Name: Alexander Miranda
Date: October 4, 2016
Assignment: 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) $T(n) = T(n-2) + 2$

(Using Muster Method)

- $T(n) = a * T(n-b) + f(n)$
- $a = 1, b = 2$ and because $f(n) = \theta(n^0) = 2, d = 0$
- Because $a = 1$ the recurrence bound is $\theta(n^{d+1})$
- $\theta(n^{d+1}) = \theta(n^{0+1})$
- Therefore $T(n) = \theta(n)$

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

(Using Muster Method)

- $T(n) = a * T(n-b) + f(n)$
- $a = 3, b = 1$ and because $f(n) = \theta(n^0) = 1, d = 0$
- Because $a > 1$ the recurrence bound is $\theta(n^d * a^{\frac{n}{b}})$
- $\theta(n^d * a^{\frac{n}{b}}) = \theta(n^0 * 3^{\frac{n}{1}})$
- Therefore, $T(n) = \theta(3^n)$

c) $T(n) = 2T(\frac{n}{4}) + n^2$

(Using Master Theorem)

- $a = 2, b = 4, f(n) = n^2$
- $n^{log_b(a)} = n^{log_4(2)} = n^{\frac{1}{2}}$
- $f(n) = O(n^{log_4(2)+\epsilon}) = O(n^{\frac{1}{2}+\epsilon})$
- $\frac{1}{2} + \epsilon = 2$ (Case 3 applies)
- $\epsilon = 2 - \frac{1}{2} = 1.5$
- $a * f(\frac{n}{b}) \leq c * f(n)$ (Regularity condition)
- $2 * (\frac{n}{4})^2 \leq c * n^2$
- $2 * \frac{n^2}{16} \leq c * n^2$
- $\frac{n^2}{8} \leq c * n^2$
- $\frac{1}{8} \leq c$ Therefore c can be $c < 1$ so the regularity condition holds
- $1 > c$ (Regularity condition holds)
- Therefore $T(n) = \theta(f(n)) = \theta(n^2)$

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

(Using Master Theorem)

- $a = 9, b = 3, f(n) = 6n^2$
- $n^{log_b(a)} = n^{log_3(9)} = n^2$
- $f(n) = 6n^2 = \theta(n^2)$
- $f(n) = \theta(n^{log_3(9)}) = \theta(n^2)$ (Therefore case 2 applies)
- Therefore $T(n) = \theta(n^{log_b(a)} * log_2(n)) = \theta(n^{log_3(9)} * log_2(n))$
- $T(n) = \theta(n^2 * log_2(n))$

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)

The method will do nothing if the passed in array is empty or only contains one element. (Trivial case).

For when $n = 2$ (base case) `STOOGESORT(A[0, 1])` will properly sort the array by swapping the two elements if the first element is greater than the second, otherwise the array will be unchanged because it is already sorted.

However when the length of the array is greater than 2 e.g. ($n \geq 3$) an integer k is calculated which is the ceiling of the $\frac{2}{3}$ of length n. Then there are the three recursive calls to STOOGESORT where the passed in arrays for each call are as follows:

- First subarray is the first $\frac{2}{3}$ of the total array which is then sorted in the first call (and subsequent recursive calls if necessary)
- Second subarray is the last $\frac{2}{3}$ of the total array which is then sorted in the second call (and subsequent recursive calls if necessary)
- The third recursive call sorts the first $\frac{2}{3}$ of the total array again like the first call before it. (and subsequent recursive calls if necessary)

This algorithm sorts the total array because it passes sub-arrays that fully encompass the array that was originally passed in. The method is inefficient however because it overlaps in the sub-arrays that it sorts in the sense that there will be overlap between the first $\frac{2}{3}$ and the second $\frac{2}{3}$ of the array when sorting.

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. A good counter example would be if we passed the following array into STOOGESORT:

- $[23, 12, 5, 67]$

Now I will step through the function with the given input to show that it would not sort correctly when using $m = floor(\frac{2n}{3})$:

- First we check if the length of the passed in array is of length 2 ($n = 2$) which it is not so we go to the else if block and see that that is in fact true because $n = 4 > 2$
- We now calculate the variable m to be equal to $m = floor(\frac{2*4}{3}) = 2$
- The first recursive call of STOOGESORT will be passed in the sub-array of $[23, 12]$ which will satisfy the $n = 2$ condition which will swap the two elements so the total array is now: $[12, 23, 5, 67]$.
- The second recursive call of STOOGESORT will be passed in the sub-array of $[5, 67]$ which will satisfy the $n = 2$ condition which will not do anything to the sub-array because $A[0] < A[1]$ already, leaving the total array unchanged also.
- The third and final recursive call of STOOGESORT will be passed in the sub-array of $[12, 23]$ which will satisfy the $n = 2$ condition again but nothing will be done because they were sorted in the first recursive call.
- Therefore the resulting array at the end of all of the calls to STOOGESORT will be:
- $[12, 23, 5, 67]$ which is not a complete sort therefore showing if the algorithm changes to use $m = floor(\frac{2n}{3})$ it will not correctly sort at least in the case shown above which is my counter-example.

