Tyler Cope

Analysis of Algorithms Homework 2

Question 1

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

   Using the muster method, we see that a = 1, b = 2, d = 0. Since a = 1 we're bounded by $\Theta(n^{d+1}) = \Theta(n)$

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

   We can use the muster method for this as well. a = 3, b = 1, d = 0. Since a > 1 it's the case where $\Theta(n^d a^{n/b}) = \Theta(n^0 3^n) = \Theta(3^n)$

c. $T(n)=2T(n/8)+4n^2$

   Now we can use the master method. a = 2, b = 8, $n^{\log_b a} = n^{.33}$, $f(n) = 4n^2$. Case 3 applies. $\Theta(n^2)$.

Question 2

It first swaps two elements if there are only two and the first is greater than the second. If there are more than two elements, it breaks the array into 2/3 and sorts those (first 2/3 and last 2/3). The last call sorts again. It works because the entire array is being sorted: the first call deals with the beginning, the second call the end. The entire array will be divided until it reaches the base case and swaps the two elements.

However, it wouldn't work with k = floor(2n/3). Consider an array with [34, 14, 49, 12]. In this case n is 4 so we know we'll have to go past the base case. If the array is of length 4 then

the k = 3 (floor of 2n/3). Walking through the algorithm we see that it will break up 34 and 14, effectively swapping those. Then it will swap 49 and 12. Finally it will go from 0-1, which is just the first subarray again. The final subarray will be [14, 34, 12, 49] which is not sorted.

A recurrence could be T(n) = 3T(2/3n) + 1. We make 3 recursive calls so that's a, 2/3 is the split so that's b. f(n) is 1 because the swapping of two elements is constant. To solve the recurrence, we have it in the perfect form for the master method. Since a = 3, b = 2/3, we know that we'll have $n^{\log_{2/3}3}$ which is roughly 2.7. Comparing with f(n), which is a constant in this case, we know that it's larger so our answer is $\Theta(n^{2.7})$.

Question 3

First we break the array into quarters:

beginning = 0

first = arraylength / 4 – 1

second = arraylength / 2 – 1

third = arraylength x ¾ - 1

end = arraylength – 1

Then we search for the element in each quarter of the array. If the element is in a quadrant, it will be greater than the lower number and smaller than the second number. Then we just keep breaking the array into pieces using recursion until the search element is found:

if array[element] > beginning and array[element] < first:

return quaternary(array[beginning:first], searchElement)

We do this for each piece of the algorithm. Compare the search element to a range and start the recursion in that spot once we find the range. Return 0 if it isn't found (similar to C-style Boolean values). We could also make it a Boolean function and return True or False.

The recurrence would be T(n) = T(n/4) + 1. We're making 4 recursive calls and the comparisons are a constant time function. This can also be solved using the master method. a = 1, b = 4, f(n) = 1, $n^{lob_b{}^a} = n^0 = 1$. Well comparing f(n) and $n^{lob_b{}^a}$ we see that they are both equal to a constant. Therefore, we have case 2. The answer is Θ(lgn).

The worst-case for each algorithm is that the element to be found is either at the very beginning or the very end. From lecture, we know that the solution the recurrence of binary search is Θ(lgn). Thus, the worst-case running time is the same for both algorithms.

Question 4

First we take care of the base cases:

```
If arraylength = 1:
        min = arr[0], max = arr[0]
if arraylenght = 2:
        if arr[0] < arr[1]:
                min = arr[0], max = arr[1]
        else:
```

```
min = arr[1], max = arr[0]
```

Since we want to divide and conquer, just split the array in two, get the max and the min for each sub-array, then compare them and set the correct max and min.

```
minLeft, maxLeft = findMinMax(arr[0:n/2])

minRight, maxRight = findMinMax([(n/2) + 1:n – 1])

if minLeft < minRight:

    min = minLeft

else:

    min = minRight

if maxLeft > maxRight:

    max = maxLeft

else:

    max = maxRight
```

We can set up the recurrence by analyzing the algorithm. We're splitting the array in two so we know we'll have n/2. We also have to do 2 recursive calls so a = 2. Finally, 2 comparisons mean we have a constant factor of 2. So the recurrence is $T(n) = 2T(n/2) + 2$. Again, we can use the master method to solve it. a = 2, b = 2, f(n) = 2 (constant), and $n^{lob_b a} = n^1 = n$. Comparing n to f(n) = 2 we see that n dominates, so we have case 1. Thus, it's $\Theta(n)$.

If we were to do this iteratively, we know that we need to look at every element and compare it to the current max and min (the max and min would be set to the first element in the array to start). So the running time would $\Theta(n)$, same as the recursive algorithm.

Question 5

We can make use of the fact that the element is a majority element if it's greater than n/2. Keep splitting the array until it's down to a single element each. Then start merging each array together and returning the majority of that array. Once you get to the top, we just need to compare the total count of the returned majority of each sub-array (i.e. if the majority of the left sub-array is 5 and it occurs 4 times in an array of length 8, see if it occurs at all in the right sub-array to determine if it is a majority). If the majority of each sub-array is the same, then we can go ahead and return the majority. If it's different, whichever occurs

Pseudocode:

```
if arraylength = 1:
        return arr[0]
left = majority(arr[0:(arraylength/2)])
right = majority(arr[arraylength/2 : arraylength – 1])
if left.count > right.count:
        if left.count > (arraylength / 2):
                return left
```

else:

if right.count > (arraylength / 2)

return right

Finding the recurrence: We make two recursive calls so a = 2. Since we're dividing and checking we have n/2 with b = 2. Finally, we make 2 comparisons so f(n) = 2n. $n^{\log_b a} = n^1 = n$. n = f(n) so it's case 2. The answer to the recurrence is Θ(nlgn).