**CS 325 - Homework Assignment 2**
**Due Sunday at 11:59pm**
Victor Ness

**Problem 1**: *(3 points)* 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) $(n) = (n-2) + n$

Using the Muster Method. T(n)=aT(n-b)+f(n)

a=1, b=2, f(n)=n, so d=1. $so\ f(n) = \theta(n^d)$

Therefore, T(n) = $\theta(n^{d+1}) = \theta(n^2)$

B) $(n) = 3(n-1) + 1$

Using the Muster Method

a=3, b=1, f(n)=1, so d=0. $so\ f(n) = \theta(n^0)$

Therefore, T(n)=$\theta\left(n^d a^{\frac{n}{b}}\right) = \theta(n^0 3^{\frac{n}{1}}) = \theta(3^n)$

C) $T(n) = 2T\left(\frac{n}{8}\right) + 4n^2$

Using the Master Method

a=2, b=8, $\log_b a = 1/3$->$n^{1/3}$, f(n)=$4n^2$.

$f(n) = \Omega n^{\frac{1}{3}}$, case 3

Check regularity

$2 * 4\left(\frac{n}{8}\right)^2 \leq c4n^2 = \frac{n^2}{8} \leq c4n^2$, c = .5 is a solution

$T(n) = \theta(n^2)$

**Problem 2:** *(4 points)* 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).
-Stoogesort works by breaking down the input array into the first 2/3 and last 2/3. It recursively breaks down until n=3. The next recursive calls after that will examine the first 2 elements of a 3 element portion of the array and switch them if needed. Next it will examine the second 2 elements in that portion and switch them if needed. Lastly it will examine the first 2 again to see

if they need to be switched again. Using this method, stoogesort will iterate through the array looking at 3 element chunks, moving smaller elements towards the front as it goes.

b) Would STOOGESORT still sort correctly if we replaced k = ceiling(2n/3) with k = floor(2n/3)? If yes prove if no give a counterexample. (Hint: what happens when n = 4?)
-No it would not sort correctly. For example if n=4, it would enter the else if setting k=2. Then it would examine elements [0,1], [2,3], then [0,1] again and finish. There is no overlap between the examined sections of the array so elements cannot move more than 1 index closer to their correct position.

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

$$T(n) = 3T\left(\frac{2n}{3}\right) + c$$

d) Solve the recurrence to determine the asymptotic running time..

$By\ the\ Master\ Method$

$T(n) = 3T\left(\frac{2n}{3}\right) + c$

$a = 3, b = \frac{3}{2}, f(n) = n^0$

$f(n) = \mathrm{O}(\mathrm{n}^{\sim 2.7})$

Case 1

$T(n) = \theta(n^{2.7})$

**Problem 3:** *(6 points)* 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.

a) Verbally describe and write pseudo-code for the quaternary search algorithm.

```
qSearch(array, left, right, searched)
  mid1 = left + (right -1)/4
  mid2 = mid1 + (right -1)/4
  mid3 = mid2 + (right -1)/4

  if (array[mid1] == searched) return mid1
  if (array[mid2] == searched) return mid2
  if (array[mid3] == searched) return mid3
  if (searched < array[mid1]) qSearch(array, left, mid1-1, searched)
  if (searched < array[mid2]) qSearch(array, mid1, mid2-1, searched)
  if (searched < array[mid3]) qSearch(array, mid2, mid3-1, searched)
  if (searched < array[right]) qSearch(array, mid3, right-1, searched)

Return -1 //value indicating searched was not found
```

My qSearch algorithm first finds the 3 mid indices and checks to see if any of them are searched. Next, it goes quarter by quarter checking to see if the searched for value is inside. If searched is in the quarter, the function will recursively call itself with the left and right boundaries equaling the boundaries of that quarter. If the searched value is either lesser/greater than the original array boundaries, or if after recursing the value is not found, the function will return -1.

b) Give the recurrence for the quaternary search algorithm

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

c) Solve the recurrence to determine the asymptotic running time of the algorithm.

$Using\ the\ Master\ Method$

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

$a = 1, b = 4$

$\log_4 1 = 0 \rightarrow n^0 = 1$

$f(n) = \theta(1) = \theta(n^0)$

$Case\ 2$

$T(n) = \theta(n\text{^}(\log_b a)\ lgn$

$T(n) = \theta(lgn)$

d) Compare the worst-case running time of the quaternary search algorithm to that of the binary search algorithm.

