QUESTION 1

ASSUMPTIONS

- Consider the skills one by one in the order of their occurrence within the skill array.

  Example: skills= [4,5,2,3,3, -5]

  Take 4 first. Create the first team to assign 4.

  ⇨ Team 1 = {4}

  Next, take 5.

  ⇨ Team 1 = {4,5}

  Next, take 2.

  ⇨ Team 1 = {4,5,2}

  Next, take 3.

  ⇨ Team 1 = {4,5,2,3}

  Next, take 3.

  ⇨ Team 1 = {4,5,2,3}   Team 2 = {3}

  Next, take -5.

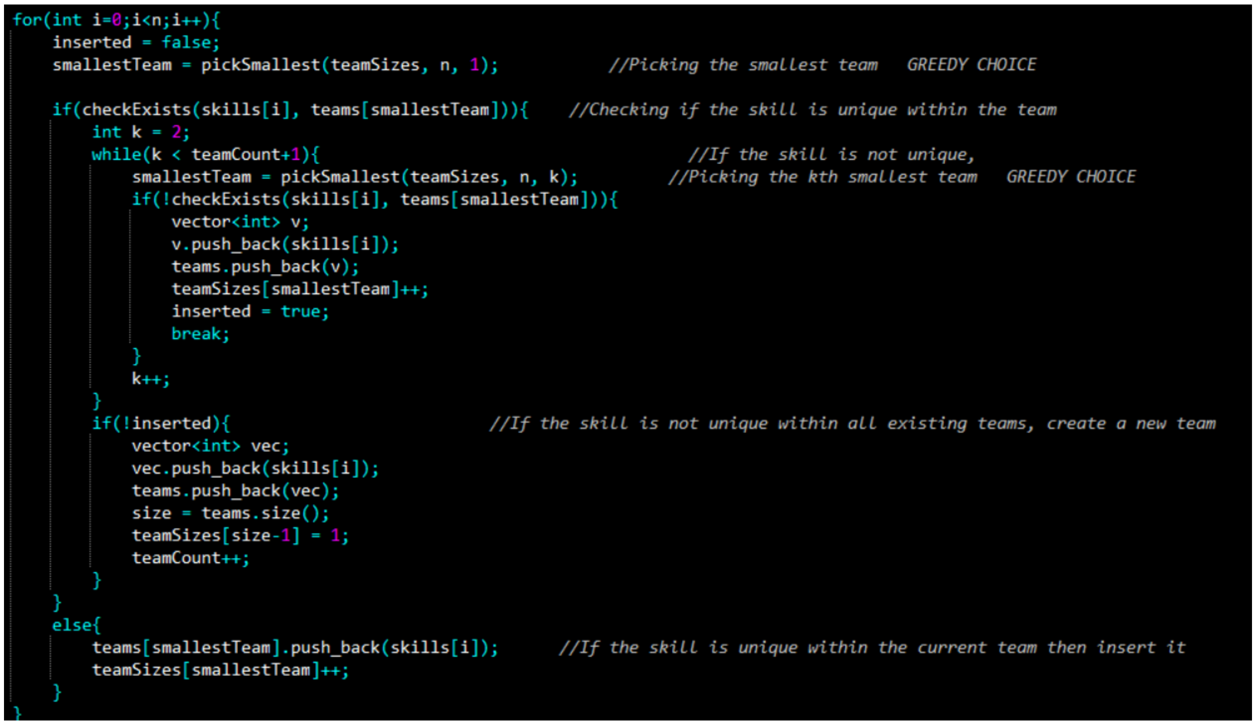
  ⇨ Team 1 = {4,5,2,3}   Team 2 = {3, -5}

  Finally, the largest possible smallest team size = 2 which has 3 and -5 as its elements.

- There is only the uniqueness constraint. The skills within a team do not have to be consecutive.
  As in the previous example, team 2 can have 3 and -5 without the skill levels in between.