By using floor when the array is of length 4 (and possibly other lengths) The two sub-arrays do not overlap and as a result only will sort the two sub-arrays (in the $n = 4$ case) (first half and second half with no overlap) locally and not globally.

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

- $T(n) = a * T(\frac{n}{b}) + f(n)$ Whereas $a = 3$ because of 3 recursive calls and $b = \frac{2}{3}$ because the input n is sub-divided into $\frac{2}{3}$ sub-problems therefore:
- $T(n) = 3 * T(\frac{2n}{3}) + \theta(1)$

d) Solve the recurrence.

Based on the recurrence determined in part c we can use the Master Method to solve the recurrence:

- $a = 3, b = \frac{3}{2}, f(n) = c$
- $n^{log_b(a)} = n^{log_{\frac{3}{2}}(3)}$
- $log_{\frac{3}{2}}(3) = \frac{log_{10}(3)}{log_{10}(\frac{3}{2})} \approx 2.71$
- $n^{log_{\frac{3}{2}}(3)} = n^{2.71} = \Omega(f(n))$ Because $f(n) = c$ where c is some constant
- Therefore it follows: $f(n) = O(n^{2.71-\epsilon})$ where the $\epsilon \approx 2.71$
- Therefore case 1 applies and the runtime for the recurrence is $T(n) = \theta(n^{2.71})$

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.

```
def quaternary_search(arr, el):
    start = 0
    first_quad = int(len(arr) / 4) - 1
    mid_point = int(len(arr) / 2) - 1
    third_quad= int(len(arr) * (3 / 4)) - 1
    end = len(arr) - 1

    list_of_quads = [start, first_quad, mid_point, third_quad, end]

    try:
        index = [arr[i] for i in list_of_quads].index(el)
        return list_of_quads[index]
    except:
        pass

    if (arr[start] < el < arr[first_quad]):
        return quadternary_search(arr[start:first_quad], el)
    elif (arr[first_quad] < el < arr[mid_point]):
        return quadternary_search(arr[first_quad:mid_point], el)
    elif (arr[mid_point] < el < arr[third_quad]):
        return quadternary_search(arr[mid_point:third_quad], el)
    elif (arr[third_quad] < el < arr[end]):
        return quadternary_search(arr[third_quad:end], el)
    else:
        return -1
```

The recurrence for quaternary search is as follows:

- $T(n) = T(\frac{n}{4}) + c$

The reason for this is because only a single recursive call is made in the worst case and the searching for the element in the list through comparisons which is a constant time operation (c) where c represents the number of comparisons made. The b factor is 4 because the search sub-divides the passed in array into quarter sized sub-problems rather than the halves encountered with binary search. Based on the above recurrence we can solve it to show the runtime of quaternary search:

(Using the master method)

- $a = 1, b = 4, f(n) = c$
- $n^{log_b(a)} = n^{log_4(1)} = n^0$
- $n^0 = 1 = \theta(1) = \theta(c)$ (Therefore case 2 applies)
- Therefore $T(n) = \theta(n^{log_b(a)} * log_4(n)) = \theta(1 * log_4(n))$
- The runtime for quaternary search is $T(n) = \theta(log_4(n))$

The worst case runtime for the quaternary search is fundamentally equivalent to the worst case runtime for binary search because the runtime of binary search is $T(n) = \theta(log_2(n))$. And we have shown previously that the runtime of logirithms of different bases are equivalent when it comes to asymptotic analysis.
$\theta(log_2(n)) = \theta(log_4(n))$

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.

Pseudocode for max_min method which will return the (min, max) pair:

```
max_min(arr[0..n]) {
    if (n == 1) {
        return (arr[0], arr[0])
    } else if (n == 2) {
        if (arr[0] < arr[1]) {
            return (arr[0], arr[1])
        } else {
            return (arr[1], arr[0])
        }
    } else {
        (min_left, max_left) = max_min(arr[0..(n/2)])
        (min_right, max_right) = max_min([(n/2 + 1)..n])

        if (max_left < max_right) {
            max = max_right
        } else {
            max = max_left
        }

        if (min_left < min_right) {
            min = min_left
        } else {
            min = min_right
        }

        return (min, max)
    }
}
```

Determine the recurrence:

When the array only has one term no comparisons are made therefore $T(1) = 0$ and when there are only two elements only one comparison is made therefore $T(2) = 1$.

