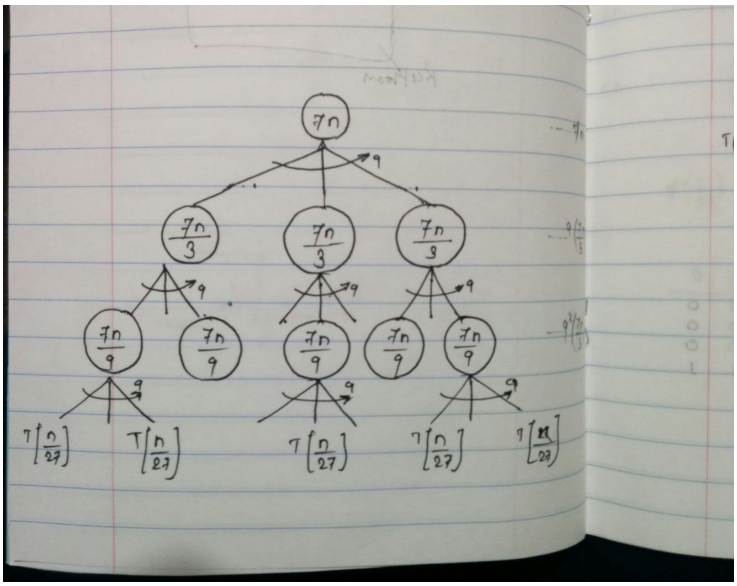
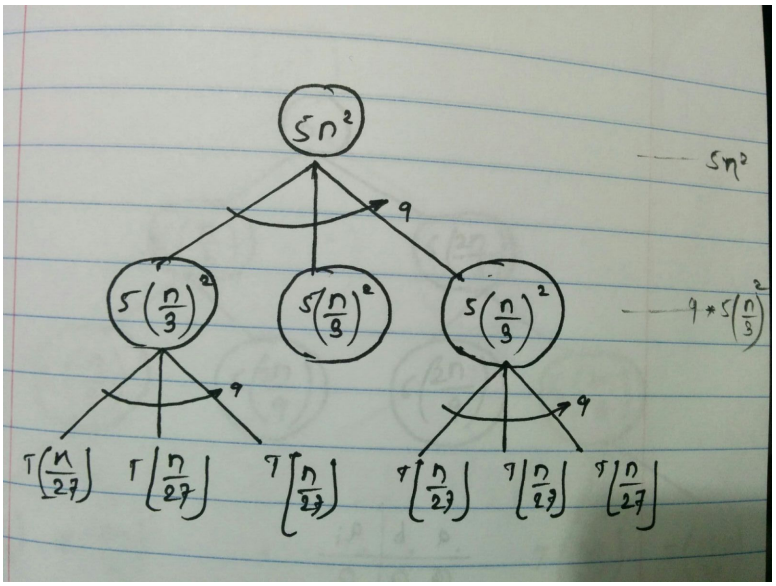
1 (a)



This factorization will go till $(n/3^i)$ does not become **1**.
If $n/3^i = 1$ then $n = 3^i$ *therefore* $\underline{i = \log_3(n)}$
Hence, there will be *'i'* ($\underline{\log_3(n)}$) terms in the end.

$$
\begin{aligned}
T(n) &= 7n + 9^1*(7n/3) + 9^2*(7n/3) + \ldots + 9^i*(7n/3) \\
&= 7 * [n + 9*(n/3) + 9^2*(n/3^2) + \ldots + 9^i*(n/3^i)] \\
&= 7n * [1 + 3 + 3^2 + \ldots + 3^i] \\
&= 7n * [1 + 3 + 3^2 + \ldots + 3^{\log_3(n)}] \\
&= 7n * [1 + 3 + 3^2 + \ldots + n] \\
&= 7n^2 * [1/3^i + 1/3^{i-1} + 1/3^{i-2} + \ldots + 1] \\
&= 7n^2 * [\ 1/(1-(1/3))\ ]\ ] \\
&= 7n^2 * [\ 3/(3-1)\ ] \\
&= 7n^2 * [\ 3/2\ ] \\
&= \theta\ (n^2)
\end{aligned}
$$

1 (b)



