1. **Placement Sort:**
   Time Complexity:
   Task1: O(n)for sorting, radix sort. Our maximum digit is 8, due to the range of package provided. So radix sort with base 10 gives linear complexity of O(n).
   Task2: O(n) Searching. Even though I am applying binary search, if the number of similar element is order of n, counting the number of occurrences will take O(n) time.
   Overall: O(n).

   Algorithms:
   Task1: Radix sort
   Task2: Binary Search

   Space Complexity:
   Task1: O(n)
   Task2: O(1)

```java
package placementsort_;

import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Scanner;

class student{
    String rn;
    String name;
    int p;
}
public class placementSort_ {

    //search
    //since this search function will get a sorted array
    //as the array is being sorted as part 1 of this question
    //we can implement binary search to search in this array

    public static int _search(int key, student[] array, int l, int r){
        int count = 0;
        int c = (l+r)/2;
        if (array[c].p==key) {
            //if the middle element is the key,
            //all other elements equal to it must be present in it's
vicinity only
            // look left, look right, count
            count++;
            for(int x =c-1; x>-1; --x) {

                if (array[x].p == key) {
                    count++;
                }
                else break;
            }
            for(int x =c+1; x<array.length; x++) {
                if (array[x].p == key) {
                    count++;
                }
```

```java
                else break;
            }
            return count;

        }
        //if one or less element remains and then also it's not a match,
        //the key is not present
        else  if(r-l<=1) return count;
        else{
            if(array[c].p>key) {
                count = _search(key, array, l, c);
            }
            else if(array[c].p<key) {
                count = _search(key, array, c, r);
            }
            return count;
        }

    }


    public static void placementSort(student[] array){

        //auxiliary space complexity = O(n)
        //at each pass n values needs to be stored
        ArrayList<ArrayDeque<student>> buckets = new ArrayList<>(10);
        for(int i = 0; i<10;i++){
            buckets.add(new ArrayDeque<student>());
        }
        //total number of passes <=8 as max value is 10^7
        for(int i = 0; i<8; i++){
            //inner loop complexity = O(n)
            int extractor = (int)Math.pow(10,i);
            for(int j = 0; j < array.length; j++){
                //extract digit at
                int x = array[j].p/extractor;
                //to the bucket no x%10
                buckets.get(x%10).addLast(array[j]);
            }

            int a = 0; //variable to loop through array

            //popping buckets and adding to array

            for(int q = 0; q<10; q++){
                //empty bucket q
                while(!buckets.get(q).isEmpty())
                    array[a++]=buckets.get(q).removeFirst();
            }
        }
        //end of outer for loop
        //complexity is O(n), as there will be 8 passes for n values



    }

    public static void main(String[] args){

        Scanner s = new Scanner(System.in);
        int n = s.nextInt();
```

```
        student[] array = new student[n];

//      getting input and constructing the student array
        for(int i=0;i<n;i++){
            student x = new student();
            x.rn = s.next();
            x.name = s.next();
            x.p = s.nextInt();
            array[i]=x;
        }


//      sorting the student array
        placementSort(array);


//      getting number of test cases to search
        int k = s.nextInt();
        int[] res = new int[k];
        //getting search test cases and passing them to the searching
algorithm
        //result is stored in the res[] array
        for(int i=0;i<k;i++){
            int key = s.nextInt();
            res[i]=_search(key,array,0,n);
        }
//        printing the outputs
        for(int i =0;i<n;i++)
            System.out.printf("%s %s
%d%n",array[i].rn,array[i].name,array[i].p);
        for(int i=0;i<k;i++){
            System.out.println(res[i]);
        }



    }
}
```

Output:

```
5
csb1 sara 3000000
csb3 lara 7000000
csb7 zara 4000000
csb6 vara 7000000
csb9 qara 5000000
3
6000000
7000000
5000000
csb1 sara 3000000
csb7 zara 4000000
csb9 qara 5000000
csb3 lara 7000000
csb6 vara 7000000
0
2
1
```

```
6
MT19112 Shivansh 3000000
MT19100 Akanksha 2000000
MT1912 Divya 3000000
MT1923 Shailja 2000000
MT1911 Anupam 3000000
MT1915 Ritika 2500000
3
2000000
3000000
1550000
MT19100 Akanksha 2000000
MT1923 Shailja 2000000
MT1915 Ritika 2500000
MT19112 Shivansh 3000000
MT1912 Divya 3000000
MT1911 Anupam 3000000
2
3
0
```

