Travis Robinson
CS325
Homework 2

1)

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

Guess: $O(n^2)$

Induction Goal: $T(n) <= d*n^2$

Induction Hypothesis: $T(n-2) <= d(n-2)^2$

Proof:

$T(n) = T(n-2) + n <= d(n-2)^2 + n$

$=d(n^2-2n-2n+4)+n$

$=dn^2-4nd+4d+n <= dn^2$

$=dn^2-(4nd-4d-n) <= dn^2$

True when $4nd-4d-n >= 0$ -or- $4nd-n >= 4d$

$4nd >= 4d-n$, True for $n, d >=1$

b)

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

Guess: $O(n)$

Induction Goal: $T(n) <= n$

Induction Hypothesis: $T(n-1) <= c(n-1)$

Proof:

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

$= T(n-1)<= c(n-1)$

$=cn-c<=cn$

True, for $c >= 0$

c)

$2T(n/4)+n$; $a = 2$, $b = 4$, $\log_4 2 = .5$, $f(n) = n$

Case 3: $.5 < 1$

Check regularity:

$2(n/4)<=cn$

$.5n<=cn$, True for $n>=.5$

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

d)$T(n) = 4T(n/2) + n^{2.5}$; $a =4$, $b = 2$, $\log_2 4 = 2$, $f(n) = n^{2.5}$

Case 3, since $2<2.5\Theta(n^{2.5})$

Check regularity:

$4(n/2)^{2.5} <= c(n)^{2.5}$

$4n^{2.5}/2^{2.5} <= cn^{2.5}$

$c = 4/(2*2^{.5}) = 2/2^{.5} = 2^{.5}$

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

2)
A: $T(n) = 5T(n/2) + n$
$a=5$, $b=2$, $f(n) = n$, $\log_2 5 = 2.322$
Case 3, $2.322>1$
Check regularity:
$5(n/2)<=c(n)$, $c = 5/2$
$T(n) = \Theta(n)$

B: $2T(n-1)+c$
Guess: $O(n)$
Induction Goal: $T(n) <= cn$
Induction Hypothesis: $T(n-1) <= c(n-1)$
Proof:
$T(n) = 2T(n-1)+c <= c(n-1)+c$
$=cn-c+c = cn <= cn$
$T(n) = O(n)$

C: $T(n) = 9T(n/3)+n^2$
$a=9$, $b=3$, $f(n) = n^2$, $\log_3 9 = 3$
Case 3, $3>2$
Check regularity:
$9(n/3)<= n^2$
$=3n <=n^2$
$T(n) = \Theta(n^2)$

I would choose algorithm B; It works in linear time, and is big-O so could potentially work faster, as opposed to alforithm A, which works in Theta time, meaning it can't work faster

3) $T(n) = 4T(n/2)+n$
$a=4$, $b=2$, $f(n) = n$, $\log_2 4 = 2$
Case 3, $2>1$
Check regularity:
$4(n/2)<=cn$
$=2n<=cn$
$c = 2$

4)

```
ternarySearch(root, x)//x is value searched for
link = root
while (link->next != NULL)
{
        if (link→value) == x
                return
        else
                {
                        // compare(x)  function written by user to compare x values and determine which
                        //branch to follow—will run in constant time (comparing two values is a constant
                        //time function)

                        if (compare(x) == 1)
                                link→next = left_branch
                        else if (compare(x) == 2)
                                link→next = center_branch
                        else if (compare(x) == 3)
                                link→next = right_branch
                }
}
```

$T(n) = T(n/3)+c$

$a=1$, $b=3$, $f(n) = c$, $\log_3 1 = 0$

Case 2, $0 = 0$

$T(n) = \Theta(n^0 \lg n) = \Theta(\lg n)$

In the worst case, the value 'x' is never found; in this case, the ternary search will provide better time than the binary, because it can rule out 2/3 of the remaining tree with each iteration, while the binary tree can only rule out ½. To put it another way, in the worst case, a tree search algorithm will have a time corresponding to the height of the tree. A ternary tree will be shorter because it has more branches, so it will work faster in the worst case ($\log_3$ vs $\log_2$)

5)

a) Each time an array gets merged, there are one fewer merges that need to happen, so first there's k merges, then (k-1), then (k-2) etc down the last merge. The sum of this is the same as the of Σk, which is k(k+1)/2. Since each k has n elements, the total time is $nk(k+1) = nk^2+nk$, or $\Theta(nk^2)$ complexity.

b) If we act as if there is an array of these arrays, we can recursively merge them, with each iteration merging the two lower levels of merging. So if the base case is two arrays, they get merged. At the next level up, two of these merged arrays get merged, and this happens until all arrays are merged together.

6)
a: Worst time happens when all elements are in reverse order; this means that each element needs to moved k times, so in the worst case, insertion sort runs in nk time.
Guess: T(n) = Θ(nk)
Induction Goal: T(n) <= dnk
Hypothesis: T(n/k) <= d(n/k)k
Proof: T(n) = nk <= dnk/k = dn
True, for d>=k

b: To do this we would go through the n/k sublists, merging the values into the main list. This would take nlg n time for the merging, but because we have n/k sublists, it will instead take the log of that, being n lg (n/k)

c: nk + n lg (n/k) = nk + n(lg n – lg k)
nk <= n(lg n – lg k)
k <= lg n – lg k
k + lg k <= lg n
The largest value for k is k+lg(k)<=lg(n)

d: After finding k, experiment to see if that k value is still more efficient on that particular machine with that particular language, due the cost of the constants of the algorithm costing different times across different machines.

7)
```
min_and_max_algorithm(array)
        if (array.Length <= 2) //base case-finds element that is smaller and larger, returns them
        {
                return smallerElement
                return largerElement
        }
        else (array.length > 2) //recursive case-
        {
        //smaller element from left array
                smallerElement1 = min(min_and_max_algorithm(array[0..length/2]))

        //larger element from left array
                largerElement1 =  max(min_and_max_algorithm(array[0..length/2]))

        //smaller element from right array
                smallerElement2 = min(min_and_max_algorithm(array[length/2+1..length]))

        //larger element from right array
                largerElement2 =  max(min_and_max_algorithm(array[length/2+1..length]))

        //return the smaller and larger from the two arrays
        if (smallerElement1 <= smallerElement2)
                return smallerElement1
        else
                return smallerElement2
```

```
    if (largerElement1 > largerElement2)
        return largerElement1
    else
        return largerElement2
}
```

Guess: O(lg n)
Induction Goal: T(n) <= d lg n
Induction Hypothesis: T(n/2)<=d lg(n/2)
Proof:2T(n/2)+c <= 3d lg (n/2) + c
=3d lg n – 3d lg 2 + c
=3d lgn – 3d (1) + c
= 3d lg n – 3d +c <= 3d lg n
if -3d+c<= 0, True when c<=3d, or d>= 1/3 c

Iterative Algorithm:

```
min = infinity
max = -infinity
for (i = 0; i<array.length)
{
    if (array[i] > max)
        max = array[i]
    if (array[i] < min)
        min = array[i]
}
return min
return max
```

This is a linear algorithm Θ(n) because it goes through each element one time.
The recursive, divide and conquer algorithm works much faster for large n. It may be slower for small n due to the recursive function calls, though this will depend on the machine.