This factorization will go till ***(n/3^i)*** does not become **1**.
If ***n/3^i = 1*** then ***n = 3^i*** *therefore* ***<u>i = log$_3$(n)</u>***
Hence, there will be *'i'* (*<u>log$_3$(n)</u>*) terms in the end.

$$T(n) = 5n^2 + 9^1*5(n/3)^2 + 9^2*5(n/3^2)^2 + \ldots + 9^i*5(n/3^i)^2$$

$$= 5 * [n^2 + 9*(n^2/9) + 9^2*(n^2/9^2) + \ldots + 9^i*(n^2/9^i)]$$

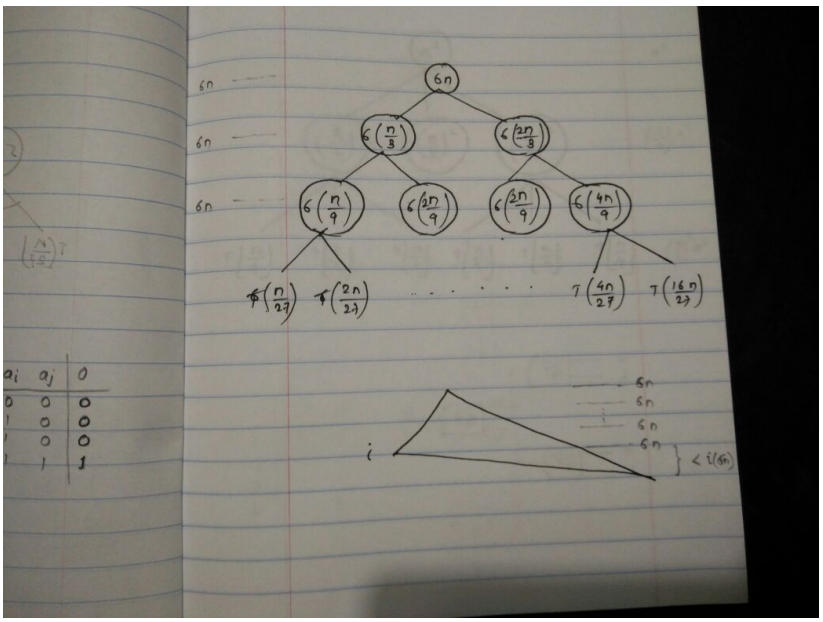
$$= 5n^2 * [1 + 1 + 1^2 + \ldots + 1^i] \quad \ldots(i \text{ times})$$

$$= 5n^2 * [1 + 1 + 1^2 + \ldots + 1^{\log_3(n)}]$$

$$= 5n^2 * [\log_3(n)]$$

$$= \theta\,(n^2 \log_3(n))$$

1 (c)



We can see that this factorization will be skewed instead of being symmetric. The (2n/3) factor will take longer time to become 1 as compare to the (n/3) term.
(n/3) term will become 1 when $n/3^i = 1$, then $n = 3^i$ *therefore* $\underline{i = \log_3(n)}$
From the diagram, it is clear for the first 'i' terms will give out **6n**.
Thus, at the end of first 'i' terms addition would be $6n*i = 6n * \log_3(n)$
For the remaining terms the addition would not go beyond addition of first 'i' terms and thus 2 times value of addition of first 'i' terms would upper bound.
Thus,

$T(n) = 6n * \log_3(n) + \text{Constant}$

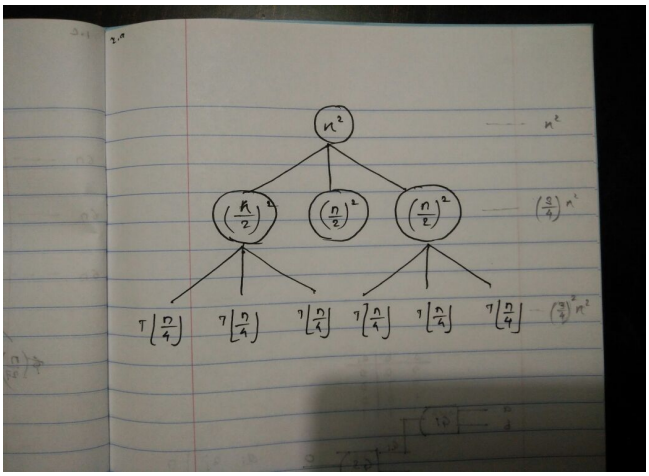
$<= 2 * 6n * [\log_3(n)]$

$<= 12n * [\log_3(n)]$

$<= 12n[\log_3(n)]$

$= \theta(n\log_3(n))$

2 (a)



T(n) = $3T\lfloor n/2 \rfloor + n^2$

Will prove this by induction.