GREEDY CHOICE PROPERTY

The greedy choice property in this solution is choosing the smallest team among the existing teams in each instance.



```
for(int i=0;i<n;i++){
    inserted = false;
    smallestTeam = pickSmallest(teamSizes, n, 1);          //Picking the smallest team   GREEDY CHOICE

    if(checkExists(skills[i], teams[smallestTeam])){    //Checking if the skill is unique within the team
        int k = 2;
        while(k < teamCount+1){                              //If the skill is not unique,
            smallestTeam = pickSmallest(teamSizes, n, k);      //Picking the kth smallest team   GREEDY CHOICE
            if(!checkExists(skills[i], teams[smallestTeam])){
                vector<int> v;
                v.push_back(skills[i]);
                teams.push_back(v);
                teamSizes[smallestTeam]++;
                inserted = true;
                break;
            }
            k++;
        }
        if(!inserted){                          //If the skill is not unique within all existing teams, create a new team
            vector<int> vec;
            vec.push_back(skills[i]);
            teams.push_back(vec);
            size = teams.size();
            teamSizes[size-1] = 1;
            teamCount++;
        }
    }
    else{
        teams[smallestTeam].push_back(skills[i]);      //If the skill is unique within the current team then insert it
        teamSizes[smallestTeam]++;
    }
}
```

At each instance, we pick the smallest team and check whether we can insert the current skill level to that team. If we can, we insert it, and if we cannot, then we check the second smallest team. And the process goes on.

At each instance we also check the uniqueness of the skill within the team to consider whether we can insert it to the team or not.

By inserting the skill into the smallest team at that instance, we hope to make the smallest team large as possible. Therefore, by considering the sub optimal choice, we hope to get the global optimal choice in this algorithm. This is the greedy choice property of this algorithm.

OPTIMAL SUB-STRUCTURE PROPERTY

Consider the example skills = {-1,0,1,2,2,3}. The solution for this test case would yield team 1 = {-1,0,1,2} and team 2 = {2,3} using this algorithm.

When you consider each team, you would realize that team 1 is the largest possible smallest team if you only consider the skills {-1,0,1,2} with a size 4, and team 2 is the largest possible team size if you only consider the skills {2,3} with a size 2. Each team in the final solution is the largest possible smallest team on its own. Therefore, this has the optimal sub-structure property.

Since this solution has both greedy choice property and optimal sub-structure property, we can say that this is a greedy solution (algorithm) for the problem.

QUESTION 2

There are many sequences of choices to buy 7KG sacks of paddy. The most optimum solution would be to buy the maximum quantity of paddy by paying the least amount of money.

Therefore, theory of the fractional knapsack problem can be applied here to find the optimal solution by making the best choice (sub-optimal) available at every step.

