Problem 1

Following are the functions in the order of growth:

1. 3
2. $log3\,(n) \quad (Base\ 3)$
3. $n^{\frac{1}{3}}$
4. $log(n)$
5. $log(n^3)$
6. $n/log(n)$
7. $(log(n))^3$
8. $n - log(n)$
9. $n * log(n)$
10. n^2
11. $(n^2) * log(n)$
12. n^3
13. $100^{100} * n^2 + n^3$
14. $8^{log\ n}$
15. 8^{n-1}
16. 2^{n^3}
17. 2^{3*n}
18. 3^{2*n}
19. $(log(n))^{log(n)}$
20. $n!$
21. n^n

Problem 2

- Following is the Pseudo Code for the problem:
  int n; // number of computers;
  int [ ][ ] pairs; // storing the pairs of computer network connections, a 2d array
  int [ ] solution; // will contain the smallest number of computers that should be trusted
  Pairs will be a 2d array that contains the information about the different connections of each computer. For example if we have an input of something like the following:
  1 2
  1 3
  2 3
  3 4

  let **int c** represent the number of connections obtained from the input i.e. 4.
  Here each integer represents a computer, and we will have a 2-d array matrix that will look something like this:

$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ 1 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 \end{array}$$

  Initially the 2-d array will have 0s in it and the numbers around the array represent the graph with connections.
  We will initialize the array with the provided input and it will look something like this: since each connection is bidirectional.

$$\begin{array}{ccccc} & 1 & 2 & 3 & 4 \\ 1 & 0 & 1 & 1 & 0 \\ 2 & 1 & 0 & 1 & 0 \\ 3 & 1 & 1 & 0 & 1 \\ 4 & 0 & 0 & 1 & 0 \end{array}$$

The average complexity for initializing the array to 0s would be O(n^2) where n is the number of computers and then the complexity for updating the connections to 1, would be O(count).

```
getCount( arr )
        This function returns the count of 1s in each row of the array
        for arr[0].length

Merge(A, B)
        Create a list for output C which can fit arrays inside it
        i = 0; j = 0; k = 0;
        while(i < A.size() && j < B.size())
                if (getCount(A[ i ] <= getCount(B[ j ]))
                   C[k] = A[i]; i++;
                else
                   C[k] = B[j];  j++;
                k++;
        append the remainder of the list which have not finished to C
        RETURN C

MergeSort(pairs, n):
        if(n==1) RETURN pairs
        middle = (n-1)/2 (round down)
        A = {Pairs0, Pairs1, … …. , Pairs middle}  // here A is an array of arrays, each Pairs
represent an array
        B = {Pairs middle+1, …. , Pairs n-1}  // here B is an array of arrays, each Pairs
represent an array
        As =  MergeSort(A, middle +1)
        Bs = MergeSort(B, n-middle-1)
        RETURN Merge (As, Bs)
```

MAIN( )

Once initialized the next step would be to get a count of all the 1s, which will simply be 2 times the number of connections received from the input ie 2***c**.

```
int count =2*c
   while( count <=0 )
      int[ ][ ] sortedPairs = MergeSort(pairs, pairs[0].length)
         Merge sort will sort all the rows of the arrays based on the count of 1s in each row,
```

Example:

	1	2	3	4	Counts
1	0	1	1	0	C1=2
2	1	0	1	0	C2=2
3	1	1	0	1	C3=3
4	0	0	1	0	C4=1

Sorted pairs array:

	1	2	3	4	Counts
3	1	1	0	1	C3=3
1	0	1	1	0	C1=2
2	1	0	1	0	C2=2
4	0	0	1	0	C4=1

After sorting the arrays, we will choose the connections in the computer with the highest number of connections, which in this case is Computer3 and will update all the 1s in the array to 0.
and decrease the **count,** which represents the total count of the connections by the corresponding count of the row, that is C3=3,
so, **count = count** -C3
which will give us: 8-3
so, **count =** 5

while updating the 1s in the first row to 0, we will update the corresponding pair of the connection, to 0 as well.

Example:

	1	2	3	4	Counts
3	0	0	0	0	C3=0
1	0	1	1	0	C1=2
2	1	0	1	0	C2=2
4	0	0	1	0	C4=1

after updating the elements in the first row, now also updating the corresponding elements given by [column][0] // here row is 0 as it is the first row of the array.
So for (3,1) we'll update (1,3) as well to 0 and so on.
After updating it will look something like this:

	1	2	3	4	Counts
3	0	0	0	0	C3=0
1	0	1	0	0	C1=1
2	1	0	0	0	C2=1
4	0	0	0	0	C4=0

and the count is updated:
count = count - 1-1-1
which is
count = 5-3 = 2
so, now the count is 2
Now we will place a software engineer at computer number 3 and make it a trusted computer and add that to the solution

solution.add(3)

and the loop will run again, sort the array again and perform all the operations again until the **count** has become 0.