For base case :

T(1) = 1 -----<1>

T(2) = $3T(1) + (2^2)$ = 3 + 4 = 7 -----<2>

T(3) = $3T(1) + (3^2)$ = 3 + 9 = 12 -----<3>

For smaller values of 'n' we can say that

T(1) < T(2) < T(3)

For larger values :

1 < k < n

T(k) = $3T\lfloor k/2 \rfloor + k^2$ ---<4>

T(k + 1) = $3T\lfloor (k+1)/2 \rfloor + (k+1)^2$ = $3T\lfloor (k+1)/2 \rfloor + (k^2 + 2k + 1)$ ---<5>

T(k - 1) = $3T\lfloor (k-1)/2 \rfloor + (k-1)^2$ = $3T\lfloor (k+1)/2 \rfloor + (k^2 - 2k + 1)$ ---<6>

Now let`s compare equation 4,5 and 6

Since we are taking the floor of the values,

thus $\lfloor k/2 \rfloor <= \lfloor (k+1)/2 \rfloor$ and same way $\lfloor k/2 \rfloor >= \lfloor (k-1)/2 \rfloor$ ---<7>

For 4, 5

T(k) ? T(k + 1)

Using equation 7, to compare functions we just need to compare the right side.

k^2 ? $k^2 + 2k + 1$

0 < 2k + 1

T(k) < T(k + 1) ---<8>

For 5, 6

T(k) ? T(k - 1)

Using equation 7, to compare functions we just need to compare the right side.

k^2 ? $k^2 - 2k + 1$

0 < - 2k + 1

2k > 1

T(k) > T(k - 1) ---<9>

Thus, from 8 and 9 we can say that,

T(k - 1) < T(k) < T(k + 1)

As the value of function increases with the increase in value of 'n',

we can say that the function is non-decreasing.

2 (b)

$T(n) = 3T\lfloor n/2 \rfloor + n^2$

This factorization will go till ***(n/3^i)*** does not become **1**.
If ***n/2^i = 1*** then ***n = 2^i*** *therefore* ***i = log$_2$(n)***
Hence, there will be *'i'* (*log$_2$(n)*) terms in the end.

$T(n) = n^2 + (3/4)n^2 + (3/4)^2n^2 + (3/4)^3n^2 + \dots + (3/4)^in^2$
$T(n) = n^2 * [1 + (3/4) + (3/4)^2 + (3/4)^3 + \dots + (3/4)^i]$
$T(n) <= n^2 * 2$
$T(n) <= 2n^2$
$T(n) = \theta(n^2)$

Thus, we need to prove that there exists $C_1, C_2 > 0$ and $n_o > 0$, such that for all values of $n > n_o$,
Please note that value of n_o would be different for left and right hand side of the below equation.

$C_1(n^2) <= T(n) <= C_2(n^2)$

Left Part:-
For base case :

$T(1) >= C_1(1^2)$
$T(1) >= C_1$ and from given we know that $T(1) = 1$
Therefore, for above equation to hold true we need $C_1 <= 1$ and $n_o > 0$ …<1>

For higher values :

Assume a variable $m >= 2$
$T(m) = 3T\lfloor m/2 \rfloor + m^2$ (given)
$T(m) >= C_1m^2$ (to prove)
Therefore, we need to prove
$3T\lfloor m/2 \rfloor + m^2 >= C_1m^2$
(by mathematics *m/2 < m/3*)
Since, we have to prove the left-hand side of the equation as greater, it`s okay to reduce the left side.
Thus, we can replace $3T\lfloor m/2 \rfloor$ by $3T(m/3)$ without changing the signage.
$3T(m/3) + m^2 >= C_1m^2$
Since, $T(m) >= C1m^2$ again we can apply this in the left-hand side of the equation
$3\ C1(m/3)^2 + m^2 >= C_1m^2$
$C1(m^2/3) + m^2 >= C_1m^2$
$C1m^2(1/3 - 1) >= -m^2$
Divide both sides by m^2 the equation
$C1(1/3 - 1) >= -1$
$C1(-2/3) >= -1$
Divide both sides by -1 the equation, this will change the signage.
$C1*(1/1.5) <= 1$
Multiply both sides by 1.5 the equation
$C1 <= 1.5$ …<2>

From <1> and <2> we can say that there exists a constant C1 <=1 and $n_o > 0$ such that for all values of $n > n_o$, the equation $C_1(n^2) <= T(n)$ holds true.

