Problem 3.1

a) The code implementation is in 1-Recursion.c file.

b) The code implementation is in 2-Recursion.c file. The time taken for each method is taken separately by restricting with #define and commenting out the definition of other methods while measuring one method with different manual inputs of n. The random time taken results are saved in particular text files, and then imported to an excel sheet to calculate the average time.

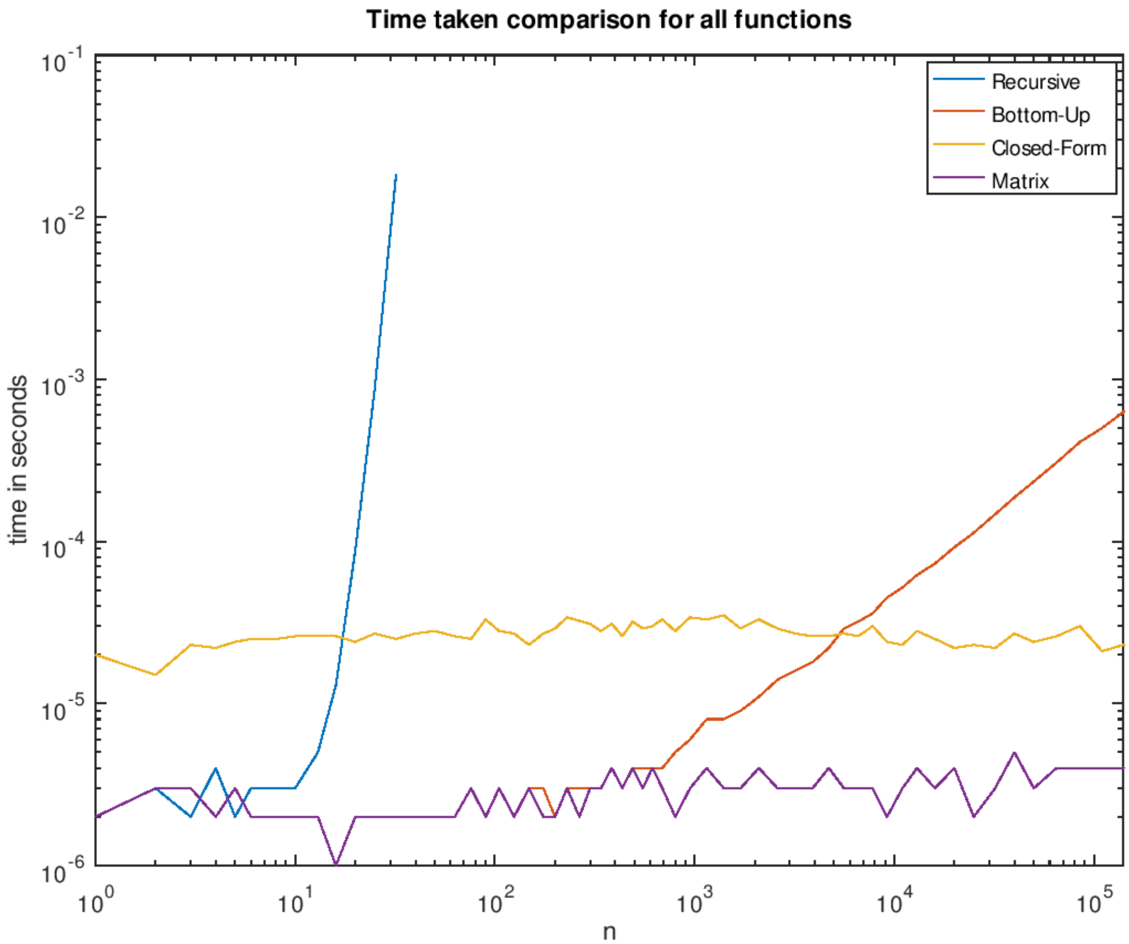
   Initially, I planned to stop sampling for all methods is stopped if the time taken exceeds 0.0006 seconds. However, I had to stop data sampling for closed-form and matrix method before reaching the time limit due to the slower growth rate for larger n values.

