Rešitev enajste domače naloge (Fiat lux!)

Tudi tokrat je možnih več pristopov. Najprej si bomo ogledali naivno rešitev, ki deluje le za dovolj majhne vrednosti M, nato pa bomo prikazali programa, ki bosta pokrila vse testne primere in ... ne bosta prav nič zapletenejša.

Rešitev za 40% ($M = 1000$)

Izdelali bomo tabelo `osvetlitev` tipa `boolean[][]`, v kateri vsak element z vrednostjo `true` pove, da je pripadajoča celica koordinatnega sistema osvetljena. Na začetku bodo vsi elementi tabele imeli vrednost `false`, nato pa za vsako luč (recimo, da ima koordinati (x, y)) in za vsako od devetih celic, ki jih luč osvetljuje (torej za celice $(x-1, y-1)$, $(x, y-1)$, $(x+1, y-1)$, $(x-1, y)$, (x, y), $(x+1, y)$, $(x-1, y+1)$, $(x, y+1)$ in $(x+1, y+1)$), nastavimo pripadajoči element tabele na `true`. Na koncu se sprehodimo po tabeli `osvetlitev` in preštejemo elemente z vrednostjo `true`.

Koordinat celic ne moremo neposredno uporabiti kot indekse v tabelo `osvetlitev`, saj so lahko tudi negativne. Vemo pa, da se bodo vse koordinate nahajale v intervalu $[-M-1, M+1]$, saj luč v skrajnem primeru stoji na koordinati $x = \pm M$ ali $y = \pm M$. Koordinate zato zlahka pretvorimo v nenegativne indekse, saj moramo zgolj prišteti $M+1$. Interval možnih indeksov tako postane $[0, 2M+2]$, kar pomeni, da mora biti tabela `osvetlitev` velika vsaj $(2M+3) \times (2M+3)$.

```
import java.util.Scanner;

public class FiatLux {

    private static final int M = 1000;

    private static final int[][] ODMIKI = {
        {-1, -1}, {0, -1}, {1, -1},
        {-1,  0}, {0,  0}, {1,  0},
        {-1,  1}, {0,  1}, {1,  1}
    };

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int stLuci = sc.nextInt();
        boolean[][] osvetlitev = new boolean[2 * M + 3][2 * M + 3];

        // sprehod po lučeh
        for (int i = 0;  i < stLuci;  i++) {
            int x0 = sc.nextInt();
            int y0 = sc.nextInt();

            // sprehod po celicah, ki jih osvetljuje luč
            for (int[] odmik: ODMIKI) {
                int x = x0 + odmik[0];
                int y = y0 + odmik[1];
                osvetlitev[y + M + 1][x + M + 1] = true;
```

```
            }
        }

        // preštejemo osvetljene celice
        int stOsvetljenih = 0;
        for (int i = 0;  i < osvetlitev.length;  i++) {
            for (int j = 0;  j < osvetlitev[i].length;  j++) {
                if (osvetlitev[i][j]) {
                    stOsvetljenih++;
                }
            }
        }
        System.out.println(stOsvetljenih);
    }
}
```

Rešitev za 100% (z uporabo seznama)

Če je število M preveliko, da bi v pomnilnik stlačili dvodimenzionalno tabelo s stranico dolžine $2M+3$, imamo na voljo sledečo možnost: za vsako luč dodamo vseh devet osvetljenih celic v *seznam* in določimo število različnih celic v njem (kot vemo, se lahko zgodi, da isto celico osvetljuje več luči). To najlažje in najučinkoviteje storimo tako, da seznam uredimo, nato pa se po njem sprehodimo in preštejemo, kolikokrat je trenutna celica različna od predhodne.