Thus we can say that $T(n) = \Omega(n^2)$ …<3>

Right Part:-
For base case :

$T(1) <= C_2(1^2)$
$T(1) <= C_2$ and from given we know that $T(1) = 1$

Therefore, for above equation to hold true we need $C_2 >= 1$ and $n_o > 0$ …<4>

For higher values :

Assume a variable m >= 2

$T(m) = 3T\lfloor m/2 \rfloor + m^2$ (given)

$T(m) <= C_2m^2$ (to prove)

Therefore, we need to prove

$3T\lfloor m/2 \rfloor + m^2 <= C_2m^2$

(by mathematics $\lfloor m/2 \rfloor <= m/2$)

Since, we have to prove the left-hand side of the equation as smaller, it`s okay to increase the left side by small value.

Thus, we can replace $3T\lfloor m/2 \rfloor$ by 3T(m/2) without changing the signage.

$3T(m/2) + m^2 <= C_2m^2$

Since, $T(m) <= C2m^2$ again we can apply this in the left-hand side of the equation

$3\ C_2(m/2)^2 + m^2 <= C_2m^2$

$C_2\ (3m^2/4) + m^2 <= C_2m^2$

Divide both sides by m^2 the equation

$C_2\ (3/4 - 1\ ) <= -1$

$C_2\ (-1/4\ ) <= -1$

Divide both sides by -1 the equation, this will change the signage.

$C_2*(1/4) >= 1$

Multiply both sides by 4 the equation

$C_2 >= 4$ …<5>

From <1> and <2> we can say that there exists a constant $C_2>=4$ and $n_o > 0$ such that for all values of $n > n_o$, the equation $C_2(n^2) >= T(n)$ holds true.

Thus we can say that $T(n) = O(n^2)$ …<6>

Thus, from <3> and <6> $T(n) = \theta(n^2)$

Master Theorem:

a = 3 , b = 2, f(n) = n2

Now, for the other side,

$f(n) >= c_1.a\ f(n/b)$

$n2 >= c_1.3\ f(n/2)$

$n2 >= c_1.3.n2/4$

$c1=1+\alpha$

$n2 >= (1+\alpha).3.n2/4$

Dividing by n2 on both sides,

$1/3 >= \alpha$

as $c1 = 1+ \alpha$,

$4/3 >= c_1$

$f(n) >= c_1.a\ f(n/2)$

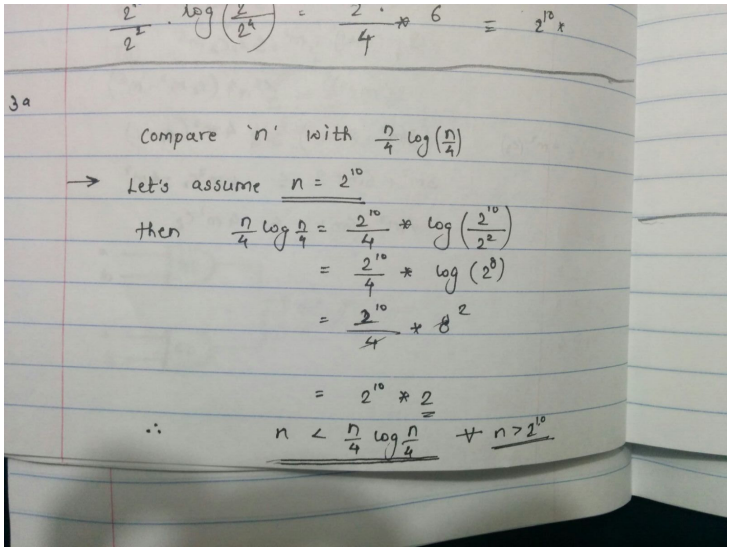
$T(n) = \theta(n2)$

3 (a) Disproved.

There exists a comparison algorithm that returns first [n/4] elements of an array of size n in the sorted order in linear time.

