Richard Moot
CS 325
July 3, 2016

Homework 2

Problem 1:

a. $\boldsymbol{T(n) = T(n\text{-}2) + 4}$

Assume: $T(2) = 4$ and T(1) = 0

$$T(3) = T(1) + 4 = 0 + 4 = 4$$
$$T(4) = T(2) + 4 = 4 + 4 = 8$$
$$T(5) = T(3) + 4 = 4 + 4 = 8$$
$$T(6) = T(4) + 4 = 8 + 4 = 12$$
$$T(7) = T(5) + 4 = 8 + 4 = 12$$
$$T(8) = T(6) + 4 = 12 + 4 = 16$$
$$T(9) = T(7) + 4 = 12 + 4 = 16$$

We can see that the recurrence grows at a linear rate since it increases by 4 every 2 instances, which would give us an asymptotic boundary of O(n).

Even using the Muster Theorem (wasn't sure if this is official/real), we can see a = 1 and d = 0 giving us a time complexity of O(n).

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

Again here we can utilize the Muster Theorem with a = 3, b = -1, and d = 0. This appears to give us the boundary of $\Theta(3^n)$ since $T(n) = O(n^d a^{n/b})$ translates to $O(n^0 a^{n/1})$. We have n^0 drop out since this is just 1 and are left with 3^n since we can also drop out the 1 since it doesn't influence the value.

c. $\boldsymbol{T(n) = 2T\left(\frac{n}{8}\right) + 4n^2}$

We can actually use the Master Theorem here to determine the asymptotic boundaries of this recurrence. We have a = 2, b = 8, and d = 2. We can see that $\log_8 2 < d$ so we have case 1 of the Master Theorem. This would give us $O(n^d)$ which is $O(n^2)$ since d = 2.

Problem 2:

- *Algorithm A* solves problems by dividing them into five sub-problems of half the size, recursively solving each sub-problem, and then combining the solutions in linear time

$$T(n) = 5T\left(\frac{n}{2}\right) + n$$

$O(n^{log_2 5})$ is the run time according to the Master Theorem which is falls between powers 2 and 3.

- *Algorithm B* solves problems of size n by recursively solving two sub-problems of size $(n - 1)$ and then combining the solutions in constant time.

$$T(n) = 2T(n-1) + 1$$

$O(2^n)$ is the runtime using the Muster Theorem.

- *Algorithm C* solves problems of size n by dividing them into nine sub-problems of size $\frac{n}{3}$, recursively solving each sub-problem, and then combining solutions in $\Theta(n^2)$ time.

$$T(n) = 9T\left(\frac{n}{3}\right) + \Theta(n^2)$$

$O(n^2 \log n)$ is the runtime according to the Master Theorem.

What are the running times of each of these algorithms and which would you choose?

I would ultimately go for Algorithm A since Algorithm C would ultimately grow at a faster rate since it's additional log n will cause it to grow faster compared to the slightly higher power of 2 that Algorithm A has. Algorithm B should be obviously dismissed since it is exponential time.

Problem 3:

```
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).

The STOOGESORT works because it sorts the first two thirds of an array, then the last two thirds of the array, followed by the first two thirds sorted again. This allows the algorithm to successfully cover all elements to ensure that they would be sorted at the extremes.

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?)

STOOGESORT would not work correctly replacing ceil() with floor(), since some values of n would cause there to be gaps in the sorting of values. The hint even demonstrates that for 8/3 rounded down would give only 2. This would sort the first half and the second half, but your data set could have extremes that would never be checked from end points of the array.

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.

Using the Master Theorem, we can see that it is of case 3 and can give us $O(n^{log_{\frac{3}{2}}3})$ since a = 3, b = 2/3, and d = 0.

Problem 4:

$$T(n) = T\left(\frac{n}{4}\right) + 1$$

Using the Master Theorem, we have a = 1, b = 4, and d = 0. This would give us 0 = $\log_4 1$ which is case 2 of the Master Theorem. Ultimately this would be O(n^0 log (n)). We can disregard the n^0 and see the run time is O($\log_4$ n).

Assume a sorted array has been given.

```
QUARTERNARY ( A[1…n], start, end, x )
        first = start + end/4
        left = left + end/4
        right = mid + end/4
        last = right + end/4

        if (r ≥ 1)
                if A[first] = x
                        return x
                if A[left] = x
                        return x
                if A[right] = x
                        return x
                if A[last] = x
                        return x

                if A[first] < x < A[left]
                        QUARTERNARY(A, start, first, x)
                if A[left] < x < A[right]
                        QUARTERNARY(A, left, left, x)
                if A[right] < x < A[last]
                        QUARTERNARY(A, right, last, x)

                return QUARTERNARY(A, last, end, x)
        else
                return -1
```

Problem 5:

```
MIN_MAX(A[1…n], min, max, size)
    if size = 1
        min = max = A[0]
        return min max
    else if size = 2
        if A[1] > A][2]
            max = A[1]
            min = A[2]
            return min max
        else
            max = A[1]
            min = A[2]
            return min max
    else // size greater than 2
        left = MIN_MAX(a[1…n/2], min, max, size/2)
        right = MIN_MAX(a[n/2+1…n], min, max, size/2)

        if left min > right min
            min = right min
        else
            min = left min
        if left max > right max
            max = left max
        else
            max = right max
        return min max
```

The recurrence of this algorithm would be $T(n) = 2(\frac{n}{2}) + 2$ with a running time of O(n) according to the Master Theorem which has this as case 3 but since $\log_2 2$ is 1, it gives us linear time. An iterative algorithm would be comparable since you would still require scanning the entire array in the worst case to find the minimum and maximum values and require linear time.

Problem 6:

```
MAJORITY(A[1…n])
        if n = 1
                return A[1]

        left = MAJORITY(A[n…n/2])
        right = MAJORITY(A[n/2+1…n])

        if (left == null && right == null)
                return null
        else if (left == null && right != null)
                return right
        else if (left != null && right == null)
                return left
        else if (left != right)
                return null
        else
                return left
```

This algorithm works be first continually dividing the elements in half until it reaches individual elements. Then as it starts to return, it will only return a majority element from its subarray if one exists.
There are several cases to consider as we start returning out towards the base case. Either we can have no majority in either side, in which case we return null. We can also have left be null but right contains a majority, so we would want to return right. Same goes for if left has a majority by right does not. We then want to check if left and right both returned a majority and if those majorities are the same. If they aren't we want to return null, since that means that we do not actually have a majority overall (it would have to be the majority in both to be true). Finally, we want to just return left if we have our majority since left and right were the same.

Our recurrence is $T(n) = 2T(\frac{n}{2}) + 2n$ for this algorithm. Using Case 2 of the Master Theorem we can see that the runtime of this algorithm is Θ (n log n).