Benjamin Tate
04/16/2017
CS 325 – Section 400
Homework #2

1)

a. $T(n) = T(n – 2) + n$
$=T(n-4) + (n-2) + n$
$=T(n-6) + (n-4) + (n-2) + n$
$=T(n-8) + n – 6 + n – 4 + n – 2 + n$
$=T(n-k) + kn/2 – ((k/2)^2 - (k/2))$
Stop when k=n-1 and T(1) = 1
$T(1) +(n^2-n)/2 – ((n^2-2n+1)/2) – (n-1)/2$
$=\Theta(n^2)$

b. $T(n) = 3T(n-1) + 1$
$=3*3T(n-2) + 3 + 1$
$=3*3*3T(n-3) +3^2+3+1$
$=3^kT(n-k) + … 3^2 + 3 + 1$
Stop when k = n-1 and T(1) = 1
$=3^{n-1}T(1) + 3^{n-2} + … + 3 + 1$
$=3^{n-1} + 3^{n-2} + … + 3 + 1$
$=\Theta(3^n)$

c. $T(n) = 2T(n/8) + 4n^2 \rightarrow a=2, b=8, f(n)=4n^2$
$n^{\log_8(2)} = n^{1/3}$
Case 3: $T(n) = \Theta(n^2)$

2)

a. STOOGESORT works because the three self-calls (0 to k-1, n-k to n-1, and 0 to k-1 again) guarantee overlap between each other, when k is the ceiling of 2n/3. This means that no list element will be left alone until it has reached its rightful place (e.g. the element is correctly oriented with those around itself), and even if the lowest element starts at the end of the list, it will gradually be "walked back" to its place at the front.

b. No, STOOGESORT will not work if k=floor(2n/3), because that value of k doesn't guarantee overlapping calls. For example, in the list [4, 5, 6, 1], n=4, so k=floor(8/3)=2. In the first recursive call, STOOGESORT(A[0 … 1]), 4 and 5 will be left alone, in the second, STOOGESORT(A[n-k … n-1]), 6 and 1 will be swapped, but in the third call, 4 and 5 will just be left alone again, leaving us with the list [4, 5, 1, 6]. The problem here is that there was no overlap, so 1 was never compared to 5 or 4.

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

d. $T(n) = 3T(n-1) + 1$
$=3*3T(n-2) + 3 + 1$
$=3*3*3T(n-3) +3^2+3+1$
$=3^kT(n-k) + … 3^2 + 3 + 1$
Stop when k = n-1 and T(1) = 1
$=3^{n-1}T(1) + 3^{n-2} + … + 3 + 1$
$=3^{n-1} + 3^{n-2} + … + 3 + 1$
$=\Theta(3^n)$

3)

a. Rather than finding one midpoint and comparing its value to the target value, a quaternary search algorithm would find 3 points that divide the 4 quarters of the list, and sequentially compare each of them to the target, discarding irrelevant sections of the list as necessary.

Psuedocode:

```
function quatSearch(list[0...n-1], target, low, high) //Initially run with low=0 and high=n-1
   p1 = low + ((high – low) / 4)
   p2 = low + ((high – low) / 2)
   p3 = low + (3(high – low) / 4)
   if list[p1] = target:
      return list[p1]
   else if list[p1] > target:
      return quatSearch(list, target, low, p1 – 1)
   else if list[p2] = target:
      return list[p2]
   else if list[p2] > target:
      return quatSearch(list, target, p1 + 1, p2 – 1)
   else if list[p3] = target:
      return list[p3]
   else if list[p3] > target:
      return quatSearch(list, target, p2 + 1, p3 – 1)
   else: //list[p3] < target
      return quatSearch(list, target, p3 + 1, high)
```

b. $T(n) = T(n/4) + 6$

c. $a=1, b=4, f(n)=6$
$n^{\log_4(1)}=n^0=1$
$T(n) = \Theta(n^{\log_4(1)}\lg n) = \mathbf{\Theta(lgn)}$

d. Quaternary search is equally as fast as binary, which has a speed of $\Theta(\lg n)$.

4)

a. The divide part of this minMax algorithm is simply to divide the list into 2 roughly equal parts until we are left with list segments that are 1 or 2 elements long. The conquer part will be to find the min and max of each of those segments and compare them to one another, keeping track of the overall min and max.

Psuedocode:

```
//Declare min and max variables before function call
function minMax(list, low, high) //Initially, low=0 and high=n-1
   mid = (low + high) / 2
   if high – low > 2:
      minMax(list, low, mid)
      minMax(list, mid, high)
   else if high – low = 2:
      if list[0] > list[1]
         if list[0] > max:
            max = list[0]
         if list[1] < min:
            min = list[1]
       else:
         if list[1] > max:
            max = list[1]
         if list[0] < min:
            min = list[0]
```

b. $T(n) = 2T(n/2) + 4$

c. $a=2, b=2, f(n)=4$
$f(n)=4=O(n^{\log_2(2)-1})=O(1)$
Case 1 applies, where e=1.
$T(n) = \Theta(n^{\log_2(2)}\lg n) = \mathbf{\Theta(n)}$

d. An iterative minmax algorithm would essentially just run through a list once to find the min and

once to find the max, giving it an approximate complexity of Θ(n), meaning that this divide and conquer method is approximately equal in speed.

5)

a. Since a list has a majority element only if the element appears more than n/2 times in the list, that means that the element would be a majority of the left and/or right side(s) of the list as well (i.e. the element appears more than n/4 times in the left and/or right sides). This is the principle the divide and conquer algorithm will work on. If n = 1, the algorithm would just return the first element of the list. If n > 1, the majority algorithm would recursively call itself on the left and right halves of the list. After that, if a majority element is found in the left side, the function will return that element, and if it is found in the right side, it will return that element. Otherwise, the function will return "NOMAJ" to indicate that no majority was found.

b. Psuedocode:

```
function majority(list[0...n-1], low, high)
    mid = (low + high) / 2
    if n = 1:
        return list[0]
    leftMaj = majority(list, low, mid)
    rightMaj = majority(list, mid, high)
    if leftMaj != "NOMAJ":
        return leftMaj
    if rightMaj != "NOMAJ":
        return rightMaj
    return "NOMAJ" //no majority in either half
```

c. $T(n) = 2T(n/2) + \Theta(n)$

d. $a=2, b=2, f(n)=\Theta(n)$

$n^{\log_2(2)}=n^1=n$

$f(n)=\Theta(n^{\log_2(2)})$, so case 2 applies.

$T(n) = \Theta(nlgn)$