   The table below indicates the data sampling for all four methods.
   (The text files are updated with final calculated average values.)

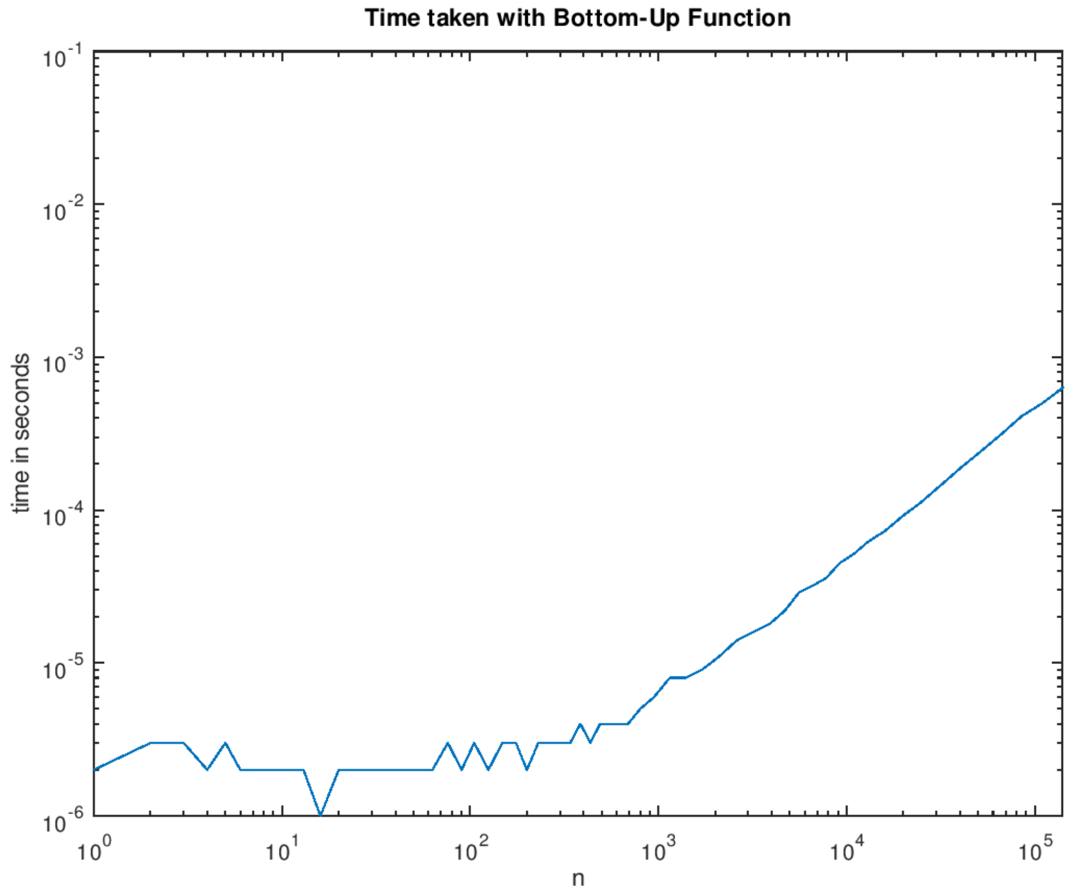
N	Recursive	Bottom-Up	Closed Form	Matrix
0	0.000002	0.000003	0.000021	0.000003
1	0.000002	0.000002	0.00002	0.000002
2	0.000003	0.000003	0.000015	0.000003
3	0.000002	0.000003	0.000023	0.000003
4	0.000004	0.000002	0.000022	0.000002
5	0.000002	0.000003	0.000024	0.000003
6	0.000003	0.000002	0.000025	0.000002
8	0.000003	0.000002	0.000025	0.000002
10	0.000003	0.000002	0.000026	0.000002
13	0.000005	0.000002	0.000026	0.000002
16	0.000013	0.000001	0.000026	0.000001
20	0.000088	0.000002	0.000024	0.000002
25	0.000865	0.000002	0.000027	0.000002
32		0.000002	0.000025	0.000002
40		0.000002	0.000027	0.000002
50		0.000002	0.000028	0.000002
63		0.000002	0.000026	0.000002
76		0.000003	0.000025	0.000003
90		0.000002	0.000033	0.000002
105		0.000003	0.000028	0.000003
125		0.000002	0.000027	0.000002
148		0.000003	0.000023	0.000003
175		0.000003	0.000027	0.000002
200		0.000002	0.000029	0.000002
230		0.000003	0.000034	0.000003
265		0.000003	0.000029	0.000002
300		0.000003	0.000031	0.000003
340		0.000003	0.000028	0.000003
385		0.000004	0.000031	0.000004
435		0.000003	0.000026	0.000003
490		0.000004	0.000032	0.000004
550		0.000004	0.000029	0.000003
615		0.000004	0.00003	0.000004
690		0.000004	0.000033	0.000003
800		0.000005	0.000028	0.000002
950		0.000006	0.000034	0.000003
1150		0.000008	0.000033	0.000004
1400		0.000008	0.000035	0.000003
1700		0.000009	0.000029	0.000003
2100		0.000011	0.000033	0.000004
2600		0.000014	0.000029	0.000003
3200		0.000016	0.000027	0.000003
3900		0.000018	0.000026	0.000003
4700		0.000022	0.000026	0.000004
5600		0.000029	0.000027	0.000003
6600		0.000032	0.000026	0.000003
7800		0.000036	0.00003	0.000003
9200		0.000045	0.000024	0.000002
11000		0.000052	0.000023	0.000003
13000		0.000062	0.000028	0.000004
16000		0.000073	0.000025	0.000003
20000		0.000092	0.000022	0.000004
25000		0.000113	0.000023	0.000002
32000		0.000147	0.000022	0.000003
40000		0.000187	0.000027	0.000005
50000		0.000234	0.000024	0.000003
65000		0.000306	0.000026	0.000004
85000		0.00041	0.00003	0.000004
110000		0.000501	0.000021	0.000004
140000		0.000627	0.000023	0.000004