Seznam bomo predstavili kot objekt tipa `List<Celica>`, objekti razreda `Celica` pa bodo seveda predstavljali posamezne celice. Vsaka celica je določena s svojima koordinatama:

```
public class Celica {
    private int x;
    private int y;

    public Celica(int x, int y) {
        this.x = x;
        this.y = y;
    }
    ...
}
```

Ker bomo seznam celic uredili, je smiselno, da razred `Celica` implementira vmesnik `Comparable<Celica>`. Druga možnost bi bila, da bi izdelali objekt tipa `Comparator<Celica>` in ga podali metodi za urejanje seznama.

```
public class Celica implements Comparable<Celica> {
    ...
    @Override
    public int compareTo(Celica druga) {
        int dx = this.x - druga.x;
        return (dx == 0) ? (this.y - druga.y) : (dx);
    }
}
```

Metoda `compareTo` primerja podani celici po koordinatah x, celici z enakima koordinatama x pa po koordinatah y. Za potrebe te naloge moramo pri implementaciji metode `compareTo` zagotoviti le to, da se bodo celice z enakimi koordinatami v urejenem seznamu nahajale na zaporednih mestih. Zaradi tega je možnih več pristopov, seveda pa lahko trdimo, da so nekateri »bolj naravni« od drugih.

Glavni razred lahko sedaj napišemo takole:

```
import java.util.*;

public class FiatLux {

    private static final int[][] ODMIKI = {
        {-1, -1}, {0, -1}, {1, -1},
        {-1,  0}, {0,  0}, {1,  0},
        {-1,  1}, {0,  1}, {1,  1}
    };

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int stLuci = sc.nextInt();
        List<Celica> osvetljene = new ArrayList<>();

        for (int i = 0;  i < stLuci;  i++) {
            int x0 = sc.nextInt();
            int y0 = sc.nextInt();
            for (int[] odmik: ODMIKI) {
                osvetljene.add(new Celica(x0 + odmik[0], y0 + odmik[1]));
            }
        }

        // Seznam naravno uredimo, zato kot parameter podamo null.
        osvetljene.sort(null);

        // Preštejemo, kolikokrat je trenutna celica različna od predhodne.
        // Ker je seznam urejen, bo vsaka celica po naravni urejenosti
        // sodila za svojo predhodnico, če bo različna od nje.
        int stOsvetljenih = 0;
        Celica predhodna = null;
        for (Celica celica: osvetljene) {
            if (predhodna == null || celica.compareTo(predhodna) > 0) {
                stOsvetljenih++;
            }
            predhodna = celica;
        }
        System.out.println(stOsvetljenih);
    }
}
```

Nujno je, da seznam uredimo z dovolj učinkovitim postopkom. Metoda `sort`, ki jo implementira razred `ArrayList`, je dovolj dobra, metoda za urejanje z algoritmom navadnega vstavljanja, ki smo jo napisali na predavanjih, pa žal ni ...

Rešitev za 100% (z uporabo množice)

Najbolj naravna je rešitev z uporabo množice. Ker množica sama od sebe zagotavlja, da noben element v njej ne bo imel svojega dvojnika, lahko nalogo rešimo enostavno tako, da si pripravimo množico tipa `Set<Celica>` in vanjo za vsako luč dodamo vseh devet osvetljenih celic, množica pa bo sama poskrbela, da v njej ne bo duplikatov. Na koncu le še pridobimo število elementov množice in imamo želeni rezultat.

Glede na to, da razred `Celica` implementira vmesnik `Comparable`, se nam množico splača ustvariti kot objekt tipa `TreeSet`. Če bi želeli uporabiti množico tipa `HashSet`, bi morali v razredu `Celica` redefinirati metodi `equals` in `hashCode`, implementacijo vmesnika `Comparable` pa bi lahko odstranili.

```
import java.util.*;

public class FiatLux {

    private static final int[][] ODMIKI = {
        {-1, -1}, {0, -1}, {1, -1},
        {-1,  0}, {0,  0}, {1,  0},
        {-1,  1}, {0,  1}, {1,  1}
    };

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int stLuci = sc.nextInt();
        Set<Celica> osvetljene = new TreeSet<>();

        for (int i = 0;  i < stLuci;  i++) {
            int x0 = sc.nextInt();
            int y0 = sc.nextInt();
            for (int[] odmik: ODMIKI) {
                osvetljene.add(new Celica(x0 + odmik[0], y0 + odmik[1]));
            }
        }
        System.out.println(osvetljene.size());
    }
}
```

Ta rešitev je glede porabe časa povsem primerljiva s prejšnjo, je pa od nje nesporno krajša in elegantnejša.