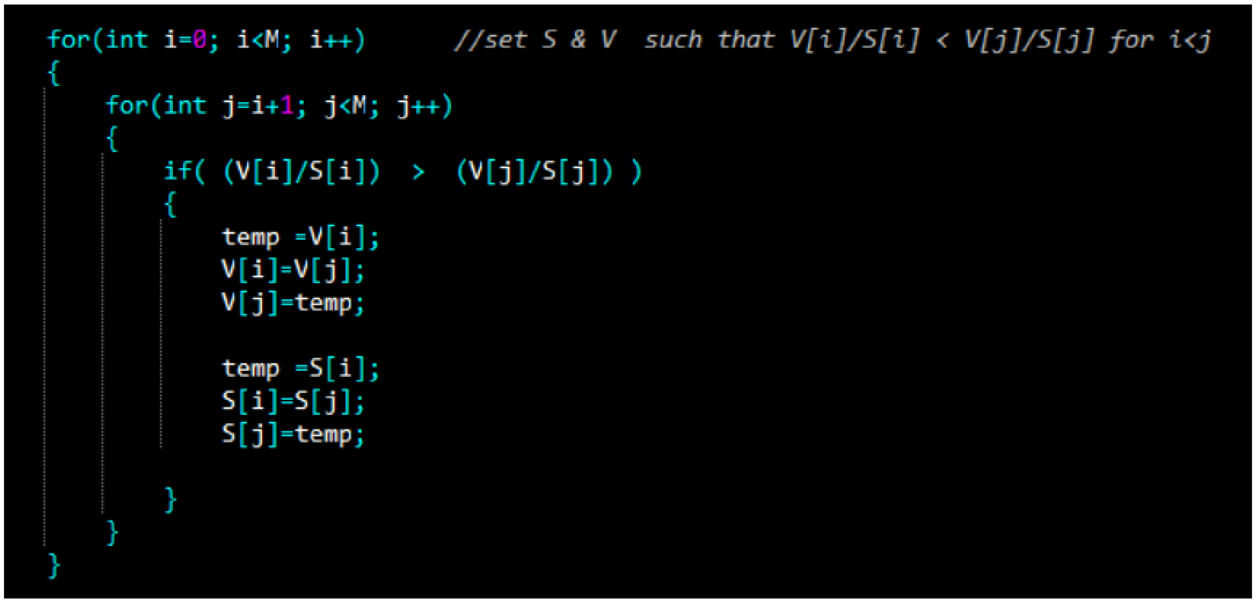
GREEDY PROPERTY

The greedy choice property in this solution is picking the item which has the least value/weight at each instance.

Sort the items in ascending order of their value/weight.

Item	1	2	3
Value	8	14	18
Weight (No. of sacks)	1	2	3
Value/Weight	8	7	6

Sorting the arrays in ascending order:

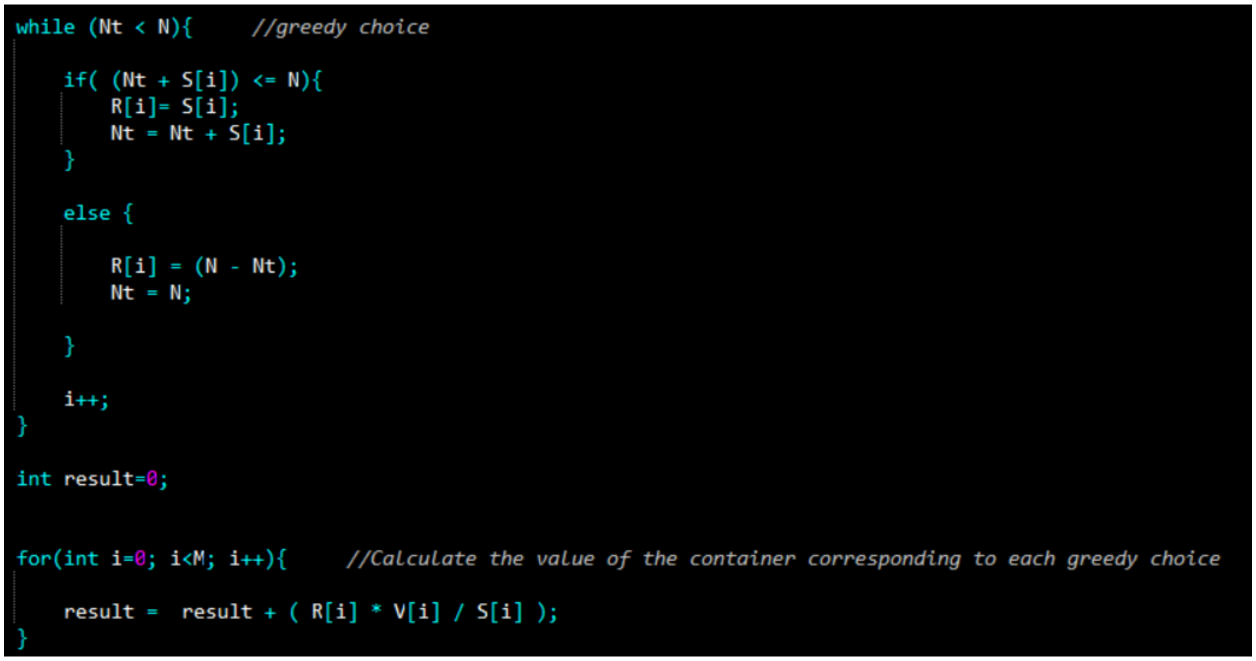


```
for(int i=0; i<M; i++)      //set S & V  such that V[i]/S[i] < V[j]/S[j] for i<j
{
    for(int j=i+1; j<M; j++)
    {
        if( (V[i]/S[i])  >  (V[j]/S[j]) )
        {
            temp =V[i];
            V[i]=V[j];
            V[j]=temp;

            temp =S[i];
            S[i]=S[j];
            S[j]=temp;

        }
    }
}
```