> This can be done in two steps.
1) Find the element of rank (n/4 + 1)
2) Shift all the elements smaller than selected element to the left and sort the first (n/4) elements.

For the first step, we can use linear selection algorithm to find an element of rank (n/4 + 1), thus the running time for this step would be '*n*'.
Now for the second step the most efficient sorting algorithm (comparison based - merge, quick) would take n*log(n) (for array of length n) time. To sort array of length n/4, the algorithm would take (n/4)*log(n/4) time and to get the first n/4 elements we would compare at maximum n-1 times.
Thus the total running time of the algorithm would be [n + n + (n/4)*log(n/4)]
At very large values of 'n', the time complexity of second step would supersede the first one, thus we can say the running time of the algorithm would be (n/4)*log(n/4). (Please refer image)
This is clearly not a linear running time.
Hence, there does not exists a comparison algorithm that returns first quarter elements in the sorted order.



3 (b) Prove.

There exists a comparison algorithm that returns first $\sqrt{n}$ elements of an array of size n in the sorted order.

> This can be done in two steps.

1) Find the element of rank ($\sqrt{n}$ + 1), let`s call this element as 'k'
2) Shift all the elements smaller than k to the left of k.
3) Sort the first ($\sqrt{n}$) elements.

For the first step, we can use linear selection algorithm to find an element of rank (n/4 + 1), thus the running time for this step would be '*n*'.

In the second step, in worst case we shall compare all n-1 elements with the k and then shift them to left if they are smaller. Therefore, this step would require (n-1) comparisons. So, we can say that running time for this step would be '*n*'.

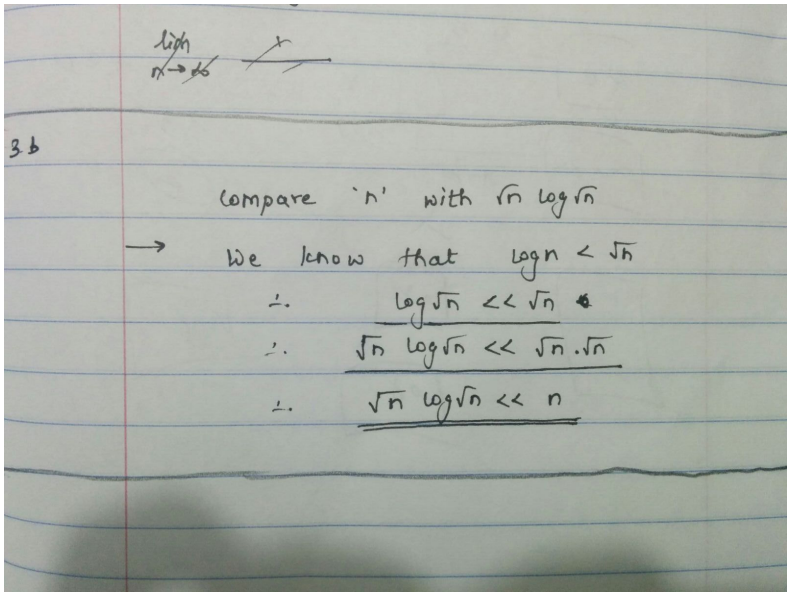
Now for the third step the most efficient sorting algorithm (comparison based - merge, quick) would take n*log(n) (for array of length n) time. To sort array of length $\sqrt{n}$, the algorithm would take ($\sqrt{n}$)*log($\sqrt{n}$) time.

Thus, the total running time of the algorithm would be [n + n + ($\sqrt{n}$)*log($\sqrt{n}$)]

Even at very large values of 'n', the time complexity of third step would not supersede the first two steps, thus we can say the running time of the algorithm would be (2n). (Please refer image)

This is clearly a linear running time.

Hence, there does exists a comparison algorithm that returns first $\sqrt{n}$ elements in the sorted order in linear time.



