Task 1

Write a function solution that, given an array A of N integers, returns the largest integer K > 0 such that both values K and -K (the opposite number) exist in array A. If there is no such integer, the function should return 0.
Examples:
1. Given A = [3, 2, -2, 5, -3], the function should return 3 (both 3 and -3 exist in array A).
2. Given A = [1, 1, 2, -1, 2, -1], the function should return 1 (both 1 and -1 exist in array A).
3. Given A = [1, 2, 3, -4], the function should return 0 (there is no such K for which both values K and -K exist in array A).
Write an efficient algorithm for the following assumptions:
- N is an integer within the range [1.. 100,000];
- each element of array A is an integer within the range
- [-1,000,000,000.. 1,000,000,000].

For the first task I developed two solutions. To reduce the complexity from O(n*2) to O(n)

Final Solution Task-1:

```
    public int solution(int[] A) {
        Set<Integer> set = new HashSet<Integer>();
        int maxK = 0;
        for(int i=0;i<A.length;i++){
            int val = A[i];
            if(set.contains(-val)){
                maxK = Math.max(maxK, Math.abs(val));
            }
            set.add(val);
        }
        return maxK;
    }
}
```

Initial Solution Task-1:

```
public class Q1n2 {
    public int solution(int[] A) {
        Arrays.sort(A);
        for (int i=0;i<A.length;i++){
            if (A[i] < 0 ){
                int x = Math.abs(A[i]);
                if (Arrays.stream(A).anyMatch(y -> y == x)){
                    return x;
                }
            }
        }
        return 0;
    }
}
```

Task 2

A group of friends is going on holiday together. They have come to a meeting point (the start of the journey) using N cars. There are P[k] people and S[k] seats in the K-th car for K in range [0..N-1]. Some of the seats in the cars may be free, so it is possible for some of the friends to change the car they are in. The friends have decided that, in order to be ecological, they will leave some cars parked at the meeting point and travel with as few cars as possible.
Write a function:
class Solution { public int solution(int [] P, int [l S) ; }
that, given two arrays P and S, consisting of N integers each, returns the minimum number of cars needed to take all of the friends on holiday.
Examples:
1. Given P = [1, 4, 1] and S = [1, 5, 1], the function should return 2. A person from car number 0 can travel in car
number 1 instead. This way, car number 0 can be left parked at the meeting point.
2. Given P = [4, 4, 2, 4] and S = [5, 5, 2, 5], the function should return 3. One person from car number 2 can travel
in car number 0 and the other person from car number 2 can travel in car number 3.
3. Given P = [2, 3, 4, 2] and S = [2, 5, 7, 2], the function should return 2. Passengers from car number 0 can travel
in car number 1 and passengers from car number 3 can travel in car number 2.
Write an efficient algorithm for the following assumptions:
N is an integer within the range [1.. 100,000];
each element of arrays P and S is an integer within the range 11.9;
every friend had a seat in the car they came in; that is, PikIs Sikfor each K within the range [O.N-1].

Solution Task 2:

```
public int solution(int[] P, int[] S) {

// calculate total passengers
    int totalPassengers = Arrays.stream(P).sum();
    int seats = 0;
    Arrays.sort(S);
    int carUsed = 0;

// loop through the sorted array of car seats until total passengers fits
in and then return used cars.

    for (int i=S.length-1;i>=0;i--){
        seats+=S[i];
        carUsed++;
        if(seats>=totalPassengers){
            return carUsed;
        }
    }
    return 0;
}
```

Task 3

You are given an undirected graph consisting of N vertices, numbered from 1 to N, and M edges.

The graph is described by two arrays, A and B, both of length M. A pair (A[K], B[K]), for K from 0 to M-1, describes an edge between vertex A[k] and vertex B[K].

Your task is to check whether the given graph contains a path from vertex 1 to vertex N going through all of the vertices, one by one, in increasing order of their numbers. All connections on the path should be direct.