- Here is the estimated running time complexity for the algorithm:
  Calculating complexity as of steps in the algorithm:
  1. Initializing the array to 0s was O(n^2)
  2. updating the array to 1s was O(count) where count represents the number of different connections obtained from the input like in the example in the pseudo code. (count<n)
  3. Sorting the pairs array, which is a 2d array being sorted row-wise based on the count of 1s in it. Here the merging part is O(n^2) as we are merging based on counting the number of 1s in each row which is O(n^2) . This happens for O(log n) times, which gives us a total complexity of O($n^2 log(n)$).
  4. Complexity for updating the values of 1s to 0s in the row would be the O(count of 1s in row) which is O(count) where (count<n).
  5. The sorting and updation of the array operations are performed until the **count** becomes 0. It highly depends on the types of connections input. For the worst case, the loop will have to run n/2 times to completely remove all the pairs(make the computers trusted), but for best case it will have to run once, as all the computers will be connected to only one single computer which can be made trusted.
     which will give us an overall complexity of O($n * n^2 log(n)$).

  Total complexity of the algorithm is O($n^3 log(n)$)

- Following is the counterExample:
  After going through multiple examples and brainstorming to somehow break it, the above strategy always seems to work. Although there were examples whose results could be achieved even without the strategy but the algorithm seems to work and always provides the optimal solution.

Problem 3

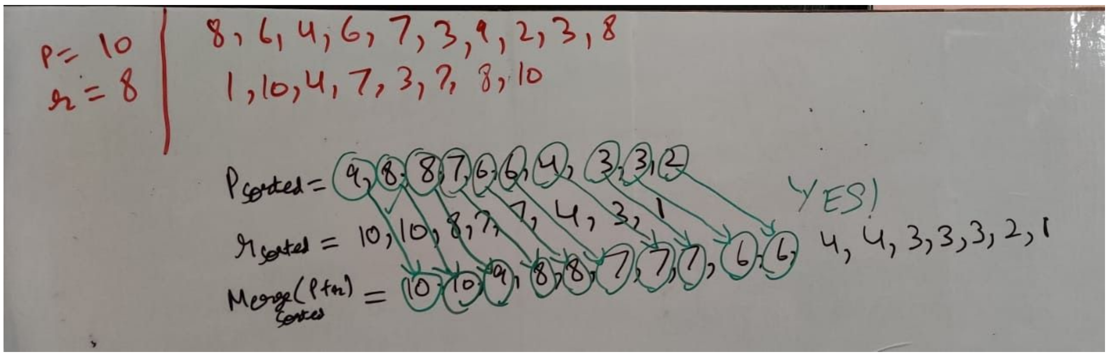
Description:
We currently have **p** plants in the existing planters that need to be shifted to planters of a bigger size. We also have **r** new planters available as mentioned in the questions that can be used to fill up the space. To solve this problem, we can sort both the planters according to their size in descending order, and we can obtain **Psorted** and **Rsorted**. Then we can merge both the sorted arrays and form a new array that is sorted that contains both the elements from **p** and **r,** which will be **MergedPlanters**. Now we can compare all the elements of the **Psorted** with the **MergedPlanters** at the same index and it should always be less than the **MergedPlanters** elements. If there is even one single element that does not follow the above condition, the shifting of planters will not be possible.