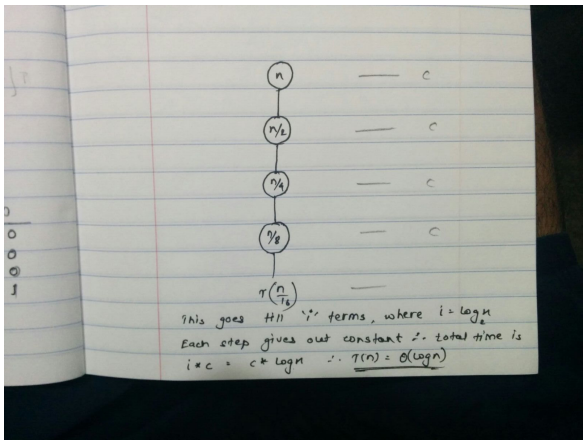
4 Selection from two sorted lists.

→ Pseudo code for the function that returns selection from two lists.

```
function search-k ( a, b, k ):
If      k == 1
        return    min (a[0], b[0])                    ... STEP1
else
        i = min (k/2, a.length)
        j = min (k/2, b.length)
        if      a[i] < b[j]
                search-k ( a[i ... a.length], b[0 ... j], k-i )    ... STEP2
        else if a[i] > b[j]
                search-k ( a[0 ... i], b[j ... b.length], k-j )    ... STEP3
```

Logic:

1 If size of k is 1 then, just compare the first elements of both the arrays and return the smallest.
2 Otherwise compare elements in the index i, j which are nothing but minimum of k/2 and length of the arrays, respectively.
3 Then all ignore the elements that are on the left-hand side of the smallest element and all the elements right-hand side of the biggest element.
4 Decrease the value of k by the index of smallest between (i, j)
5 Then call recursive function on the remaining array.



Running Time:

In the above algorithm, all the steps except for the step 2 and 3 would be done in constant time. Step two and three both divides the splits the array into half and then starts computation on the middle portion. In the worst case that is case when value of k = 2n = a.lenght + b.length at that time in every iteration step 2 or 3 will remove k/2 = 2n/2 = n elements.
Thus, in every pass we divide the problem into two halves. Therefore, the running time of the algorithm can be given as follows –

T(n) = T($\lfloor n/2 \rfloor$) + constant.

If we solve this recursion will get that

T(n) = θ(log(n))

5 Hidden surface removal.

→ Pseudo code for the function that returns hidden surface removal.

1. Sort the lines according to the slopes.
2. Now keep on dividing the total number of lines into n/2 lines, till final number of lines in a sector are less than equal to 3.
3. Once you have 3 lines then, line with the highest y co-ordinate is marked as visible.
4. Now while combining lines check the highest points at the intersection and mark the line visible.

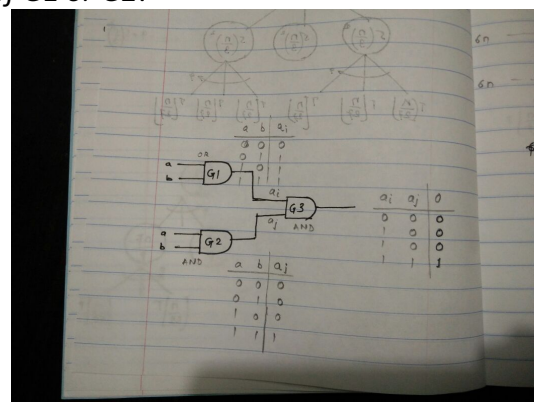
6 (a) Design a 1-AND using smallest number of gates.

→ # Gates: - ***3***

Correctness: -

There are three unique cases in which the diagram can have OR gate.

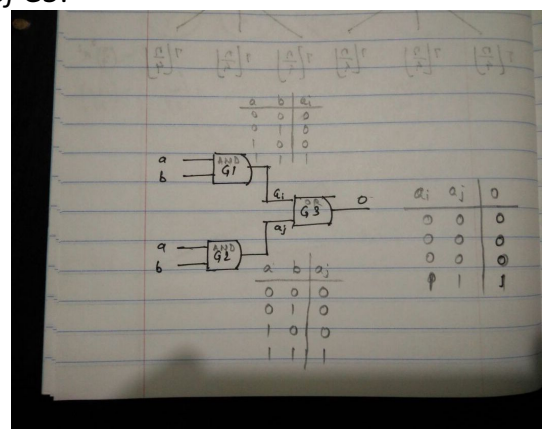
1) *OR gate in place of G1 or G2*:



In both the cases the OR gate would provide (a + b) as an output, whereas the other AND gate would provide (a*b) as an output. Now G3 is again an AND gate thus it will multiply the outputs of two previous gates. Thus, final output would be (a*b) * (a + b) = $a^2b + b^2a$, which is product of input terms a and b.

