Jan. 20, 2016 Ayesha Ahmed CS430 – Homework Assignment 1

#1)
Base Step:
If $n = 3$, then clearly, $T(3) = 9$ $T(3) = 3^2 = 9$. This is trivial because of recurrence.
Hypothesis Step:
Now assume $T(n) = n^2$ is true when $n = 3^k$ for some integer $k > 0$.
Induction Step:
Let $k \rightarrow k + 1$ so that $n = 3^{k+1}$, then
$T(3^{k+1})$
$= 6T(3^{k+1}/3) + 1/3*(3^{k+1})^2$
$= 6T(3^k) + 1/3*(3^{2k+2})$
$= 6T(3^k) + 3^{2k+1}$
$= 6*(3^k)^2 + 3^{2k+1}$
$= 6*3^{2k} + 3^{2k+1}$
$= 2*3^{2k+1} + 3^{2k+1}$
$= 3^{2k+1} * (2 + 1)$
$= 3^{2k+1} * 3$
$= 3^{2k+2}$
$= (3^{k+1})^2$
Citations: 1) http://cs.boisestate.edu/~jhyeh/teach/cs242_fall04/h1_sol.pdf Author: Professor Jyh-haw Yeh from Boise State University, Title: cs242_fall04/h1_sol, Date: Jan. 17, 2008.
2) http://answers-by-me.blogspot.com/2010/07/clrs-2e-exercise-23-3.html Author: Justin Mancinelli, Title: CLRS 2e: Exercise 2.3-3, Date: Tuesday, July 13, 2010.

#2)
Binary Search is an algorithm that finds the position in a sorted array by checking the midpoint of the sequence and repeats this procedure with the remaining portion. Since we are taking half the sequence each time we divide n by 2. So since it will always be a constant, we have $T(n) = T(n/2) + 1$
Base Case:
We have the base case as $n = 1$ where the list is already sorted so there is no work. Thus we have constant time $O(1)$.
Worst Case:
When the value is not in list, the algorithm must continue iterating until the span has been made empty. This will take at most $\log(n) + 1$ iterations. Thus, Binary Search on a sorted array has worst case time complexity $\Theta(\log n)$. Note this is log base 2 of n.
Therefore the recurrence of Binary Search is:

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ T(n/2) + 1 & \text{if } n > 1. \end{cases}$$

The solution to this recurrence is: $T(n) = \Theta(\log n)$.
Citations: 1) http://cs.boisestate.edu/~jhyeh/teach/cs242_fall04/h1_sol.pdf
Author: Professor Jyh-haw Yeh from Boise State University, Title: cs242_fall04/h1_sol, Date: Jan. 17, 2008.
2) http://cs.stackexchange.com/questions/13168/recurrence-for-recursive-insertion-sort
Author: Aseem Bansal, Title: Recurrence for recursive insertion sort, Date: Jul. 9, 2013.

3) http://stackoverflow.com/questions/18808429/understanding-recurrence-for-running-time
Author: Harrison, Title: Understanding recurrence for running time, Date: Sep. 15, 2013
4) https://en.wikipedia.org/wiki/Binary_search_algorithm
Author: Wikipedia, Title: Binary Search Algorithm, Updated: Jan. 2016.

#3)
By doing the matrix multiplication $A_{mxn} \times A_{nxo}$ we get A_{mxo} which is a matrix with m rows and o columns.
Let A = A_{mxn} , B = A_{nxo} and C = A_{mxo} so that AxB = C then using Strassen's algorithm we get:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

From this we can make an algorithm which loops over the indices *i* from 1 through *m*, and *j* from 1 through *o*, and *k* from 1 through *n* using nested loops:

```
Input: matrices A and B
Let C be a new matrix of the appropriate size
For i from 1 to m:
        For j from 1 to o:
                Let sum = 0
                For k from 1 through n:
                        Set sum ← sum + A_ik + B_kj
                Set C_ij ← sum
Return C
```

Since we iterate through every k for a j value and every j for an i value then we will have ixjxk time complexity to go through all dimensions of the two matrices. Thus the time complexity is Θ(n)* Θ(m)* Θ(o) = Θ(n*m*o). If both matrices being multiplied were square n by n matrices then the time complexity would be $\Theta(n^3)$.

