A-I-1 Skriňa na chemikálie

Našou úlohou bolo pridávať a odstraňovať chemikálie zo skrine, pričom chemikáliu s číslom x sme mohli do skrine pridať iba vtedy, ak žiadna iná chemikália v skrini nebola s číslom x súdeliteľná.
Najprv sa zamyslime, čo môžeme použiť na reprezentovanie samotnej skrine. Potrebujeme štruktúru, do ktorej vieme hodnoty vkladať, vyberať ich odtiaľ a tiež zisťovať, či je zadaná hodnota v tejto štruktúre už uložená. Klasické polia (listy) vedia tieto operácie robiť, sú však neefektívne. Lepším riešením je preto použiť `set`, teda efektívnu implementáciu množiny, ktorú nájdeme vo väčšine štandardných programovacích jazykov. Tá vie v konštantnom (alebo logaritmickom) čase robiť všetky vyššie uvedené operácie.
Ako však zistiť, či je nová hodnota x súdeliteľná s nejakou hodnotou, ktorá sa už v skrini nachádza? Najjednoduchšie je postupne zobrať každú hodnotu y, ktorá sa v skrini nachádza a zistiť, či sú x a y súdeliteľné. Napríklad tak, že vyskúšame všetky možné delitele týchto čísel. Zložitosť takéhoto riešenia je však $O(xq^2)$, čo je príliš pomalé.
Pozrime sa však na definíciu súdeliteľnosti. Čísla x a y sú súdeliteľné, ak majú aspoň jedného spoločného deliteľa väčšieho ako 1. To ale platí práve vtedy, ak ich *najväčší spoločný deliteľ* je väčší ako 1. A na hľadanie najväčšieho spoločného deliteľa existuje štandardná knižničná funkcia `gcd()`, ktorej zložitosť je logarimická. Využitím tejto funkcie si kód nielen skrátime, ale aj zefektívnime na $O(q^2 \log x)$.

Listing programu (Python)

```
import math

q = int(input())

skrina = set()
for _ in range(q):
    t, x = map(int, input().split())
    # odstranujem zo skrine
    if t == 2:
        if x in skrina:
            skrina.remove(x)
    # pridavam do skrine
    else:
        vloz = True
        for y in skrina:
            if math.gcd(x, y) != 1:
                vloz = False
                break
        if vloz:
            print("ano")
            skrina.add(x)
        else:
            print("nie")
```

Rýchlejšia kontrola súdeliteľnosti

Aktuálne najmenej efektívna časť nášho riešenia spočíva v tom, že pre každé pridávané číslo x musíme skontrolovať všetky už vložené hodnoty y. V úlohe však nepotrebujeme vedieť, s ktorým y je naše x súdeliteľné. Iba či s nejakým číslom súdeliteľné je.
Namiesto toho, aby sme teda skúšali všetky čísla, mohli by sme sa pozrieť na všetky delitele čísla x a zistiť, či tento deliteľ delí aj niektoré číslo y.

Okrem množiny čísel, ktoré sú v skrini si preto spravíme aj množinu všetkých *deliteľov* čísel, ktoré sú v skrini. Následne, keď do skrine chceme pridať hodnotu x, prejdeme cez všetkých deliteľov čísla x a skontrolujeme, či sa tento deliteľ nenachádza aj v našej množine. Všimnime si, že v takomto prípade priamo nevidíme, ktorému y tento deliteľ patrí, vieme však, že nejaké súdeliteľné y existuje.
Množinu deliteľov musíme upravovať pri každom pridaní a odobraní čísla zo skrine. Pri pridaní čísla x do tejto množiny vložíme všetky delitele čísla x. Keď číslo x zo skrine vyberáme, tiež bude postačovať, že z množiny všetky jeho delitele odstránime. Uvedomme si totiž, že v skrine nemôžu byť naraz dve súdeliteľné čísla. To znamená, že žiaden z deliteľov čísel zo skrine nemôže patriť dvom číslam, znamenalo by to totiž, že sú súdeliteľné.
Číslo x má nanajvýš x rôznych deliteľov, ktorých musíme vyskúšať, či už pri pridávaní alebo odoberaní. Časová zložitosť takéhoto riešenia je preto $O(qx)$, čo je vzhľadom na limity v zadaní o kúsok horšie ako predchádzajúce riešenie.