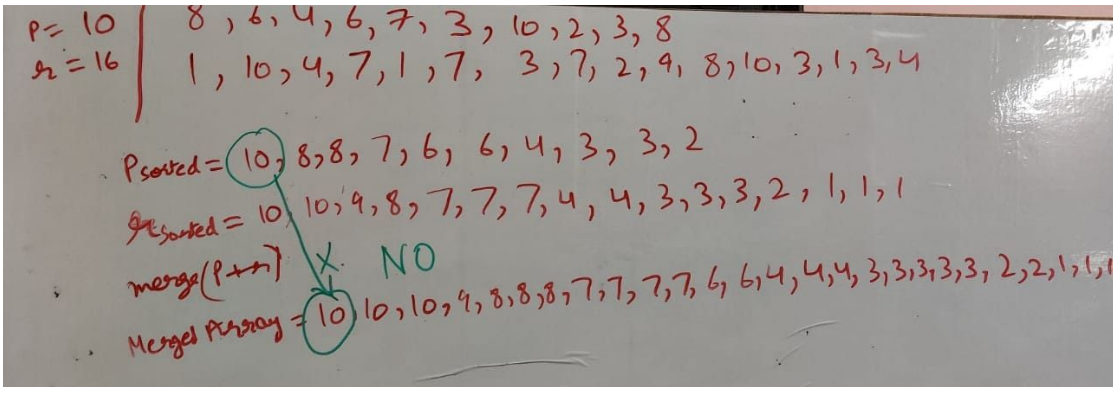
PseudoCode:

```
boolean CanBePlaced( mergedArray, initialArray )
        for i to initialArray.size
           if( initialArray[ i ] >= mergedArray[ i ])
                 return false
        // if the loop is complete and it satisfies the condition
        return true

MERGE( array, left, middle, right )
        n1= middle - left +1
        n2 = right - middle
        L[ ] = new array of size [ n1 ]
        R[ ] new array of size [ n2 ]

        Copy elements from array to L[ ] and R[ ] according to left, right and middle index
        for i to n1
           L[ i ]= array [ left+i ]
        for j to n2
           R[ j ] = array [ middle +1+j ]

        // now merging them
        i = 0
        j = 0
        k=left
        while( i <  n1 and j < n2)
                // sorting in descending order
                if L[ i ] >= R[ j ]
                   array[ k ]=L[ i ]
                   i++
                else
                   array[ k ] = R[ j ]
                   j++
        Now Append all the remaining elements in L and R to the array because they are
already sorted
```

```
SORT( array , int left, int right)
        if( left < right )
                middle = left+right/2
                SORT( array, left, middle)
                SORT( array , middle+1, right)

                MERGE( array, left, middle, right )

Main()
        SORT(p, 0, length of p)
        SORT(r , 0, length of r)
        mergedArray = p+r
        MERGE(mergedArray)
        print(CanBePlaced(mergedArray))
```

Sketch of the Algorithm:


p= 10
r = 8
8, 6, 4, 6, 7, 3, 9, 2, 3, 8
1, 10, 4, 7, 3, 7, 8, 10
Psorted = 9, 8, 8, 7, 6, 6, 4, 3, 3, 2
rsorted = 10, 10, 8, 7, 7, 4, 3, 1
Merge(P+r) Sorted = 10, 10, 9, 8, 8, 7, 7, 7, 6, 6, 4, 4, 3, 3, 3, 2, 1
YES!


p= 10
r =16
8, 6, 4, 6, 7, 3, 10, 2, 3, 8
1, 10, 4, 7, 1, 7, 3, 7, 2, 9, 8, 10, 3, 1, 3, 4
Psorted = 10, 8, 8, 7, 6, 6, 4, 3, 3, 2
rsorted = 10, 10, 9, 8, 7, 7, 7, 4, 4, 3, 3, 3, 2, 1, 1, 1
merge(P+r)
X NO
Merged Array = 10, 10, 10, 9, 8, 8, 8, 7, 7, 7, 7, 6, 6, 4, 4, 4, 3, 3, 3, 3, 3, 2, 2, 1, 1, 1

Running Time Estimate:

- Sorting the P array which is O(p log p)
- Sorting the R array which is O(r log r)
- Merging the P and R array, which is O(r+p)
- Comparing all the elements of P with the Merged array, which is O(p)

The running time estimate for the algorithm given above is O(p log p) + O(r log r)+O(p+r)+ O(p).

Problem 4

Description:
We were given a set of numbers S of size n and we need to perform the operation S+S and provide the count of all the unique elements obtained in the new set. First step would be to create all the combinations possible and store them in an array of size n^2. After finding all the combinations, we need to sort the array using merge sort of any sorting strategy. Once the array is sorted it still contains the duplicates which we need to remove in order to find the size which will be the answer. Since the array is sorted we can start from the beginning and count each unique element occurred in the array because the array is sorted we just need to do it once.