Citations: 1) Textbook Page 75.
2) https://en.wikipedia.org/wiki/Matrix_multiplication_algorithm
Author: Wikipedia, Title: Matrix Multiplication Algorithm, Updated: December 2015

#4)
Ordered: <u>slow</u> $(\sqrt{2})^{\lg n}$, $\lg n^2$, $2^{\lg n}$, $\sqrt{n} * (n/e)^n$, $(n+1)!$, n^n <u>fast</u>
Justifications:

- First n^n = ω(2^n) and (n + 1)! = ω(2^n) but the limit as n→∞ of (n+1)n! / n^n converges to 0 so therefore n^n is faster and (n + 1)! = Ω(n^n).
- Then $\sqrt{n} * (n/e)^n$ = Ω((n+1)!) because it has a negative polynomial exponent which hinders its growth so it is e^(-n) * n^(n + ½) < n!.
- Also, $2^{\lg n}$ = n because $c^{\log_b a} = a^{\log_b c}$. So this is a linear polynomial function which is slower than exponential functions. Thus, n = Ω(e^(-n) * n^(n + ½)).
- Now lg n^2 = 2 lg n is a poly-logarithmic function which is slower than polynomial function so lg n^2 = Ω(n).

Jan. 20, 2016 Ayesha Ahmed CS430 – Homework Assignment 1

- Lastly, $(\sqrt{2})^{\lg n} = \sqrt{n}$ because $(\sqrt{2})^{\lg n} = 2^{(1/2)*\lg n} = 2^{\lg \sqrt{n}} = \sqrt{n}$. This has a decimal exponential which grows slower than a linear polynomial function so $\sqrt{n} < \Omega(n)$ and $\sqrt{n} < \Omega(\lg n\text{^}2)$.

Citations: 1) http://ocw.mit.edu/resources/res-18-005-highlights-of-calculus-spring-2010/derivatives/growth-rate-and-log-graphs/MITRES18_05S10_Growth_Rate_Log_Graphs.pdf Author: Gilbert Strang, Title: Highlights of Calculus, MIT OpenCourseWare, Date: Spring 2010.
2) Recitation notes January 15.

#5)

Master Theorem:

a = 2, b = 2, f(n) = n. So, $\frac{af(n/b)}{f(n)} = \frac{2(n/2)}{n} = \frac{2n}{2n} = 1$ for all n. So, since $af(n/b) = f(n)$ then we have case 2, $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(n^{\log_2 2} \lg n) = \Theta(n \lg n)$.

Secondary Recurrence:

Let $n_i = n$ and $n_{i-1} = n/2$. Now assume $T(1)$is the base case and $n_0 = 1$ and solve for Θ notation of the function:

$n_i = 2n_{i-1} \rightarrow n_i = \alpha 2^i$ (corresponds to (E – 2))

Since $n_0 = 1$, $\alpha = 1$, and $n_i = 2^i$. Now define $F(i) = T(n_i)$. Then, the original recurrence:

$T(n) = T(n_i) = 2T(n/2) + n = 2T(n_{i-1}) + n$

Becomes:

$F(i) = 2F(i-1) + n$

We have supposed $n = n_i$, and we derived that $n_i = 2^i$. Therefore, the final recurrence to solve is:

$F(i) = 2F(i-1) + (2^i)$

Which is annihilated by $(E-2)^2$. The corresponding closed formula is $(\alpha_1 + \alpha_2)*2^i$, which is $\Theta(2^i)$. Recall that $n = 2^i$. We can achieve the final Θ notation by undoing the substitution as follows:

$T(n) = F(i) = \Theta(2^i) = \Theta(2^{\log_2 n} \lg n) = \Theta(n^{\log_2 2} \lg n) = \Theta(n \lg n)$.

Also, we could solve by substituting $n = 2^k$ into $T(n) = 2T(n/2) + n$ because n/2 will be one lower in the sequence than 2^{k-1}. This gives us $T(2^k) = 2T(2^{k-1}) + 2^k$. Now, let $t_k = T(2^k)$ so that we get $t_k = 2t_{k-1} + 2^k$. The annihilator for the homogeneous part $t_k = 2t_{k-1}$ is (E – 2) and the annihilator for the non-homogenous part 2^k is (E – 2) so the result annihilator for the whole equation is $(E-2)^2$. Now un-substitute $n = 2^k$ so that $k = \lg_2(n)$ to solve the recurrence. Since $(E-2)^2$ annihilates sequences of type $2^k k$ then plugging in $k = \lg_2(n)$ will give us $2^{\lg n}\lg n = n \lg n = \Theta(n \lg n)$.

Citations: 1) http://cs.iit.edu/~cs430/scribbling/jan13.pdf Author: Edward Reingold, Title: in-class scribbling jan13, Date: Jan 13 2016.
2) Textbook page 94
3) http://cs.iit.edu/~cs430/lecture-notes/jan13.pdf Author: Edward Reingold, Title: Lecture 2: January 13, Date: Jan 13 2016.