c) While testing the return values for all four methods, I used round() function from math.h library, so that the return value will be correct. However, I found out the return value from closed-form method becomes slightly different from other methods if n>70. The return Fibonacci values from all 4 functions are equal until n≤70.

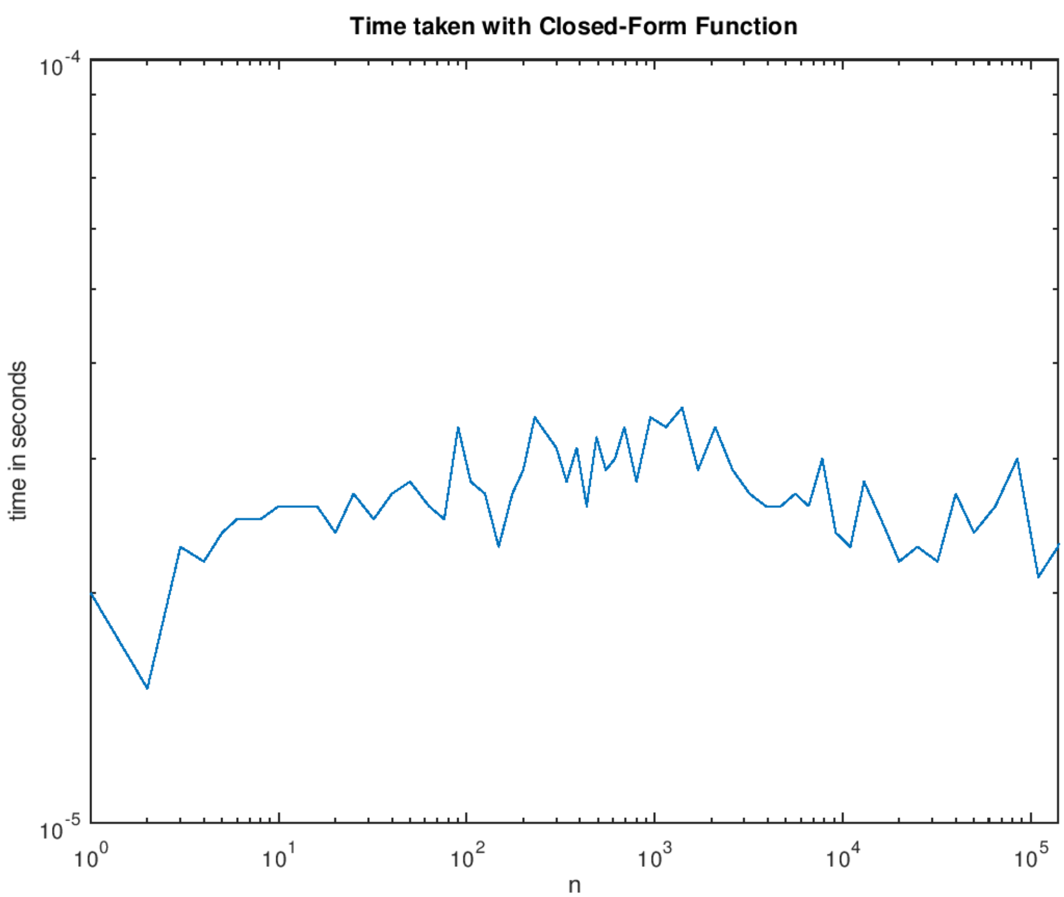
d) The plots are drawn by using Matlab. The codes are written in MatlabCode.txt, and particular parts are pasted on Matlab command line and run.