PseudoCode:

```
MERGE( array, left, middle, right )
        n1= middle - left +1
        n2 = right - middle
        L[ ] = new array of size [ n1 ]
        R[ ] new array of size [ n2 ]

        Copy elements from array to L[ ] and R[ ] according to left, right and middle index
        for i to n1
           L[ i ]= array [ left+i ]
        for j to n2
           R[ j ] = array [ middle +1+j ]

        // now merging them
        i = 0
        j = 0
        k=left
        while( i <  n1 and j < n2)
                if L[ i ] <= R[ j ]
                   array[ k ]=L[ i ]
                   i++
                else
                   array[ k ] = R[ j ]
                   j++
        Now Append all the remaining elements in L and R to the array because they are
already sorted

SORT( array , int left, int right)
        if( left < right )
                middle = left+right/2
                SORT( array, left, middle)
                SORT( array , middle+1, right)
                MERGE( array, left, middle, right )
```

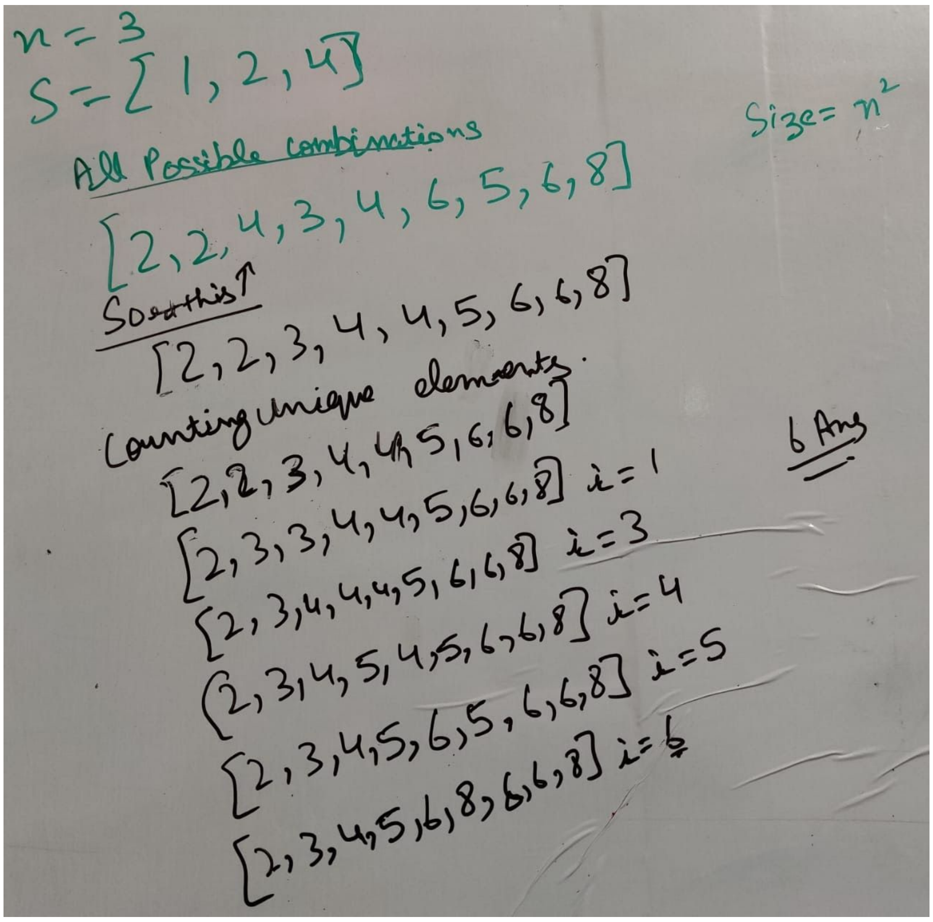
```
CreateAllPossibleCombinations(double[ ] array, double[ ] solution)
        k = 0
        for i to array.length
           for j to array.length
                 solution [ k ] = arr [ i ] + arr [ j ]
                 k++
        // solution will have all the possible combinations that can be created with the array.
```

// because the array is sorted, we just need to make sure that our adjacent elements with the same value are not counted multiple times.

```
GetCount( double [ ] solution)
        int i = 0
        for j till solution.length
                 // counts all the unique elements in the array
                 if( solution [ i ] != solution [ j ])
                    i++
                    solution[ i ] = solution [ j ]
        return i+1; // +1 for the first element
```

```
MAIN()
        CreateAllPossibleCombinations( input, possibleCombinations )
        SORT ( possibleCombinations, 0, possibleCombinations.length-1)
        print(GetCount(possibleCombinations))
```

Sketch of the algorithm:



Running Time Estimate:

- Creating all the possible combinations will require O(n*n)
- Sorting the array which is already of size n^2, will require O(n^2 log n^2) which is O(2*n^2logn).
- Now getting the count of unique elements will be O(n^2)

Overall Complexity will be: O (n^2) + O(2*n^2log n) +O (n^2)
The complexity will be given by the dominating quantity which is O (n^2 log n).