Kelsey Helms
CS 325
10/3/16
Homework 2

1. **Give the asymptotic bounds for T(n) in each of the following recurrences. Make your bounds as tight as possible and justify your answers. Assume the base cases T(0)=1 and/or T(1) = 1.**
   a. $\boldsymbol{T(n) = T(n-2) + 2}$

      By the muster method, a = 1, b = 2, and f(n) = 2. By the second case, $T(n) = \Theta(n^{0+1}) = \Theta(\mathrm{n})$

   b. $\boldsymbol{T(n) = 3T(n-1) + 1}$

      By the muster method, a = 3, b = 1, and f(n) = 1. By the third case $T(n) = \Theta\left(\mathrm{n}^0 3^{\frac{n}{1}}\right) = \Theta(3^{\mathrm{n}})$

   c. $\boldsymbol{T(n) = 2T\left(\frac{n}{4}\right) + n^2}$

      By the master theorem, a = 2, b = 4, and f(n) = n^2. $n^{log_4 2} = n^{\frac{1}{2}} < n^2$. So by the third case, we must check the regularity condition:
      $2\left(\frac{n}{4}\right)^2 = \frac{n^2}{8} \leq cn^2$ when c = 1/8, so: $T(n) = \Theta(n^2)$

   d. $\boldsymbol{T(n) = 9T\left(\frac{n}{3}\right) + 6n^2}$

      By the master theorem, a = 9, b = 3, and f(n) = 6n^2. $n^{log_3 9} = n^2 == n^2$. So by the second case: $T(n) = \Theta(n^2\ log\ n)$

2. **Consider the following algorithm for sorting.**

```
STOOGESORT(A[0 ... n - 1])
    if n = 2 and A[0] > A[1]
        swap A[0] and A[1]
    else if n > 2
        k = ceiling(2n/3)
        STOOGESORT(A[0 ... k - 1])
        STOOGESORT(A[n - k ... n - 1])
        STOOGESORT(A[0 ... k - 1])
```

   a. **Explain why the STOOGESORT algorithm sorts its input. (This is not a formal proof).**

      STOOGESORT sorts its input by breaking the array down into two overlapping sections repeatedly until there are only three items. It then sorts the first two, then the last two, then the first two again to account for the smallest item being in the last cell. It then follows this pattern up through the recurrences.

b. **Would STOOGESORT still sort correctly if we replaced k = ceiling(2n/3) with m = floor(2n/3)? If yes prove if no give a counterexample. (Hint: what happens when n = 4?)**

This would not sort correctly. If n = 4 and we had an array A = [3, 5, 1, 2], then k = floor(2*4/3) = 2. The first recursive call would be STOOGESORT(A[0, 1]) and n = 2. A[0] < A[1], so 3, 5 would stay in place. Next recursive call would be STOOGESORT(A[2, 3] and n = 2. A[2] < A[3], so 1, 2 would stay in place. The first recursive call would repeat exactly the same, as nothing has been swapped. Therefore, the "sorted" array would remain A = {3, 5, 1, 2], which is incorrect.

c. **State a recurrence for the number of comparisons executed by STOOGESORT.**

$T(n) = 3T\left(\frac{2n}{3}\right) + 1$

d. **Solve the recurrence.**

By the master theorem, a = 3, b = 3/2, and f(n) = 1. $n^{log_{\left(\frac{3}{2}\right)}3} = n^{2.7} > 1$ so by the first case $T(n) = O\left(n^{log_{\left(\frac{3}{2}\right)}3}\right) = O(n^{2.7})$

3. **The quaternary search algorithm is a modification of the binary search algorithm that splits the input not into two sets of almost-equal sizes, but into four sets of sizes approximately one-fourth. Write pseudo-code for the quaternary search algorithm, give the recurrence for the quaternary search algorithm and determine the asymptotic complexity of the algorithm. Compare the worst-case running time of the quaternary search algorithm to that of the binary search algorithm.**

```
assume we are searching for some number m in an array of size n:
point1 = (n-1)/4
point2 = (n-1)/2
point3 = 3(n-1)/4
if A[point1] is m
        return A[point1]
else if A[point2] is m
        return A[point2]
else if A[point3] is m
        return A[point3]
else if m < A[point1]
        quatSearch(A[0]…A[point1])
else if m > A[point3]
```

```
        quatSearch(A[point3+1]…A[n-1])
else if m < A[point2]
        quatSearch(A[point1+1]…A[point2])
else
        quatSearch(A[point2+1…A[point3])
```

The recurrence would be $T(n) = T\left(\frac{n}{4}\right) + 3$ to search ¼ the array with 3 comparisons.