Greedy Choice:



```
while (Nt < N){     //greedy choice

    if( (Nt + S[i]) <= N){
        R[i]= S[i];
        Nt = Nt + S[i];
    }

    else {

        R[i] = (N - Nt);
        Nt = N;

    }

    i++;
}

int result=0;


for(int i=0; i<M; i++){     //Calculate the value of the container corresponding to each greedy choice

    result =  result + ( R[i] * V[i] / S[i] );
}
```

Let R be the optimal subset of M items which stores the number of stacks taken from each container. Let N be the number of sacks bought and let Nt be the current number of sacks in possession.

Case 1: If the number of sacks in the container + Nt <= N, then buy the sacks and increment Nt with the number of sacks bought. Store the number of sacks bought in R[i].

Case 2: If the number of sacks in the container + Nt > N, then buy the number of sacks needed to make Nt = N. Store the number of sacks bought in R[i].

Keep doing the above operations until Nt = N. Then calculate the result by getting the sum of all R[i] * value/weight of each item.

OPTIMAL SUB-STRUCTURE PROPERTY

If we take the number of sacks bought = 4, the algorithm divides the problem size into several parts starting from 0.

For example, in the algorithm, Nt value is increased from 0 to 4. Each of these can be considered as a sub-problem. At each instance, we consider the most optimal solution for the sub problem.

- Nt = 0

  We take the item with the minimum value/weight from the containers which is item number 3.

- Nt = 1
  We take the item with the minimum value/weight from the containers which is item number 3.

- Nt = 2
  We take the item with the minimum value/weight from the containers which is item number 3.

- Nt = 3
  We take the item with the minimum value/weight from the containers. Since we have exhausted item number 3 (none left), we take the item with the next minimum value/weight which is item number 2.

For the sub-problem of N=3, the optimal solution is value 18. For the sub-problem N=2, the solution is 12. For the sub-problem N=1, the solution is 6. Hence for each of these sub-problems the algorithm chooses the most optimal solution.

Therefore, this algorithm has the optimal sub-structure property.

QUESTION 3

This question is about a drinking water treatment project. For that, the Ministry of Water Resource management wants to know the least number of water treatment plants which have the ability to supply water to the entire list of cities.

There are main points we should consider before deciding the cities to build water treatment plants.

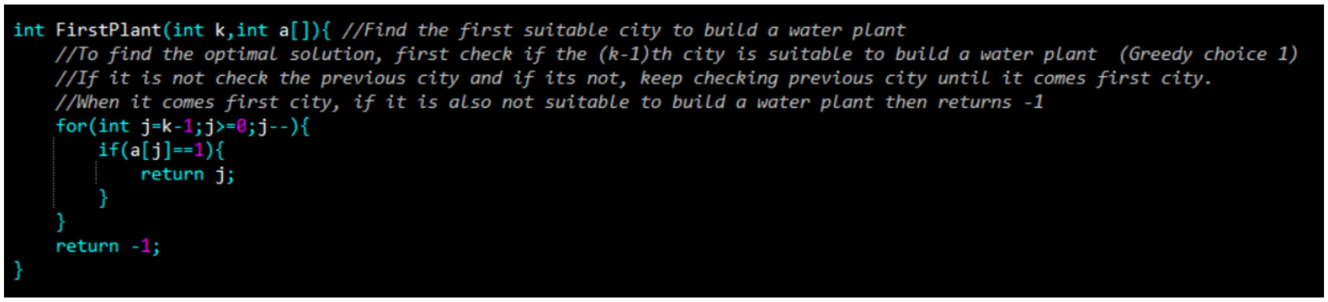
- The distance between adjacent districts is 1 unit.
- Program should return least number of water treatment plants and what they are.
- We can build a water treatment plant in city only if it is suitable.
- The distribution range limits the supply to cities where distance is less than the given k value.
- If the task can't be done, program should return -1.