-The worst case for both algorithms will be asymptotically the same (O(lgn)), however quaternary search should make many more comparisons at each step so it would run slower.

**Problem 4**: *(6 points)* Design and analyze a **divide and conquer** algorithm that determines the minimum and maximum value in an unsorted list (array).

a) Verbally describe and write pseudo-code for the min_and_max algorithm.

```
minMax(array)
  if (len(array) == 2)
    if (array[0] < array[1])
      return array
    else
      swap(array[0], array[1])
      return array
  else if(len(array)==1)
    return array
  else
    arr1 = minMax(array[:(len(array)//2)
    arr2 = minMax(array[(len(array)//2):);
    retArray[];
    if (len(arr1) == 1)
      if (arr1[0] < arr2[0]) retArray[0] = arr1[0], retArray[1]=arr2[1]
      else if (arr1[0] > arr2[1]) retArray[1] = arr1[0], retArray[0]=arr2[0]
    else if (len(arr2) == 1)
      if (arr2[0] < arr1[0]) retArray[0] = arr2[0], retArray[1]=arr1[1]
      else if (arr2[0] > arr1[1]) retArray[1] = arr2[0], retArray[0]=arr1[0]
    else
      if (arr1[0]<arr2[0]) retArray[0] = arr1[0]
      else retArray[0] = arr2[0]
       if (arr1[1]>arr2[1]) retArray[1] = arr1[1]
       else retArray[1] = arr2[1]

     return retArray;
      //I'm going to simplify this pseudocode in future assignments
```

The minMax function takes an input array and breaks it down until n = 1, 2. If the input is size 1, it will return that array, if size 2, it will set the minimum value in index 0 and the maximum in index 1. If the size of the input array is greater than 2, minMax will set arr1 and arr2 equal to minMax of the first and second half of the input array respectively. Arr1 will be a 2 element array where index 0 is the minimum of the left half of the input array and index 1 is the maximum of the left half of the array. Arr2 is the same but for the right half of the array. Arr1 and Arr2 are then compared and their min and max is put into a return array which sent up the recursion chain.

b) Give the recurrence.

$T(n) = 2T\left(\frac{n}{2}\right) + c$

c) Solve the recurrence to determine the asymptotic running time of the algorithm.

$Using\ the\ Master\ Method$

$T(n) = 2T\left(\frac{n}{2}\right) + c$

$a = 2, b = 2, \log_b a = 1 \rightarrow n\text{^}(\log_b a) = 1\,,\ \ f(n) = 0$

$Case\ 1: f(n) = O(n^1)$

$T(n) = \theta(n)$

d) 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.

-The iterative algorithm for minMax made 1.5n comparisons so it also will run at $\theta(n)$ time complexity. This means both the recursive and iterative versions of the min max function will run similarly asymptotically.

**Problem 5**: *(6 points)* 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.

a) Give a detailed verbal description for a **divide and conquer** algorithm to find a majority element in A (or determine that no majority element exists).

-The getMajority function will recursively break down the input array, setting leftMajority equal to getMajority of the left half of the array and rightMajority equal to the right half of the array. The recursion will go down until array size is 1, then return the single element in the array. At the level above n=1, if leftMajority and rightMajority are equal then that element will be return as a potential majority element, else no majority element will be returned. If a potential majority element is found, it will be checked for its frequency against the array at that particular level of recursion.

b) Give pseudocode for the algorithm described in part a)

```
getMajority(array[1…n])
  if len(array) == 1, return array[1]
  leftMajority = getMajority(1,len(array)//2)
  rightMajority = getMajority(len(array)//2+1)
  if (leftMajority==rightMajority), return leftMajority

  leftCount = frequency(array[1…n], leftMajority) //returns count of element in array
  rightCount = frequency(array[1…n], rightMajority)

  if (leftCount >= (len(array)//2)+1, return leftMajority
  else if (rightCount >= (len(array)//2)+1, return rightMajority
  else return NONE

//frequency function must be written to accommodate an input value of NONE
```

c) Give a recurrence for the algorithm

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

d) Solve the recurrence to determine the asymptotic running time.

$Using\ the\ Master\ Method$

$a = 2\ , b = 2, \log_b a = 1 \rightarrow n\text{^}(\log_b a) = n,\ \ d = 1, f(n) = n$

$Case\ 2$

$T(n) = O(n^d \log(n))$

$T(n) = O(nlogn)$

*Note: A O(n) algorithm exists but your algorithm only needs to be O(nlgn).*