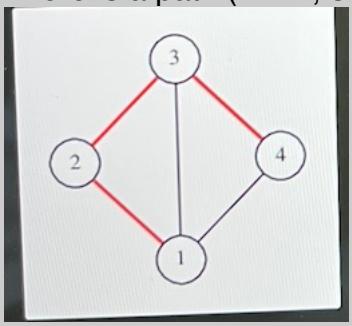
Write a function:

class Solution { public boolean solution(int N, int[I A, int[I B); }

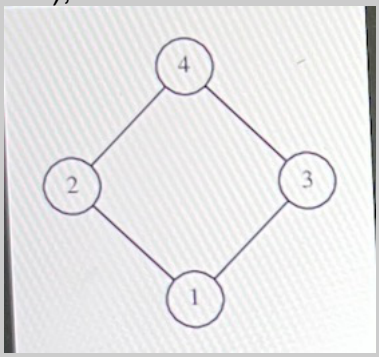
that, given an integer N and two arrays A and B of M integers each, returns true if there exists a path from vertex 1 to N going through all vertices, one by one, in increasing order, or false otherwise.

Examples:

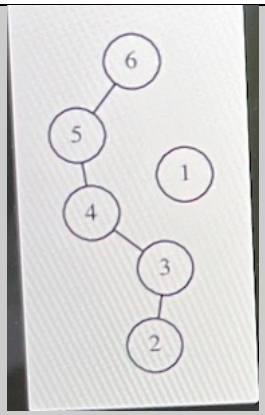
1. Given N = 4, A = [1, 2, 4, 4, 3] and B = [2, 3, 1, 3, 1], the function should return true. There is a path (1 - 2-, 3 - 4) using edges (1, 2), (2, 3) and (4, 3).



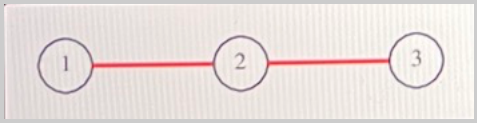
2. Given N = 4, A = [1, 2, 1, 3] and B = [2, 4, 3, 4], the function should return false. There is no path (1 - 2 - 3
-› 4), as there is no direct connection from vertex 2 to vertex 3.



3. Given N = 6, A = (2, 4, 5, 3] and B = 13, 5, 6, 41, the function should return fats@ There is no direct connection
from vertex 1 to vertex 2.



4. Given N = 3, A = [1, 3] and B = [2, 2], the function should return true. There is a path (1 - 2 - 3) using
edges (1, 2) and (3, 2).



Write an efficient algorithm for the following assumptions:
N is an integer within the range [2.. 100,000];
M Is an integer within the range [0.. 100,000];
all elements of arrays A and B are integers within the range [1..N];
there are no self-loops (edges with AlkJ= BiKI in the graph;
• there are no multiple edges between the same vertices.

Solution Task 3:

```
public boolean solution(int N, int[] A, int[] B) {
    // graph is a hashmap which maps each node to a set of its neighbours
    Map<Integer, Set<Integer>> graph = new HashMap<>();

    // initializing the graph with empty sets
    // initially all nodes point towards an empty set
    for (int i = 0; i < N; i++) {
        graph.put(i, new HashSet<>());
    }

    // for each edge A[i],B[i]
    for (int i = 0; i < A.length; i++) {
        // -1 operation is only done to translate 1 index to 0 index
        graph.get(A[i] - 1).add(B[i] - 1);  // adding B[i] as neighbour of
A[i]
        graph.get(B[i] - 1).add(A[i] - 1);  // adding A[i] as neighbour of
B[i]
    }


    for (int i = 0; i < N - 1; i++) {
        // if any consecutive nodes i and i+1 are not each other's
neighbours return false
        if (!graph.get(i).contains(i + 1)) {
            return false;
        }
    }
    return true; // path found
}
```