Using the master theorem, a = 1, b = 4, and f(n) = 3. $n^{log_4 1} = 1$==3, so by the second case $T(n) = \Theta \log n$

For worst case scenarios:

binary:	$1 * \frac{\ln n}{\ln 2} = 1.443 \ln n$
quaternary:	$3 * \frac{\ln n}{\ln 4} = 2.164 \ln n$

4. **Design and analyze a divide and conquer algorithm that determines the minimum and maximum value in an unsorted list (array). Write pseudo-code for the min_and_max algorithm, give the recurrence and determine the asymptotic running time of the algorithm. Compare the running time of the recursive min_and_max algorithm to that of an iterative algorithm for finding the minimum and maximum values of an array.**

```
assume we have parameters of the array, first array position j, and last array
position k:
if j = k
        max = min = A[j]
else if j = k – 1
        if A[j] < A[k]
                max = A[k], min = A[j]
        else
                max = A[j], min = A[k]
else
        mid = j + k / 2
        min_and_max(A[], j, mid, max, min)
        min_and_max(A[], mid, k, max2, min2)
        if max2 > max
                max = max2
        if min2 < min
                min = min2
```

The recurrence would be $T(n) = 2T\left(\frac{n}{2}\right) + 2$ to find the min and max.

Using the master theorem, a = 2, b = 2, and f(n) = 2. $n^{log_2 2} = n > 2$, so by the first case $T(n) = \Theta n$

For worst case scenarios:

iterative:	2n - 2
recursive:	3n/2 - 2

5. **An array A[1 . . . n] is said to have a majority element if more than half of its entries are the same. The majority element of A is any element occurring in more than n/2 positions (so if n = 6 or n = 7, the majority element will occur in at least 4 positions). Given an array, the task is to design an algorithm to determine whether the array has a majority element, and, if so, to find that element. The elements of the array are not necessarily from an ordered domain like the integers, so there can be no comparisons of the form "is A[i] > A[j]?". (Think of the array elements as GIF files, say.) Therefore you cannot sort the array. However you can answer questions of the form: "does A[i] = A[j]?" in constant time. Give a detailed verbal description and pseudo-code for a divide and conquer algorithm to find a majority element in A (or determine that no majority element exists). Give a recurrence for the algorithm and determine the asymptotic running time.**
Note: A O(n) algorithm exists but your algorithm only needs to be O(nlgn).

The algorithm will take a parameter of the array. It has a base case of one element, which it will return. The non-base case will split the array in two and recursively call the function on each half. On the way back up, the element returned from one half will be compared to the element returned from the other half. It will return the first element if they are equivalent. If they are not equivalent, the amount of occurrences of the two element will be counted. If one of the elements accounts for more than half of the sub-array, the element is returned. Otherwise, there is no majority element. This algorithm works because in order to account for more than half the positions, an element will have to account for more than half the positions in at least one half.

```
assume we have parameters of the array A[]:
if n = 1
        return A[n]
mid = n/2
elem1 = majorElem(A[0]…A[mid])
elem2 = majorElem(A[mid+1]…A[n])
if elem1 == elem2
```

```
        return elem1
for j = 0 to mid
        if A[j] == elem1
                count1++
for k = mid+1 to n
        if A[k] == elem2
                count2++
if count1 > mid+1
        return elem1
else if count2 > mid+2
        return elem2
else return NONE
```

The recurrence would be $T(n) = 2T\left(\frac{n}{2}\right) + n$ to find the majority element.

Using the master theorem, a = 2, b = 2, and f(n) = n. $n^{log_2 2} = n == n$, so by the second case $T(n) = \Theta \mathrm{n} \log n$