Colleen Minor
HW 2

Problem 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.

Simple…. if d >

a) $T(n)=T(n-2)+2$

The problem is in the form of
T(n) = a*T(n-b) + f(n) with a, b > 0 and d=> 0

so it can be solved with the Muster Theorem:
a = 1, b = 2, f(n) = Θ(1), d = 0

Because a = 1, and f(n) = Θ(n)^0,

T(n) = Θ(n)^(0+1) = Θ(n)

b) $T(n)=3T(n-1)+1$

T(n) = 3T(n - 1) + 1

= 3[3T(n - 2) + 1] + 1 = 32T(n - 2) + 3 + 1

= 3[32T(n - 3) + 3 + 1] + 1 = 33T(n - 3) + 32 + 3 + 1

= 3[33T(n - 4) + 33 + 3 + 1] + 1 = 34T(n - 4) + 33 + 32 + 3 + 1

Stop when (n - 1) = 1 or 0, for base case T(1) or T(0) = 1

You can see the geometric series formed above, which indicates 3kT(1) + 1 + 3 + 32 + ... + 3k-1

So, the asymptotic runtime is T(n) = θ(3n)

c) $T(n)=2T(n/4)+n^2;$

This function is in the form:

T(n) = a * T(n/b)+ f(n) where a >= 1 and b > 1

So we can use the master theorem (page 95 in the text) to solve it.

Step 1: Get, a, b, and f(n).
In this problem, a = 2, b = 4, and f(n) = n^2.

Step 2: Get c.

If we put f(n) in the form n^c, what is c? Answer: c = 2.

Step 3: Get log b(a)
That's log 4(2) = 0.5.

Step 4: Compare log b(a) to c.
0.5 < 2. That's a sign that case 3 likely applies, though there are some other conditions that also need to be true in order for it to work….

Step 4a: Test that a*f(n/b) <= c*f(n) for some constant c < 1 and all sufficiently large n. I thought that we were supposed to drop coefficients to determine which function grows faster. So doing that, yes I think it can be said that f(n/b) will not outgrow f(n), so that condition is satisfied.

Step 5: Get the answer.
Since this is a case 3, that means that T(n) = θ(f(n)). In this case f(n) is n^2, and so, the solution is

θ(n^2).

d) $T(n)$=9$T(n$/3)+6n^2

For this recurrence, we can use the master theorem. We have a = 9, b = 3, f(n) = 6n^2.

Then we look to put f(n) in the form n^c. We see that c = 2.

Then we'll compare c to logb(a):
logb(a)= log3(9) = 2.

c = logb(a) (because 2 = 2), which means we apply case 2 on page 95 in the text. This means that T(n) = Theta(n^(logb(a)) * lg n)

Plugging in our numbers,
T(n) = θ(n^2 * lg n)

2)

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

The stoogesort first compares the value at the start of the list to the value of the end of the list and swaps them if the end value is smaller. If there are 3 or more elements in the list, it then stoogesort's the first 2/3 of the array, then the second 2/3, and then the first 2/3 again.

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?)

No, it would not still sort correctly if we replaced ceiling with floor. This is because by using the floor, we eliminate the overlap between the sections being processed, which is what allows the algorithm to work.

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

T(n) = 3T(2n / 3) + O(1)

d) Solve the recurrence.
We can use the master theorem.
Get a, b, and f(n): a = 3, b = 1.5, f(n) = n^0
2. Get c: That's 0.
3. Get log b(a). log 1.5 (3) = 2.709…
4. Compare log b (a) with c. c < 2.709.
5. Get the solution. This is clearly a case 1 scenario, which means that T(n) = θ(n^(log b(a)) = θ(n^2.709).

3) **Problem 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.

The array that would be sent in would have values increasing in order.

```
function quantenarySearch(array, valToFind, lowIndex, highIndex)

        int q1 = (lowIndex + highIndex)/4;
        int q2 = q1 * 2;
        int q3 = q1 * 3;
        int q4 = highIndex;

        if valToFind in {array[q1], array[q2], array[q3], array[q4]}
                return whichever index is valToFind

        else if q1 > valToFind
                return quantenarySearch(array, valToFind, lowIndex, q1)

        else if q1 < valToFind < q2
                return quantenarySearch(array, valToFind, q1, q2)

        else if q2 < valToFind < q3
                return quantenarySearch(array, valToFind, q2, q3)

        else if q3 < valToFind < q4
```

```
            return quantenarySearch(array, valToFind, q3, q4)

        else
            return -1 //valToFind not in array
```

Recurrence relation: T(n) = T(n/4) + 1

Asymptotic complexity:
Using the master theorum,

a = 1, b = 4, f(n) = n^0

loga(b) = log1(4) = 0.

Case 2 applies:
f(n) = Θ(n^0 * lg n) = Θ(1 * lg n)
f(n) = Θ(lg n)

4) **Problem 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.

Pseudocode:

```
min_and_max(array, begin, end, min, max)
        if(begin == end)
            min, max = array[begin]

        else
            mid = (begin + end)/2

            leftMin, leftMax, rightMin, rightMax = 0;

            min_and_max(array, begin, mid, leftMin, leftMax)
            min_and_max(array, mid+1, end, rightMin, rightMax)


            max = bigger(leftMax, rightMax)
            min = smaller(leftMin, rightMin)
```

//end of algorithm

Run this algorithm live in C++ here:
http://cpp.sh/9n2bo

Recurrence relation: T(n) = 2T(n/2) + 1

Using the master theorem:
a = 2, b = 2, f(n) = n^0

log2(2) = 1

const < loga(b). Case 1 applies.

T(n) = Θ(n^log2(2)) = Θ(n^1) = Θ(n)

T(n) = Θ(n)

Comparison to iterative algorithm:

Here is iterative min and max finder I made last week:

```
var maxMinNumFinder= function(arr){
   var max = arr[0];
   var min = arr[0];
   for(var i = 1; i < arrayLength; i++){
     if(arr[i] > max){
        max = arr[i];
     }
    if(arr[i] < min){
        min = arr[i];
     }


   }
   return [max, min];
};
```

The above loop will make exactly 2n comparisons— one for each number, compared for both greater-than and less-than in separate if statements. Because the coefficient 2 makes little difference when n is very large, we drop it, making the worst-case for this one O(n) .

The iterative algorithm also has a run time of Θ(n).

5) **Problem 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 pseudocode 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).

Description: If an element m is the majority of a sublist, and a sublist is divided into two halfs a and b, m must be the majority of sublist a or b. After all if m were only half of both a and b, then it would only be half of the two sublists combined. So logically we can just divide the array into two halves, look for the a majority in both arrays, and if neither has a majority output -1 (no majority found), if 2) both have the same majority return that number, and if 3) either only one has a majority or the two have different majorities, compare the majority to each element of the array to see if the element appears more than n/2 times.

pseudocode:

```
getMajority(array[1…n])
        if(n == 1)
                return array[n]
        k = n/2
        leftMajority = getMajority(array[1…k])
        rightMajority = getMajority(array[k+1…n])
        if(leftMajority == rightMajority)
                return leftMajority
        leftCount = getCount(a[1…n], leftMajority)
        rightCount = getCount(a[1…b], rightMajority)
        if leftCount > (n/2 + 1)
                return leftMajority
        if rightCount > (n/2 + 1)
                return rightMajority
else
        return -1 //no majority
//End of aglorithm
```

Recurrence relation: $T(n) = 2T(n/2) + O(n)$

Using the master theorem,
a = 2

b = 2
c = 1

Since c = logb(a), case 2 applies.

This gives the running time O(n lg n).