2. **Deadline Sort:**
   Time Complexity: O(k), since number of digits will always be 2 if we use base k represent. In base k, 2 digits can represent up to $k^2 - 1$.
   Sorting Algorithm: Radix Sort
   Space Complexity: O(k) since in each pass, we need to store k elements.

```java
package deadlinesort;
import java.lang.Math;
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Scanner;

public class deadlineSort {
    //This function implements the radix sort algorithm
    public static void dsort(int[] array, int k){

        //auxiliary space complexity = O(k)
        ArrayList<ArrayDeque<Integer>> buckets = new ArrayList<>(k);
        for(int i = 0; i<k;i++){
            buckets.add(new ArrayDeque<Integer>());
        }
        //total number of passes = 2
        for(int i = 0; i<2; i++){
```

```java
            //inner loop complexity = O(k)
            int extractor = (int)Math.pow(k,i);
            for(int j = 0; j < k; j++){
                //extract digit at
                int x = array[j]/extractor;
                //to the bucket no x%k
                buckets.get(x%k).addLast(array[j]);
            }

            int a = 0; //variable to loop through array

            //popping buckets and adding to array
            //O(k)
            for(int q = 0; q<k; q++){
                //empty bucket q
                while(!buckets.get(q).isEmpty())
                    array[a++]=buckets.get(q).removeFirst();
            }
        }
        //end of outer for loop
        //complexity is O(k),
        //because base k representation gives only two digits for
        //0 to k^2-1
        //therefore there are two passes through k numbers


    }

    public static void main(String[] args){
        Scanner s = new Scanner(System.in);
        int k = s.nextInt();
        int maxdigit = 0;
        int[] deadlines = new int[k];

        for(int i =0; i<k;i++){
            deadlines[i]=s.nextInt();
        }

        dsort(deadlines,k);
        for(int i =0; i<k;i++){
            System.out.printf("%d ",deadlines[i]);
        }
    }
}
```

Output:

```
10
6 1 7 2 8 56 75 14 24 95
1 2 6 7 8 14 24 56 75 95
```

```
5
13 24 1 17 8
1 8 13 17 24
```

3. **Helping Hands:**

   Time Complexity: O(nlogn) for sorting and O(n) for finding the minimum number of helpers required. Overall O(nlogn).

   Algorithm: At first finding the lower bound and upper bound for each helper whose value is not -1. Sorting the helpers according to their lower bounds in ascending order. Sorting the helpers with similar lower bounds in descending order. In this way we receive the maximum span that starts the earliest.
   Next we greedily find the successive helper's span such that lower bound for it does not exceed more than one person of our current helpers span, and has the highest upper bound.
   this is repeated until either we run out of helpers or our upper bound matches the maximum number of people requiring helper's help to be fed.
   If after checking with all helpers our maximum upper bound does not match the number of people to be fed… we cannot feed all the people with our helpers, and return -1.
   Also, if the helper with the lowest lower bound has a lower bound higher than 0, that means we cannot feed all the people and must return -1.
   In all other cases, we return the number of helpers calculated in the function.

   Space complexity: O(n) to store the lower bounds and upper bounds of all the helpers with value not equal to -1.

```
package helpinghands;
import java.util.*;
class p{
    int x;
    int y;}


class xsort implements Comparator<p>{
    public int compare(p a, p b){
        if(a.x==b.x) return  0;
        else if(a.x>b.x) return 1;
        else return -1;
    }
}

class ysort implements Comparator<p>{
    public int compare(p a, p b){
        if(a.y==b.y) return  0;
        else if(a.y<b.y) return 1;
        else return -1;
    }
}

//0-4,5-5


public class helpingHands{
    public static int helpinHands(ArrayList<p> pairs, int k){
```

```
        int minimumHelpers = 1;
        int cub,ii,mub;

        if (pairs.get(0).x != 0) return -1;
        else if(pairs.get(0).y==k) return minimumHelpers;
        else {
            cub = pairs.get(0).y;
        }
//          cub is the current upper bound
        for(int i = 0; i< pairs.size();){
            ii = i+1;
            mub=cub;
            boolean whileexecuted=false;
//            System.out.printf("%n***current cub %d ii=%d%n",cub,ii);
            // greedily selecting the next best helper
            //who has the highest range
            //but also does not leave any uncovered people in the middle.
            //if our present helper is providing till kth person,
            //the next helper must start on or before k+1'th person.
            //this step ensures that we cover maximum number of people
            //using least number of helpers.
            while( ii < pairs.size() && pairs.get(ii).x<=(cub+1)){
                whileexecuted=true;
                if(pairs.get(ii).y>mub)mub = pairs.get(ii).y;
                ii++;
            }
            //once we find the next best helper, we increment the number of
helpers
            ++minimumHelpers;
             if (mub == k) return minimumHelpers;
             else{
                 cub = mub;
                 if(whileexecuted)i = --ii;
                 else i++;
             }
//               System.out.printf("%ncurrent cub %d%n",cub);
        }
        //if at the end of the procedure the upper bound of our coverage by
helpers
        //matches the maximum people (0 to n-1) then we can say that
        //it is possible to feed all the people with our helper's help
        //if not, then it is not possible
        if(cub == k) return minimumHelpers;
        else return -1;

    }

    public  static void main(String[] args){
        ArrayList<p> pairs = new ArrayList<>();
        Scanner s = new Scanner(System.in);
        int t = s.nextInt();
        for(int i =0; i<t;i++){
            int x = s.nextInt();
            if(x!=-1){
                p helper = new p();
//                while entering, we are clipping the
//                range of our helpers to minimum of 0
//                and maximum of n-1
                helper.x = Math.max(0,i-x);
                helper.y = Math.min(x+i,t-1);
```

```
                pairs.add(helper);
            }
        }
        s.close();
//          for(int i=0; i<pairs.size();i++)
//              System.out.printf("%n%d %d
%n",pairs.get(i).x,pairs.get(i).y);
        Collections.sort(pairs,new xsort().thenComparing(new ysort()));
//          for(int i=0; i<pairs.size();i++)
//              System.out.printf("%n%d %d
%n",pairs.get(i).x,pairs.get(i).y);
        System.out.println(helpinHands(pairs,t-1));
    }


}
```