Here, we should find minimum number of water treatment plants to supply water to the entire country. Therefore, we can use a greedy approach here.

GREEDY CHOICE PROPERTY

We can define two greedy choices here. First, to select first water treatment plant and other, to select the other water treatment plants.

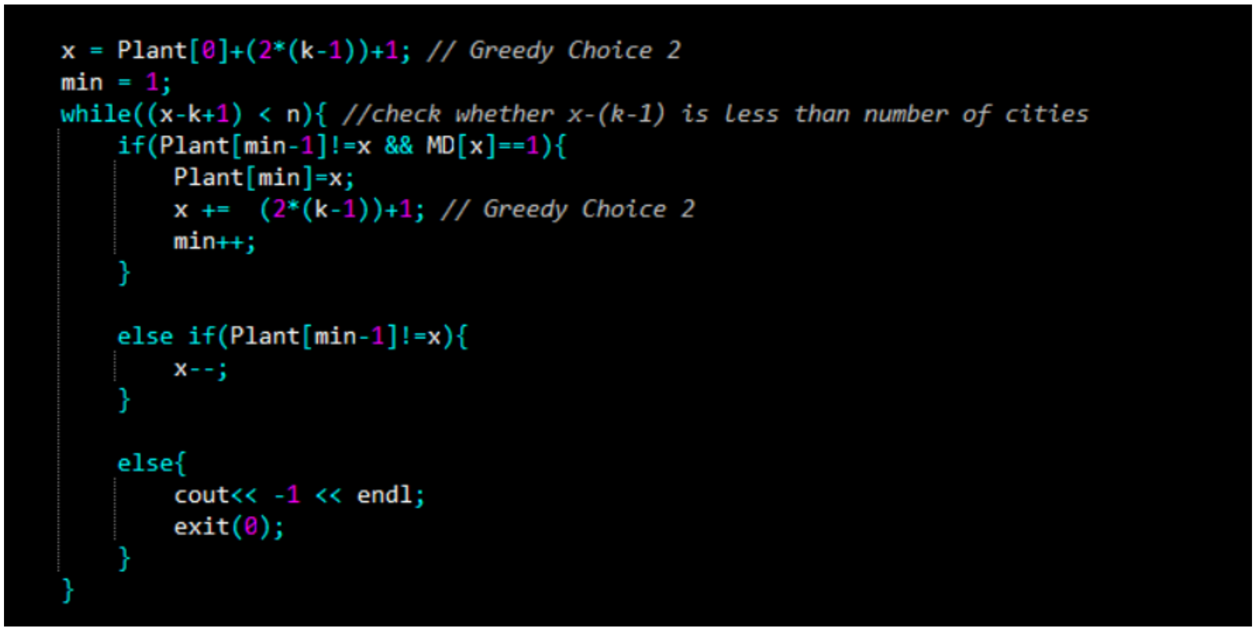
Greedy choice 1:



```
int FirstPlant(int k,int a[]){ //Find the first suitable city to build a water plant
    //To find the optimal solution, first check if the (k-1)th city is suitable to build a water plant  (Greedy choice 1)
    //If it is not check the previous city and if its not, keep checking previous city until it comes first city.
    //When it comes first city, if it is also not suitable to build a water plant then returns -1
    for(int j=k-1;j>=0;j--){
        if(a[j]==1){
            return j;
        }
    }
    return -1;
}
```

To find the optimal solution, first check whether the k-1th city is suitable to build a water plant. Here, we are getting the maximum possible distance from the first city to build a water plant. If it is not, we keep checking the previous cities until we come to the first city. When we come to the first city, if even that is not suitable to build a water plant, the program returns -1.

