There are *N* breakfasts in the restaurant “Fat Hut” where the Arun works. The *ith* breakfast has an attractiveness *Ai*? and cost *Ci*?.

Arun has noticed that nobody takes the *jth* breakfast if there exists at least one breakfast *i* such that *Ai*?≥*Aj*? and *Ci*?<*Cj*?.

In other words, if a breakfast is less attractive and more expensive than any of the other dishes, then nobody is interested in that breakfast.

Arun will be happy if all the *N* breakfasts have a chance to be taken. Unfortunately, Arun has no power over prices. On the other hand, he can change the attractiveness of some breakfasts by some real number. However, after the changes, the attractiveness of the *ith* breakfast must lie in the interval [*Li*?,*Ri*?].

He would also like to change the attractiveness of the minimum number of breakfasts. Help the Chef do it.

Input Format

- The first line of the input contains a single integer *T* denoting the number of test cases. The description of *T* test cases follows.
- Each testcase contains *N*+1 lines of input.
- The first line of each test case contains a single integer *N* - number of breakfasts.
- *N* lines follow. For each valid *i*, the h*ith* of these lines contains four space-separated integers *Ai*?, *Ci*?, *Li*?, *Ri*? - current attractiveness, cost, the minimal and maximal allowed attractiveness after change of *i*-th breakfast.

Output Format

For each test case, print in a single line containing one integer - minimum number of breakfasts the Chef should change so that all the *N* breakfasts have a chance to be taken. Print "-1"(without quotes) if it is impossible to achieve the goal.

Sample Input

2

4

5 1 1 5

4 4 2 5

2 2 2 5

3 3 2 5

4

5 1 2 5

4 4 2 5

2 2 2 5

3 3 2 5

Sample Output

1

2

```
import java.util.*;

class Main {

	public static void main(String[] args) {
		Scanner sc = new Scanner(System.in);
		try {
			int t = 0;
			if (sc.hasNextInt())
				t = sc.nextInt();
			while (t-- > 0) {
				int n = sc.nextInt();
				D[] dishes = new D[n];
				for (int i = 0; i < n; i++) {
					long a = sc.nextLong();
					long c = sc.nextLong();
					long l = sc.nextLong();
					long r = sc.nextLong();
					dishes[i] = new D(a, c, l, r);
				}
				int res = solution(dishes, n);
				System.out.println(res);
			}
		} catch (Exception e) {
			e.printStackTrace();
			return;
		} finally {
			sc.close();
		}
```

```
	}

	private static int solution(D[] dishes, int n) {
		Arrays.sort(dishes, 0, n, (a, b) -> Long.compare(a.cost, b.cost));
		TreeSet<Long> v = new TreeSet<Long>();
		long max = 0;
		boolean fail = false;
		for (int i = 0; i < n; ++i) {
			long l = dishes[i].l;
			long r = dishes[i].r;
			long a = dishes[i].attr;
			if (r <= max) {
				fail = true;
				break;
			}
			while (!v.isEmpty() && v.last() >= r)
				v.remove(v.last());
			if (a > max) {
				if (v.isEmpty() || a > v.last())
					v.add(a);
				else {
					long pos = v.ceiling(a);
					v.remove(pos);
					v.add(a);
				}
			}
			max = Math.max(max, l);
		}
		if (!fail)
			return n - v.size();
		return -1;
	}
}

class D {
	long attr;
	long cost;
	long l;
	long r;

	public D(long attr, long cost, long lattr, long rattr) {
		this.attr = attr;
		this.cost = cost;
		this.l = lattr;
		this.r = rattr;
	}
}
```