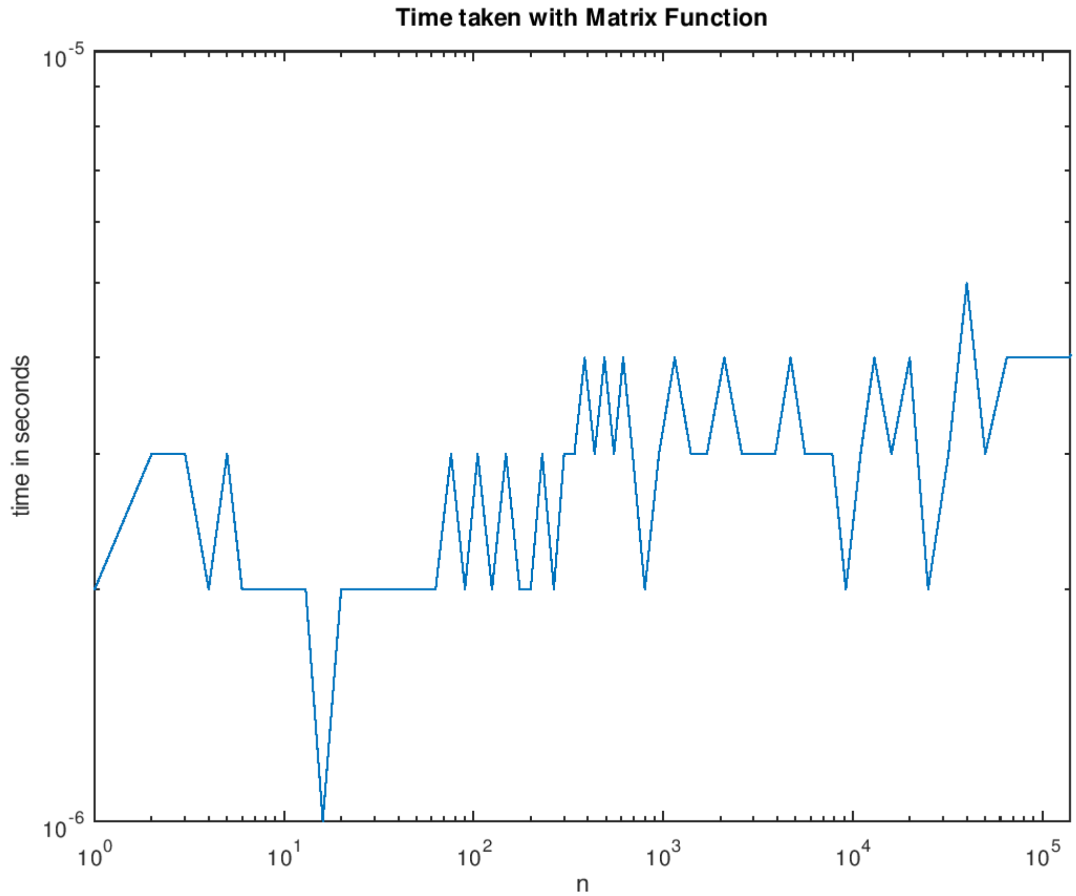
According to the figures above and below, the recursive function takes much more time as n becomes larger (exponential growth). Thus, we can say that $T(n) = \Omega(2^{n/2})$.
The growth of time taken for bottom-up function becomes linear as n becomes larger. So, $T(n) = \Theta(n)$ is correct.
The growth of time taken for the rest two functions becomes slower and slower (almost constant) if n becomes larger and larger. So, the recurrence for both functions is $T(n) = \Theta(\lg n)$.

