Brandon Swanson
CS 325 HW2
10/11/15

Problem 1

A)

$$T(n)=O(n^2)$$

this can be demostrated with the substitution method

$$GUESS: T(n)=O(n^2)$$
$$GOAL: T(n)\leq cn^2$$
$$INDUCTIVE\ HYPOTHESIS: T(n-2)\leq c(n-2)^2$$
$$T(n-2)+n\leq c(n-2)^2+n\leq cn^2$$
$$=c(n^2-4n+4)+n\leq cn^2$$
$$=cn^2-(4cn+4+n)\leq cn^2$$
$$if\ (4cn+4+n)\geq 0\ then\ the\ inequality\ holds$$
$$any\ constant\ greater\ than\ 1\ will\ work$$

B)

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

this can be seen by looking at the iterations and noting that it is some constant n times the recursion goes

c + T(n-1)
c + c + T(n-2)
c + c + c + T(n-3)

all the way until n = 0, therefore this has linear running time

C)

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

This can be proven by applying the Master Therom The recurence $T(n)=2T(n/4)+n$ fits into case 3

$$f(n)=n=\Omega(n^{\log_4 2+\epsilon})\ \text{or}\ \Omega(n^{.5+\epsilon})$$
$$n^1\ will\ always\ be\ bounded\ below\ by\ n^{.5+\epsilon}\ for\ any\ \epsilon<.5$$
$$This\ formula\ also\ satisfies\ the\ regularity\ condition$$
$$2f(n/4)\leq cf(n)$$
$$n/2\leq cn$$
$$any\ constant\leq .5\ will\ satisfy\ the\ regularity\ condition$$

D)

$$T(n)=\Theta(n^{2.5})$$

this can be found using the master therom case 3 but first it is helpful to transform the equation thusly:

$$T(n)=4T(n/2)+n^2*n^{.5}$$
$$=4T(n/2)+n^{2.5}$$
$$now\ we\ can\ see\ that\ f(n)=\Omega(n^{\log_2 4})$$
$$\Rightarrow n^{2.5}=\Omega(n^2)$$
$$therefore\ fufilling\ case\ 3\ tge\ running\ time\ is\ \Theta(f(n))\Rightarrow\Theta(n^{2.5})$$

Problem 2

Algorithm A can be expressed in the form $T(n)=5T(n/2)+n$ which can be shown using the master method case 1 to have a running time of $\Theta(n^{lg\,5})$

Algorithm B can be expressed in the form $T(n)=2T(n-1)+1$ and has a running time of $T(n)=O(2^n)$ this can easily be guessed as the recurrence will occur n times and each time doubling. It can be further proven using the substitution method

Algorithm C can be expressed in the form $T(n)=9T(n/3)+\Theta(n^2)$ and has a running time of $T(n)=\Theta(n^2\,lgn)$ This can be proven with the master method case 2 as $n^2=\Theta(n^{\log_3 9})$

I would choose algorithm A because A = O(C) = O(B)

Problem 3

The amount of times "Print" is echoed can be expressed by the recurrence $P(n)=4\,P(n/2)+n$
This recurrence can be solved using the master method and case 1 as $n=O(n^{\log_2 4-\epsilon})$ or $n=O(n^{2-\epsilon})$ therefore the amount of prints is $\theta(n^2)$

Problem 4

pseudo code for ternary search algorithm:

```
tsearch(val, array, left, right)
     #base case
     if left == right
          return array[left] == val

     #recursive case

     #length of subarry
     length = right – left + 1
     third = length/3

     if array[left+third] >= val
          return tsearch(val,array,left,left+third)
     if array[left+third*2] >=val
          return tsearch(val,array,left+third,left+third*2)
     else
          return tsearch(val,array,left+third*2,right)
```

The recurrence for this ternary search algorithm is $T(n)=T(n/3)+1$
This is a very similar recurrence to that of binary search except to a log base 3 which will get simplified asymptotically to O(lgn). So when comparing this ternary search to binary search they have the same asymptotic even though ternary will recur fewer times but also has more constant cost each iteration.