For the case of $n > 2$:

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

The reason being from the above pseudocode showing that there are two recursive calls of min_max where each call is passed in a respective half of the total array, which are then used to then make two comparisons to each others' local mins and maxs to determine the overall (global) min and max which is returned.

Now I will determine the runtime using the Master Method:

- $a = 2, b = 2, f(n) = 2$
- $n^{log_b(a)} = n^{log_2(2)} = n^1$
- $f(n) = 2$ which follows that $f(n) = n^0$ (Therefore case one applies)
- $f(n) = O(n^{log_b(a)-\epsilon})$ where $\epsilon = 1$
- Therefore it follows $T(n) = \theta(n^{log_b(a)})$
- $\theta(n^{log_2(2)}) = \theta(n^1) = \theta(n)$

An example of an iterative method to find the min and max of an array can be shown below:

```
max_min(arr[0..n]) {
    if (n == 1) {
        return (arr[0], arr[0])
    } else if (n == 2) {
        if (arr[0] > arr[1]) {
            return (arr[0], arr[1])
        } else {
            return (arr[1], arr[0])
        }
    } else {
        minimum = MAX_INT
        maximum = MIN_INT

        for (i in arr) {
            if (i > maximum) {
                maximum = i
            }

            if (i < minimum) {
                minimum = i
            }
        }

        return (minimum, maximum)
    }
}
```

The iterative example has a runtime of $T(n) = \theta(n) + 2$ because the iterative solution at worst has to traverse the length of the passed in array and make two comparisons of the elements traversed to the currently held max and min values respectively. For values of significantly large n we can drop the constant term 2 and take the runtime to be that of $\theta(n)$ which is equivalent to the recursive implementation shown above.

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).

The algorithm starts by dividing the array in half and calling the method on each respective half. This is related to the stragedy of the merge sort algorithm. When the arrays passed in become single element those single elements will be returned as the "majority" of their single element arrays. The other levels will return elements as a result of the two recursive calls. The crux of the algorithm is that if there exists a majority element in the array then that element will be a majority element for either/both of the left or right sub-arrays. There are four possible outcomes from the algorithm:

- Both function calls do not return a majority element and as a result the array passed in does not have a majority element to output
- The right side is a majority but the left side isn't. In this case the recursive call on the right side returns an element that is a majority for the right side, so the next step is to search the complete array for how many times this element occurs and if it is the majority element for the total array than return that element.
- This situation would be similar to the one directly above but it would be the case that the left side returned a majority and the right did not.
- Both calls on the left and right sub-arrays return majority elements. So there needs to be a count of both elements in the combined array to determine if either one is in fact a majority element of the total array, if so then it would be returned, otherwise no element would be returned.

The overall function will either return a majority element if found or nothing if a majority element does not exist.

Pseudo-code in Python:

```
def get_frequency(arr, el):
    count = 0

    for i in arr:
        if i == el:
            count += 1

    return count

def find_majority_el(arr):
    if len(arr) == 1:
        return arr[0]
    k = math.floor(len(arr) / 2)
    left_majority = find_majority_el(arr[0:k])
    right_majority = find_majority_el(arr[k:])

    if left_majority == right_majority:
        return left_majority

    lcount = get_frequency(arr, left_majority)
    rcount = get_frequency(arr, right_majority)

    if lcount > k:
        return left_majority
    elif rcount > k:
        return right_majority
    else:
        return None
```

When finding the recurrence we analyze the running time of the algorithm:

There are two recursive calls made in the algorithm with each of those calls being passed half of the original array. The non-recursive costs are when we have to compare each number at most twice (twice in the fourth scenario). So the non-recursive costs would be upper bounded at $2n$ comparisons where n is the number of elements in the original array. Therefore the recurrence can be written as follows:

- $T(1) = 0$
- $T(n) = 2 * T(\frac{n}{2}) + 2n$

Now to find the running time we can use the Master Method:

- $a = 2, b = 2, f(n) = 2n$
- $n^{log_b(a)} = n^{log_2(2)} = n^1 = \theta(n)$
- $f(n) = 2n = \theta(n)$ (Therefore case 2 applies)
- Therefore $T(n) = \theta(n^{log_b(a)} * log_2(n)) = \theta(n * log_2(n))$

Thus showing that the implementation shown above satisfies the requirement of having a runtime of