Časovú zložitosť tohto riešenia vieme zlepšiť na $O(\sqrt{x})$ ak si uvedomíme, že číslo x v skutočnosti nemôže mať viac ako $2\sqrt{x}$ deliteľov a tie vieme nájsť v čase $O(\sqrt{x})$. Totiž ak nejaké $d > \sqrt{x}$ delí x, tak aj $e = x/d$ je deliteľom x a zjavne $e < \sqrt{x}$. Všetky delitele x teda vieme nájsť tak, že nájdeme všetky delitele po $\sqrt{x}$ a potom ku každému takémuto deliteľu e ešte pridáme aj hodnotu x/e medzi delitele.
Nižšie si ukážeme ešte lepší pohľad na náš problém. Ten povedie najskôr k rovnako dobrému a potom ku ešte lepšiemu riešeniu.

Prvočísla

Náš hrubý odhad bol, že ak chceme nájsť všetkých deliteľov čísla x, musíme vyskúšať všetky čísla od 2 po x. Čísla totiž môžu mať pomerne veľa deliteľov, napríklad $x = 24$ má delitele $2, 3, 4, 6, 8, 12$ a 24. Väčšina z týchto deliteľov však vznikla ako kombinácia iných, menších deliteľov. A tieto kombinované delitele nám nedajú žiadnu dodatočnú informáciu.
Kľúčové je uvedomiť si, že keď chceme vedieť, ktoré iné čísla sú súdeliteľné s nejakým x, stačí sa pozerať na jeho *prvočíselné* delitele. Ak totiž čísla x a y majú spoločného deliteľa d, každý prvočíselný deliteľ čísla d tiež delí zároveň x aj y.
Pozrime sa napríklad na číslo $72 = 2 \times 2 \times 2 \times 3 \times 3$. Každé iné párne číslo s ním bude súdeliteľné, lebo jedným z ich spoločných deliteľov je číslo 2. Podobne, každý násobok troch je súdeliteľný s číslom 72. A naopak – čísla, ktoré nie sú ani párne ani násobky troch, sú s číslom 72 zjavne nesúdeliteľné.
Pri vkladaní či vyberaní čísla x nám teda stačí, keď sa budeme pozerať na prvočísla, ktoré x delia. Tie si budeme ukladať do množiny deliteľov a zase ich odtiaľ odstraňovať.
Ostáva už len efektívne určiť prvočíselný rozklad čísla x. Jednou možnosťou je postupne skúšať delitele čísla x (začnúc od 2) a vždy, keď nejakého deliteľa nájdeme, číslo x ním vydelíme. Keďže z čísla x postupne odstraňujeme vždy najmenšieho zostávajúceho deliteľa, každý takto odstránený deliteľ musí byť prvočíslo.
Navyše vieme celý proces urýchliť, keď si uvedomíme, že každé číslo má najviac jedného prvočíselného deliteľa väčšieho ako $\sqrt{x}$. Nech $x = p_1 \cdot p_2 \cdot p_3 \ldots p_k$ je prvočíselný rozklad x. Ak by dve z prvočísel p_1 až p_k boli väčšie ako $\sqrt{x}$, celý súčin by bol väčší ako x, čo je spor. Pri delení teda stačí skúšať delitele dovtedy, kým nedosiahneme $\sqrt{x}$. Ak hodnota, ktorá vznikne z x delením menšími prvočíslami, nie je 1, tak táto hodnota je už nutne prvočíslo a toto prvočíslo je tým posledným v rozklade pôvodného x.
Takéto riešenie má časovú zložitosť $O(q\sqrt{x})$ a vieme zaň získať 5 bodov.