Problem 5

a) the time it takes to merge n elements is n, or rather if you have 2 lists of size m it will take 2m time to merge them, so if n = 2m then T(n) = n
merging two unequally sized list takes the sum of the two list sizes, at the each iteration there will be a list size (k-1)*n and that will be merged with size n list, so the recurrence can be stated as
T(n) = T(k-1) + kn
this represents the series 2n + 3n + 4n + 5n …. kn
this sum of 1 through k is asymptoticly simplified to k^2 therefore this algorithm is $\Theta(k^2 n)$

b) A better algorithm would be one that separates the k lists of n length into groups of no more then 2, merging them all together, and then merges 2 at a time of each of these 2n lists (of which there are now k/2). This algorithm is very similar to a merge sort on one list.
Here is a small psuedocode

```
#returns a merged list
merge(lista,listb)

#returns a merge list from klists
mergeklists(lists)
     if len(lists) == 1: return lists[0]
     if len(lists) == 2: return merge(lists[0],lists[1])

     return merge ( mergeklists(lists.firsthalf) ,
                    mergeklists(lists.secondhalf) )
```

The recurrence for this algorithm is T(n) = 2(k/2) + kn which means this algorithm is $\Theta(k * n * lgk)$ much better than the other algorithms exponential time.

Problem 6

a)
insertion soring a list of size k will be a $\Theta(k^2)$ so performing insertion sort on all n/k lits of size n will be

$$\theta(K^2 * \frac{n}{k}) \Leftrightarrow \Theta(k * k * \frac{n}{k}) \Leftrightarrow \Theta(nk)$$

b)
again very similar to the process of merging k lists, merging 2 lists of size n/2 takes time n, when merging the n/k number of lists they are merged 2 at a time, reducing the number of lists by a division of 2 at each level. This is a recurrence exactly like merge sort only the tree stops when there are n/k leaves, giving yield to a logarithmic running time using the master method case 2 to confirm the recursive tree analysis

$$RECURANCE: T(n) = 2(\frac{n/k}{2}) + n$$

$$RUNNING\ TIME: T(n) = \Theta(n lg(n/k))$$

c)

what is the largest value of k as a function of n for wich the new has the same Θ running time as merge

$$\Theta(nk+n\lg(n/k))$$
$$=\Theta(n*(k+lg(n/k)))$$
$$=\Theta(n*(k+lgn-lgk))$$
$$=\Theta(n*(k+lgn))$$
$$=\Theta(nk+nlgn)$$

for $\Theta(nk+nlgn)=\theta(nlgn)$ nk must be the lower order term, the only way for that to be possible is if :

$$k=\Theta(lg\,n)$$

d) We should choose optimal k values by analyzing the constant factors involved in dividing and combining lists with the constant factors involved in insertion sorting small lists to decide on an optimal k. In other words we should perform experimental analysis for different values of k to find the true effect of the difference in constant factors between merge and insertion sort on smaller sized lists

Problem 7

Divide and conquer pseudo code

```
min_max(list):
     #basecase
     if len(list) == 1
          return list[0],list[0]

     leftmin,leftmax = min_max(list.firsthalf)
     rightmin,rightmax = min_max(list.firsthalf)

     return min(leftmin,rightmin),max(leftmax,rightmax)
```

Iterative pseudo code

```
min_max(list):
     min = list[0]
     max = list[0]

     for index in [0...n-1]
          if list[index] > max
               max = list[index]
          if list[index] < min
               min = list[index]

     return min,max
```

Recurrence for divide and conquer

$$T(n) = 2T(n/2) + 1$$

when the algorithm has fully recursed there will be n lists of size 1 with constant time operation on each, then there will be n/2 min and max values to consider in n/2 constant comparisons

this recurences yeilds the series of $n + \frac{1}{2}n + \frac{1}{4}n + \frac{1}{8}n$ wich will sum to 2n, therefore it has a runtime:

$$\Theta(n)$$

as we can easily see the runtime of the iterative approach is linear as it considers each element twice iterative runtime: $\Theta(n)$

The asymptotic analysis of these two running times has managed to obscure the constant factors involved, both the time needed to divide and combine and the fact that the divide and conquer makes 2n comparisons instead of n. But asymptotically these are both linear algorithms.