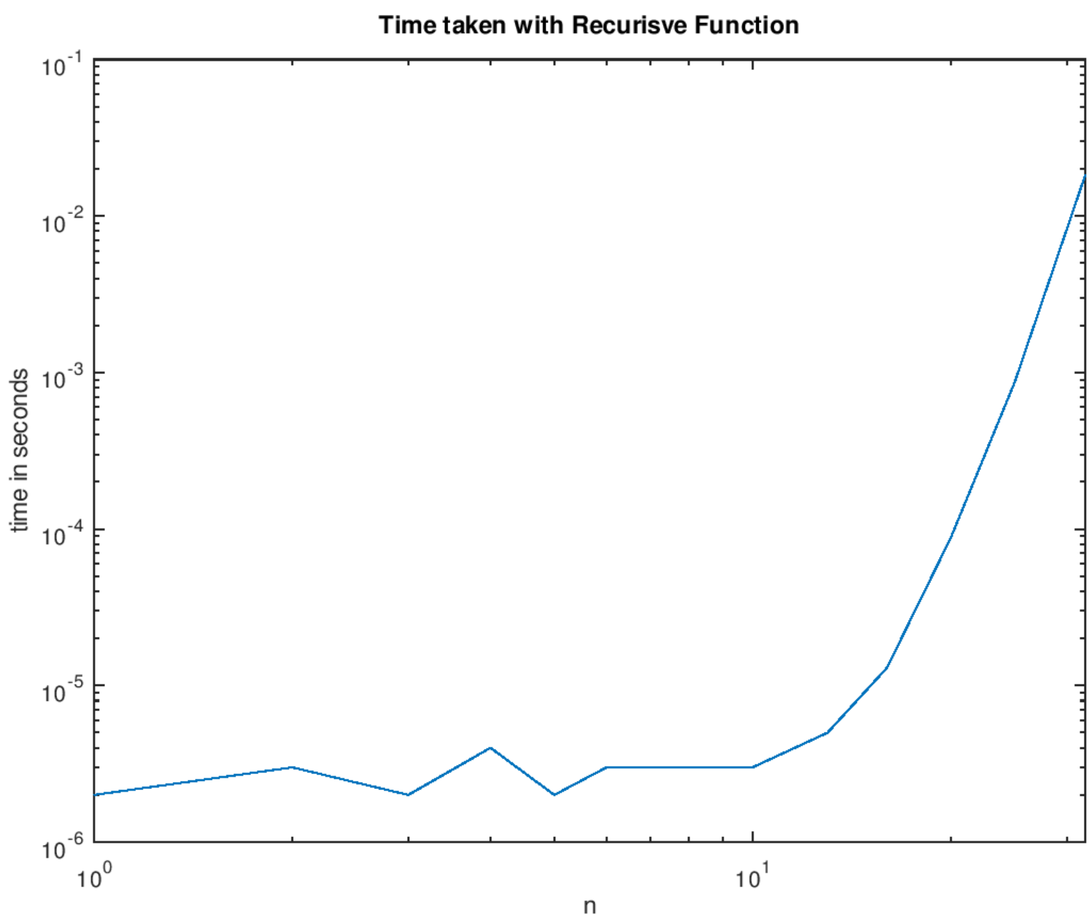


Time taken with Bottom-Up Function
time in seconds
n
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{0}
10^{1}
10^{2}
10^{3}
10^{4}
10^{5}



Time taken with Closed-Form Function
time in seconds
n
10^{-4}
10^{-5}
10^{0}
10^{1}
10^{2}
10^{3}
10^{4}
10^{5}


Time taken with Matrix Function
time in seconds
n
10^{-5}
10^{-6}
10^0
10^1
10^2
10^3
10^4
10^5


Time taken with Recurisve Function
time in seconds
n
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^0
10^1

Problem 3.2

a) Multiplying two integers in terms of bits is comparing the right-most bit of second integer with the first one with & operation, and right-shifting until the comparison with the left-most bit has been done. Then, all comparison results are added back to get the addition result.

For example,
Let a = 23, b= 45;
a*b = 575;
a in binary → 10111, b in binary→ 11001

```
              1 0 1 1 1
            * 1 1 0 0 1
-----------------------------
              1 0 1 1 1
            0 0 0 0 0
          0 0 0 0 0
        1 0 1 1 1
    +  1 0 1 1 1
-----------------------------
     1 0 0 0 1 1 1 1 1 1
```

One multiplication or addition takes $\Theta(n)$ times. In this case, the multiplication takes n times, so the time complexity for all multiplication is $\Theta(n^2)$.
Also, the time complexity for each addition is $\Theta(n)$, so the time complexity for the whole addition process is $\Theta(n^2)$. Since we have to right-shift the bits for the second integer for n times (until the left-most element becomes neutral(0)), the time complexity for this shifting process is also $\Theta(n)$.