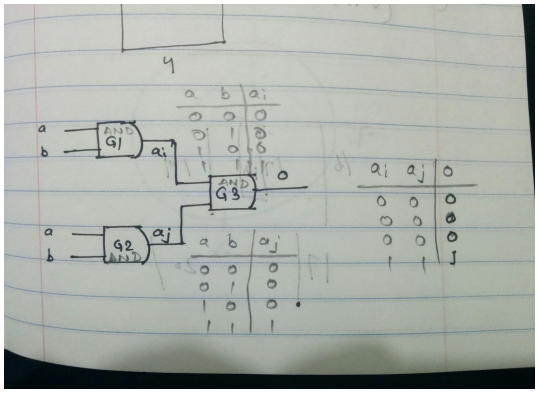
Please refer diagram for truth table and gates structure.

2) *OR gate in place of G3*:



In this both the G1 and G2 would be AND gates thus they would provide (a*b) as an output. Whereas, G3 will add the results of the previous AND gates. Thus, final output would be (a*b) + (a*b) = $\underline{2ab}$, which is product of input terms a and b.
Please refer diagram for truth table and gates structure.

3) No OR gate at all:



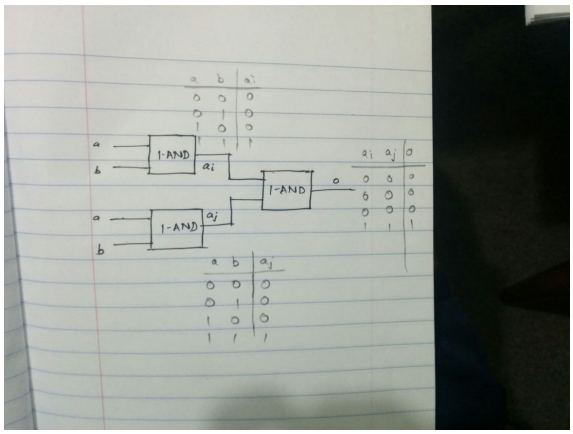
In this G1 and G2 would provide (a*b) as an output which will be multiplied by gate G3. Final output would be (a*b) * (a*b) = $\underline{a^2b^2}$, which again is nothing but product of input terms a and b.

6 (b) Using 1-AND circuit as a black box, design a 2-AND circuit.

→

$\underline{\text{\# Gates}}$: - 3*3 = ***9***

$\underline{\text{Diagram}}$: -



$\underline{\text{Correctness}}$: -

There are three unique cases in which the diagram can have OR gate.

1) B1 or B2 is faulty and acts as an OR gate:
In both the cases the OR gate would provide (a + b) as an output, whereas the other AND gate would provide (a*b) as an output. Now B3 is again an AND gate that will multiply the outputs of two previous gates. Thus, final output would be (a*b) * (a + b) = $\underline{a^2b + b^2a}$, which is product of input terms a and b.
Please refer diagram for truth table and gates structure.
2) B3 is faulty and acts as an OR gate:
In this both the B1 and B2 would be AND gates thus they would provide (a*b) as an output. Whereas, B3 will add the results of the previous AND gates. Thus, final output would be (a*b) + (a*b) = $\underline{2ab}$, which is product of input terms a and b.
Please refer diagram for truth table and gates structure.
3) No OR gate at all:
In this B1 and B2 would provide (a*b) as an output which will be multiplied by last black box B3. Final output would be (a*b) * (a*b) = $\underline{a^2b^2}$, which again is product of input terms a and b.
Please refer diagram for truth table and gates structure.

6 (c) Generic approach for k-AND circuit.
(Collaborators – Jhalak, Prasad, Sourabh)

→

We can build k-AND circuit using 3 k/2 circuit. Please refer diagram for the circuit structure. In k-AND gates we have at max k OR gates. And to convert any of the 3 k/2-AND gate into OR we just need k/2 + 1 OR gate in one AND box. Once k/2 + 1 are gone into one AND gate, only k/2 -1 OR gates will remain. And we cannot convert any of the remaining k/2-AND into OR using those OR gates.
As we will have only one k/2-OR gate in 3 k/2 gates. The overall circuit will be AND.