Output:

```
6
-1 2 2 -1 0 0
2
```

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

4. **Free Ryloth:**

Time Complexity: O(n). The algorithm run as many times as there are nodes(house) in the queue, and each house/node is queued exactly once.

Algorithm: The tree is stored as an adjacency list of nodes followed by a list of nodes it has links to. At first each node is given a level no -1. Then the source received from the input is given a level no of 0, and each of it's linked node is given a level no of one higher, that is 1. This means that, all the nodes(houses) immediately connected to the source node(house) can receive the information in one hour. Then the nodes are queued to access one by one, and exploring their immediate connections whose level no is still -1, and changing it to the level of one more than the current node(house). All the houses that received the information in the first hour can send the information to it's uninformed direct neighbours in the second hour. This process is repeated until all the houses are covered, while keeping track of the highest level reached(our level is our hour).

Then the highest level is returned.

This is similar to doing a breadth first search, using the provided node as our starting point, on a graph since tree is an acyclic graph.

Space Complexity: n*3 = O(n), because each node has maximum of three links. Parent, left child, right child.

```
package freeryloth;

import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Deque;
import java.util.Scanner;

public class freeRyloth {
    public static void makeTree(String input, String nodes,
ArrayList<ArrayList<Character>> tree ){

        for (int i = 0; i < input.length(); i++){
            int idx;
            char x = input.charAt(i);
            char rchild,lchild;
            //if ith node in the input array is not 'N'
            if(Character.isDigit(x)){
                //get its numeric value
                int xi = Character.getNumericValue(x);
               idx = nodes.indexOf(x);
               //if no child, break out of the loop
               if(idx*2+1>=input.length()) break;
               lchild = input.charAt(idx*2+1);
               rchild = input.charAt(idx*2+2);
               //if its left child is also a int
                //add this link to the lists of both
                //the current node and its child
               if(Character.isDigit((lchild))){
                   //list for the current node
                   tree.get(xi).add(lchild);
                   //list for the left child node
                   tree.get(Character.getNumericValue((lchild))).add(x);
               }
                //if its right child is also an int
                // add this link to the lists of both
                //the current node and its child
               if(Character.isDigit((rchild))){
                   //list for the current node
                   tree.get(xi).add(rchild);
                   //list for the left child node
                   tree.get(Character.getNumericValue((rchild))).add(x);

               }

            }
        }
    }
    public static int spreadInfo(ArrayList<ArrayList<Character>> tree, char
houseno, String nodes, int m){
        int[] levels = new int[m+1];
        for(int i = 0; i<m+1; i++){
            levels[i]=-1;
        }
        int k = Character.getNumericValue(houseno);
        int hours = 0;
        levels[k]=0;
        Deque<Character> X = new ArrayDeque<>();
        X.addLast(houseno);
```

```java
        //outer loop runs for all the unvisited nodes, therefore max n times
        while(true){
            //since it's a binary tree, this inner loop will run maximum 3 times.
            for(int i = 0; i <tree.get(k).size();i++){
                int x =  Character.getNumericValue(tree.get(k).get(i));
                if(levels[x]==-1) { //System.out.printf("k here is %d %n",k);
                    levels[x] = levels[k]+1;
                X.addLast(tree.get(k).get(i));
                hours = Math.max(hours,levels[x]);}
            }
            if(!X.isEmpty()) {
                k = Character.getNumericValue(X.removeFirst());
            }
            else break;
        }
        //runs at maximum 3*n. Therefore it is O(n).

    return hours;
    }

    public static void main(String[] args){
        Scanner s = new Scanner(System.in);
        String input = s.nextLine();
        char firstHouse = s.next().charAt(0);
        s.close();
        //preprocess theinpt to construct the tree
        input = input.replaceAll(" ","");
        String nodes = input.replaceAll("N","");
        int maxnode = 0;
        for(int i = 0; i<nodes.length();i++)
            maxnode = Math.max(maxnode,Character.getNumericValue(nodes.charAt(i)));
        //initialising the tree
        ArrayList<ArrayList<Character>> tree = new ArrayList<>(maxnode+1);
        for(int i = 0; i< maxnode+1; i++) tree.add(new ArrayList<>());
        //constructing the tree
        makeTree(input,nodes,tree);
        //traversing
        System.out.println(spreadInfo(tree,firstHouse,nodes,maxnode));


    }
}
```

Output:

```
1 2 3 N 4 5 6 N N N 7
5
4
```

```
1 2 3 N N 4 6 N 5 N N 7 N
3
3
```