So, the total time taken for addition and multiplication is $\Theta(2n^2+n)$ operations. We can ignore the constant 2 and small n for calculating the time complexity, so the total time taken is $\Theta(n^2)$.

b) Divide and Conquer algorithm
Let a = 23, b = 25;
a = (2*10) + 3; b = (2*10)+5;
a*b = [(2*10)+3]*[(2*10)+5] = (4*10*10) + (10*10) + (6*10) + 15 = 575

Let a = 2525; b = 3515;
a = (25*100)+25; b = (35*100)+15;
a*b = [(25*100)+25]*[(35*100)+15]
= (25*35*100*100) + (25*100*15) + (25*35*100) + (25*15)
= $(10^4 * 25*35) + (10^2 * 25*15) + (10^2 * 25*35) + (25*15)$
= 8875375
We can still spread the expression of a and b to
a = (2*1000)+(5*100)+(2*10)+5;   b = (3*1000)+(5*100)+(1*10)+5, and multiply.

We can also apply this concept to the binary numbers.
If the number of bits for integers (n) is power of 2, we can divide the binary numbers into
$a = al*2^{n/2} + ar$ (al - leftmost n/2 bits of a, ar - rightmost n/2 bits)

$b = bl*2^{n/2} + br$ (bl - leftmost n/2 bits of b, and br - rightmost n/2 bits)

$a*b = (al*2^{n/2} + ar)*(bl*2^{n/2} + br)$
$= 2^n (al*bl) + 2^{n/2}(al*br+ar*bl) + (ar*br)$
The recurrence in this case will be $T(n) = 4T(n/2) + O(n)$, and time complexity is $\Theta(n^2)$ <By Master Method>.
According to the equation above, there will be 4 recursive calls: (al*bl), (al*br), (ar*bl), and (ar*br).

