair and one additional value (3 elements).
b) the second call breaks all elements down to the last 2/3, until they become a sorted pair and one additional value (3 elements).
c) The third call does the same as case a, therefore sorting the first 2/3 into the last 2/3 of all values, which in combination with all elements being reduced until they are size of 2 means they will be swapped into their correct positions.

b) No, it would not sort correctly. A counterexample is if n=4 for the array A[0 ... n], then the first run through k=floor(2*4/3) = 2. the 3 recursive call cases in the first run through would then be:

1) A[0:k-1] = A[0:2-1] = A[0:1]
2) A[n-k:n-1] = A[4-2:4-1] = A[2:3]
3) A[0:k-1] = A[0:2-1] = A[0:1]

From this it is clear to see elements in position [1:2] are never compared/sorted. As this is the highest level in which the values will be compared (the largest size of the array at any given point), then the array is not guaranteed to be sorted.

c) T(n) = 3T(2/3n) + c, where c<=4

d) T(n) = 3T(2/3n) + 4
= 3[3T(4/9n) + 4] + 4 = 9T(4/9n) + 12 + 4
= 9[3T(8/27n) + 4] + 12 + 4 = 27T(8/17n) + 36 + 12 + 4
= 4 (1 + 3 + 3^2 ... + 3^log_3/2(n))
= 4(3^log_3/2(n+1) - 1 / 3 - 1)
= theta(3^log_3/2(n)) = theta(n^2.71)

3)

a) the algorithm would have to have 4 sections instead of 2 in which it is searching. It would check the quarter points of those 4 sections, instead of the single midpoint, then check which range the value is in to recursively call.

```
QUARTERNARY_SEARCH(A, value, low, high):
  if low <= high:
      q1 = low + ((high-low)/4)
      q2 = low + ((high-low)/2)
      q3 = low + (3*(high-low)/4)
      if A[q1] == value or A[q2] == value or A[q3] == value:
          return True
      else if value < A[q1]:
          return QUARTERNARY_SEARCH(A, value, low, q1-1)
      else if value < A[q2]:
          return QUARTERNARY_SEARCH(A, value, q1+1, q2-1)
      else if value < A[q3]:
          return QUARTERNARY_SEARCH(A, value, q2+1, q3-1)
      else:
          return QUARTERNARY_SEARCH(A, value, q3+1, high)
  return False # not found
```

b) T(n) = T(n/4) + c, where c<=11

c) T(n) = T(n/4) + c
by master theorem:
a=1, b=4 => log_4(1), f(n) = theta(1), c = 0
T(n) = O(lgn)

d) the worst case running time of quaternary search would run at the same rate as biinary search.

4)

a) the algorithm could break all values into pairs, then determine if the value is the min or max of that pair, then merge them together

```
MIN_AND_MAX(A, low, high):
    if low == high: # single el
        max = A[low]
        min = A[high]
        return (min, max)
    else if high == low+1: # pair
        if A[low] > A[high]:
            max = A[low]
            min = A[high]
        else:
            max = A[high]
            min = A[low]
        return (min, max)
    # more than pair
    mid = (low+high)/2
    min_l, min_r = MIN_AND_MAX(A, low, mid)
    max_l, max_r = MIN_AND_MAX(A, mid+1, high)
    if min_l < min_r:
        min = min_l
    else:
        min = min_r
    if max_l > max_r:
        max = max_l
    else:
        max = max_r
    return (min, max)
```

b) T(n) = 2T(n/2) + c, where c <= 18

c) by master theorem:
   a=2, b=2, f(n) = theta(1), c=0
   T(n) = O(logn)

d) The running times will both be O(logn), wiht 3n/2 - c comparisons.

5)

a) this algorithm will break the array into subarrays half the size of the original, choose which element is the majority of those halfs, then go through the array to decide if it is possible to have the element as a majority.

b)
```
MAJORITY_ELEMENT(A[1..n]):
    n = length of A
    if n == 1:
      return a[1]
    k = floor of n/2
    left = MAJORITY_ELEMENT(A[1..k])
    right = MAJORITY_ELEMENT(A[k+1..n])
```

```
if left == right:
  return left
left_count = 0
right_count = 0
for i=0 to n:
  if A[i] == left:
    left_count = left_count + 1
  if A[i] == right:
    right_count = right_count + 1
if left_count > k+1:
  return left
if right_count > k+1:
  return right
return NULL
```

c) T(n) = 2T(n/2) + n

d) by master theorem:
a=2, b=2, c= log_2(2) => 1
T(n) = O(nlogn)