Greedy choice 2:



```
x = Plant[0]+(2*(k-1))+1; // Greedy Choice 2
min = 1;
while((x-k+1) < n){ //check whether x-(k-1) is less than number of cities
    if(Plant[min-1]!=x && MD[x]==1){
        Plant[min]=x;
        x +=  (2*(k-1))+1; // Greedy Choice 2
        min++;
    }

    else if(Plant[min-1]!=x){
        x--;
    }

    else{
        cout<< -1 << endl;
        exit(0);
    }
}
```

To find optimal solution, first check the 2(k-1)+1th city from previous selected water treatment plant. Here, we are getting the maximum possible distance from the previously built water plant to the next water plant.

From there onwards the same approach is followed as the previous instance where we check whether the 2(k-1)+1th city is suitable to build a water plant and if it is not, we keep on checking the previous cities until it comes to the city with the water plant we built.

OPTIMAL SUB-STRUCTURE PROPERTY

When we consider the set of cities it can be divided into sub sets considering a distance of 2(k-1)+1.

For example, let cities = {0,1,1,1,1,0}, k=2.

If we consider the sub sets, for all the subsets, the program returns the minimum number of water plants to be built to supply water to all cities.

Let c1 = {0,1,1}. For this subset, the number of water plants needed = 1 (at city 1).

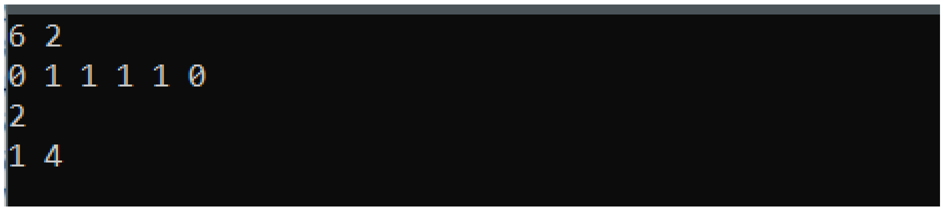
Let c2 = {0,1,1,1,1}. For this subset, the number of water plants needed = 1 (at city 1 and city 4).

Let c3 = {0,1,1,1,1,0}. For this subset, the number of water plants needed = 1 (at city 1 and city 4).

Therefore, the algorithm makes the most optimal choice for each sub problem to get the globally optimal choice. Therefore, this has the optimal sub-structure property.

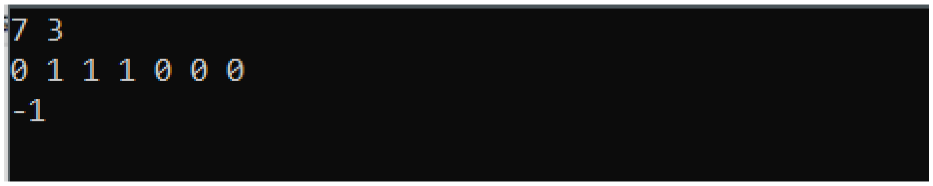
Test Cases

1)



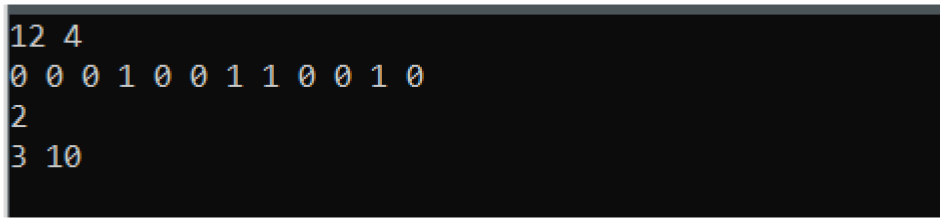
```
6 2
0 1 1 1 1 0
2
1 4
```

2)



```
7 3
0 1 1 1 0 0 0
-1
```

3)



```
12 4
0 0 0 1 0 0 1 1 0 0 1 0
2
3 10
```