HW2

James Palmer

**1)**

a)    𝑇(𝑛) = 𝑇(𝑛 − 2) + 𝑛 ;  using Muster theorem = THETA(n^2)

b)    𝑇(𝑛) = 3𝑇(𝑛 − 1) + 1; using Muster theorem = THETA(3^n)

c)     𝑇(𝑛) = 2𝑇(𝑛 /8 ) + 4𝑛^2 ; using Master theorem = THETA(n^2)

**2)**

**a)** STOOGESORT breaks the initial container down into smaller sub containers until there are only 2 values left in each container. Then, if the left value is larger than the right, their indices are swapped. This is done for the initial 2/3 of the container, final 2/3, and initial again.

**b)** No, STOOGESORT will not work correctly if the floor is used instead of the ceiling. If n=4, the first recursive call goes from [0..1], the second call goes from [2..3], and the final call goes from [0..1]. The issue here is that no call for [1..2] is ever made to see if [1]>[2].

**c)** Recurrence: T(n)=3T(2n/3)+c

**d)** Solved Recurrence: O(n^3)

**3)**

**a)** quaternary\_search(A, target):

if A[0] < A[n – 1]:

return false // interval is empty or not match

/\* Split A into 3 sections \*/

mid1 = floor ((A[0] + A[n-1)] / 4)

mid2 = mid1\* 2

mid3 = mid2 \* 2

If target == A[mid1]:

Return True // found a match

elif target < A[mid1]:

//recur on first quarter

return quaternary\_search(A[0…mid1-1], target)

elif target > A[mid1] and target < A[mid2]:

//recur second quarter

return quaternary\_search(A[mid1+1…mid2-1], target)

elif target > A[mid2] and target < a[mid3]:

//recur third quarter

Return quaternary\_search(A[mid3+1…n-1], target)

**b)** or normal binary search algorithm is given by s2 = s2(n/2)+1.

The above is a result of one comparison i.e. it breaks on element which breaks the list of elements of sorted keys into two n/2 set of elements and then it makes a point of recursion on the particular partition.

**c)** In quaternary search algorithm we do comparisons on data elements which breaks the partition list into four sections with approximately n/4 data elements. These make recurse on the particular partition. Therefore, the occurance relation for quaternary search is given by S4 = S4 (n/4)+ 2

Applying the second case under binary search as a special case we can state that S4 = (n) = 0(log(n))

**d)** The worst case running times for both algorithms is O(lgn)

**4)**

**a)** procedure maxmin(A[1...n] of numbers) -> (min, max)

f (n == 1)

return (A[1], A[1])

else if (n == 2)

if( A[1] < A[2])

return (A[1], A[2])

else

return (A[2], A[1])

else

(max\_left, min\_left) = maxmin(A[1...(n/2)])

(max\_right, min\_right) = maxmin(A[(n/2 +1)...n])

if (max\_left < max\_right)

max = max\_right

else

max = max\_left

if (min\_left < min\_right)

max = min\_left

else

min = min\_right

return (min, max)

**b)** Tn = a T n/b +F(n)

**c)** For Recursive, lets say that t(n) is  the number of steps the algorithm takes to run on input of size n.

Merging takes linear time and we run each time on two sub-problems of half the original size, so

T(n) =2 \* T (n/2) + O(n). By the master theorem, we see that this recurrence has a "steady state" tree. In short, the run time is: T(n) = O(n \* logn). This can be seen by asking how may times does n need to be divided by 2 before the size of the array for sorting is 1? Why m times of course !

More so, 2m = n , equivalent to log 2m = log n, equivalent to m x log22 = log 2 n , and since log2 2 = 1, equivalent to m = log2n.

Since m is the number of halvings of an array before the array is chopped up into bite sized pieces of 1-element arrays, and then it will take m levels of merging a sub-array with its neighbor where the sum size of sub-arrays will be n at each level, it will be exactly n/2 comparisons for merging at each level, with m ( log2n ) levels, thus O(n/2 x log n ) <=> **O ( n log n)**

**d)** For the Iterative Version, we can use a merge sort algorithm can be turned into an iterative algorithm by iteratively merging each subsequent pair, then each group of four. Due to a lack of function overhead, iterative algorithms tend to be faster in practice. However, because the recursive version's call tree is shorter, it uses less run time. Even when sorting 4 gigs of items would only require  a mere 32 call entries on the stack. The iterative version of merge sort is a minor modification to the recursive version - in fact we can reuse the earlier merging function. The algorithm works by merging small, sorted subsections of the original array to create larger subsections of the array which are sorted. To accomplish this, we iterate through the array with successively larger "strides".

5)

**a)** GetFrequency computes the number of times an element (elemlsub or elemrsub) appears in the given array a[1...n]. Two calls to GetFrequency is O(n). After that comparisons are done to validate the existence of majority element. GetFrequency is the linear time equality operation. (b)Using the proposed divide-and-conquer operation, indeed it is possible to give a linear time algorithm. Idea is to pair up the elements arbitrarily to get n 2 pairs. In each pair if the two elements are different we discard both. See the pseudocode below, there are at most n/2 elements left and if A has a majority element, then remaining elements will have the same majority element

**b)**

procedure GetMajorityElement(a[1...n])

Input: Array a of objects

Output: Majority element of a

if n = 1: return a[1]

k = (n/2)

elemlsub = GetMajorityElement(a[1...k])

elemrsub = GetMajorityElement(a[k+1...n]

if elemlsub = elemrsub:

return elemlsub

lcount = GetFrequency(a[1...n],elemlsub)

rcount = GetFrequency(a[1...n],elemrsub)

if lcount > k+1:

return elemlsub

else if rcount > k+1:

return elemrsub

else return NO-MAJORITY-ELEMENT

**c)** T(n) = 2T (n/2) +O(n)

**d)** The complexity of the function GetMajorityElement is O(n). Therefore the entire algorithm is linear in terms of N