Jonathan Nicolosi

HW2

1.

a. $T(n) = T(n-2) + 4$

$T(n) = T((n-2)-2)+4)+4 = T(n-4) + 8$

$T(n) = T(n-6) + 12$

$T(n) = T(n-8) + 16$

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

Replace k with (n-2)

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

$T(n) = T(2) + (2n-4)$

$\underline{T(n) = \Theta f(n)}$

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

Or $\mathbf{3^1T(n-1) + 3^1}$

$T(n) = 3[3T(n-2)+3] + 3$

$T(n) = [9T(n-2)] + 9 + 3$

$T(n) = [9T(n-2)] + 12$

Or $\mathbf{3^2T(n-2) + 3^1 + 3^2}$

$T(n) = 3[9T(n-3) + 12] + 3$

$T(n) = 27T(n-3) + 39$

Or

$T(n) = \mathbf{3^3T(n-3) + 3^1 + 3^2 + 3^3}$

$\underline{T(n) = \Theta f(3^n)}$

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

By Master Theorem:

$a = 2, b = 8, c = 2$

When $\log_b a < c$, $T(n) = \Theta(n^c)$

Therefore, $T(n) = \Theta(n^2)$

2.

a. $T(n) = 5T(n/2) + O(n)$

$a = 5$, $b = 2$, $d = 1$

$d < \log(a) = 1 < \log_2(5)$

So $\underline{O(n^{\log_2 5}) = O(n^{2.3219})}$

b. $T(n) = 2T(n-1) + O(1)$

$T(2) = 2T(1)$

$T(3) = 2T(2) = 2*2*T(1)$

$T(5) = 2*T(4) = 2*2*2*2*T(1)$

So $\underline{O(2^n)}$

c. $T(n) = 9T(n/3) + O(n^2)$

$a = 9$, $b = 3$, $d = 2$

$d = \log_2 a \rightarrow 2 = \log_3 9$

So $O(n^2\log(n))$

I would choose Algorithm A because it grows the slowest.

3.

a. We know STOOGESORT sorts its input because we can see that there is a swap based on a comparison between A[0] and A[1].

b. STOOGESORT would not sort correctly if we replaced k = ceiling(2n/3) with m = floor(2n/3). Depending on what value of n you have, you might divide the array too many times.

c. Recurrence for STOOGESORT:

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

d. $T(n) = 1 + 3T(\frac{2}{3}n)$

$T(n) = 1 + 3 + 9T(\frac{4}{9}n)$

$T(n) = 1 + 3 + 3^2 + \ldots + 3^{\log_{\frac{3}{2}} n}$

$T(n) = \frac{3^{\log_{\frac{3}{2}} n+1} - 1}{3-1}$

$T(n) = \Theta(3^{\log_{\frac{3}{2}} n})$

$T(n) = \Theta(3^{(\log_3 n)/(\log_3 \frac{3}{2})})$

$T(n) = \Theta(n^{1/(\log_3 \frac{3}{2})})$

$T(n) = \Theta(n^{2.71})$

4.

a. QuaternarySearch(array[], int startOfArray, int endOfArray, searchValue){

```
if( r ≥ 0 ){
        //calculate where each quarter begins and ends

        int firstQuarter = startOfArray + (endofArray – startOfArray)/4

        int secondQuarter = firstQuarter + (endofArray – startOfArray)/4

        int thirdQuarter = secondQuarter + (endofArray – startOfArray)/4

        if(array[firstQuarter] == searchValue)

                return firstQuarter; //value is in the first quarter

        if(array[secondQuarter] == searchValue)

                return secondQuarter; //value is in the second quarter

        if(array[thirdQuarter] == searchValue)

                return thirdQuarter; //value is in the third quarter

        if(array[firstQuarter] > x)

                return QuaternarySearch(array, startOfArray, firstQuarter-1, searchValue);

                //recurse backward through first quarter by returning the next lowest value

        if(array[secondQuarter] > x)

                return QuaternarySearch(array, firstQuarter, secondQuarter-1, searchValue);

                //recurse backward through second quarter by returning the next lowest
        value

        if(array[thirdQuarter] > x)

                return QuaternarySearch(array, secondQuarter, thirdQuarter-1, searchValue);

                //recurse backward through third quarter by returning the next lowest value

        return QuaternarySearch(arr, thirdQuarter, endOfArray, searchValue);

        //recurse through the fourth quarter by returning the next lowest value

}

return -1; //if value is not found
```

b. Recurrence for quaternary search:
   $T(n) = T(n/4) + 8$
c. In binary search, there are $2\text{Log}_2 n + 1$ comparisons in worst case. In quaternary search, there are $8\log_4 n + 3$ comparisons in worst case.
d. Using the Master Theorem, a = 8, b = 4 and f(n) = 8
   $T_4(n) = \Theta(\log(n))$

5.

a. Pair MaxMin(array, array_size)
   if array_size = 1
       return element as both max and min
   else if arry_size = 2
       one comparison to determine max and min
       return that pair
   else /* array_size > 2 */
       recurse for max and min of left half
       recurse for max and min of right half
       one comparison determines true max of the two candidates
       one comparison determines true min of the two candidates
       return the pair of max and min
b. $T(n) = 2T(n/2) + 2$
   Time complexity is O(n)
c. Iterative solution is also O(n)

6.

Solving the problem in O(n log n) time.

Suppose we divide array A into two halves, A_L and A_R.

Then: A has a majority element x $\Longleftrightarrow$ $\Rightarrow$ x appears more than n/2 times in A

$\Rightarrow$x appears more than n/4 times in either A_L or A_R (or both)

$\Longleftrightarrow$ x is a majority element of either A_L or A_R (or both)

This suggests a divide-and-conquer algorithm:

function majority (A[1 . . . n]){

if n = 1: return A[1]

```
        let A_L, A_R be the first and second halves of A
        M_L = majority(A_L) and M_R = majority(A_R)
        if M_L is a majority element of A:
            return M_L
        if M_R is a majority element of A:
            return M_R
        return ''no majority''
}
```

Running time: $T(n) = 2T(n/2) + O(n) = O(n \log n)$.