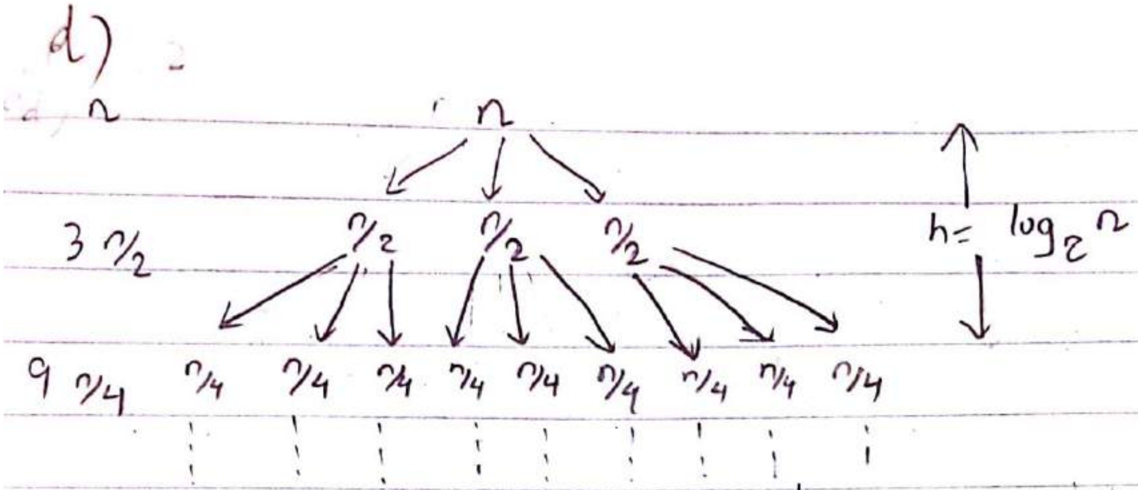
$al*br + ar*bl = (al+ar)(bl+br) – (al*bl) – (ar*br)$
$a*b = 2^n(al*bl) + 2^{n/2} [(al+ar)(bl+br) – (al*bl) – (ar*br)] + (ar*br)$
In this case, there will be only 3 recursive calls: (al*bl), (al+ar)*(bl+br), and (ar*br).
So, the recurrence will become $T(n) = 3T(n/2) + O(n)$.

<Master Method>
Let $f(n) = O(n)$; b=2; a=3;
$n^{\log_b a} = n^{\log_2 3} = n^{1.58}$ (approximately)
If $\varepsilon = 0.58$, $n^{\log_b a-\varepsilon} = n^1$, and $f(n) = O(n)$.

So, $T(n) = \Theta(n^{\log_2 3})$

c) The recurrence for Divide and Conquer algorithm: $T(n) = 3T(n/2) + O(n)$.

d)



$$\Sigma = n + 3\frac{n}{2} + 9\frac{n}{4} + \dots = \sum_{k=0}^{h} \frac{3^k n}{2^k}$$

$$= \frac{\frac{1-3^{h+1}}{-2}}{\frac{1-2^{h+1}}{-1}} \cdot n = \frac{3^{h+1}-1}{2(2^{h+1}-1)} \cdot n = \frac{3^{\log_2 n + 1}}{2^{\log_2 n + 2}} \cdot n \quad \langle \text{ignore } -1 \rangle$$

$$= n \cdot \frac{3^{\log_2 n}}{2^{\log_2 n}} = \frac{3^{\log_2 n}}{n} \cdot n = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1.58}$$

e) $T(n) = 3T(n/2) + \Theta(n)$

Let $a = 3$, $b = 2$, $F(n) = \Theta(n^1)$

$n^{\log_b a} = n^{\log_2 3} \approx n^{1.58}$

IF $\varepsilon = 0.58$, $F(n) = O(n^{\log_b a - \varepsilon}) = O(n^{1.58 - 0.58}) = O(n)$

$\therefore T(n) = \Theta(n^{\log_2 3}) \approx \Theta(n^{1.58})$

In e), since f(n) = Θ(n), we can also say that O(n) = f(n).