Listing programu (Python)

```
def rozklad_cisla(x):
    rozklad = []
    d = 2
    while d * d <= x:
        if x % d == 0:
            rozklad.append(d)
            while x % d == 0:
                x //= d
        d += 1
    if x != 1:
        rozklad.append(x)
    return rozklad

q = int(input())

skrina = set()
delitele = set()
for _ in range(q):
    t, x = map(int, input().split())
    rozklad = rozklad_cisla(x)
    # odstranujem zo skrine
    if t == 2:
        if x in skrina:
            skrina.remove(x)
            for d in rozklad:
                delitele.remove(d)
    # pridavam do skrine
    else:
        vloz = True
        for d in rozklad:
            if d in delitele:
                vloz = False
                break
        if vloz:
```

```
            print("ano")
            skrina.add(x)
            for d in rozklad:
                delitele.add(d)
        else:
            print("nie")
```

Efektívnejší rozklad

Na zisk plného počtu bodov potrebujeme ešte urýchliť rozklad čísel na prvočísla. Napríklad postačuje, ak pri zisťovaní rozkladu na prvočísla nebudeme skúšať všetky delitele do $\sqrt{x}$, ale iba prvočíselné delitele menšie ako $\sqrt{x}$, tých je totiž len 168. Týchto 168 prvočísel si môžeme buď spočítať priamo v programe, alebo si ich vypočítame na domácom počítači a do programu ich natvrdo napíšeme.
Tu si však ukážeme všeobecné riešenie s veľmi dobrou časovou zložitosťou.
Problém predchádzajúceho riešenia je, že sa na každé x pozerá samostatne a delitele začne hľadať nezávisle od ostatných hodnôt x. My však vieme, že naprieč programom dostaneme potenciálne až q rôznych čísel $x \leq 10^6$. Čo keby sme sa preto na to pripravili a ešte pred prvou požiadavkou by sme zistili prvočíselný rozklad *všetkých* možných hodnôt x. Tým, že budeme tieto rozklady zisťovať naraz pre všetky čísla, budeme vedieť náš postup zefektívniť.
Využijeme na to algoritmus Eratostenovho sita. Tento algoritmus funguje nasledovne: Zoberieme všetky čísla od 2 po n a postupne nimi prechádzame. Keď nájdeme *neškrtnuté číslo*, prehlásime o ňom, že je to prvočíslo. A následne *škrtneme* všetky násobky tohto čísla. Sú ním totiž deliteľné a nemôžu byť preto prvočísla. Práve preto je neškrtnuté číslo prvočíslo, žiadne menšie číslo ho neškrtlo, lebo nebolo jeho deliteľom. Eratostenovo sito je veľmi rýchle, na zistenie všetkých prvočísel medzi prvými n číslami potrebujeme len $O(n \log \log n)$ času.
Uvedomme si však, že Eratostenovo sito nám zároveň vie zistiť prvočíselné rozklady všetkých čísel. Totiž, ak škrtáme prvočíslom p, každému škrtanému číslu vieme oznámiť, že p je jeho prvočíselný deliteľ. Stačí teda, keď si pri škrtnutí toto prvočíslo zapamätáme. (V nižšie uvedenom programe toto robíme explicitne: Pre každé x si počas sita zostrojujeme zoznam prvočísel, ktoré ho delia. Vždy, keď objavíme nové prvočíslo p, pridáme ho do zoznamu všetkých x, ktoré sú jeho násobkami. Ak by sme chceli šetriť pamäťou, stačilo by si namiesto toho pre každé zložené x zapamätať to jedno p, pre ktoré sme ho pri Eratostenovom site škrtli.)
Pri použití tohto algoritmu potom pre každé x spracujeme len tie prvočísla, ktoré ho naozaj delia. Časová zložitosť tohto algoritmu sa odhaduje náročnejšie, keďže závisí od počtu prvočíselných deliteľov čísiel x, ktoré spracúvame. To je pri najhoršom $O(\frac{\log n}{\log \log n})$, v praxi je to však oveľa menej, keďže väčšina čísel má veľmi málo prvočíselných deliteľov, zhruba $O(\log \log n)$. Časová zložitosť tohto algoritmu je preto zhruba $O((x+q) \log \log x)$.

Listing programu (Python)

```
rozklady = [[] for _ in range(10**6 + 1)]

for x in range(2, len(rozklady)):
    if len(rozklady[x]) == 0:
        for i in range(x, len(rozklady), x):
            rozklady[i].append(x)

q = int(input())

skrina = set()
delitele = set()
for _ in range(q):
    t, x = map(int, input().split())
    # odstranujem zo skrine
    if t == 2:
        if x in skrina:
            skrina.remove(x)
            for d in rozklady[x]:
                delitele.remove(d)
    # pridavam do skrine
    else:
        vloz = True
        for d in rozklady[x]:
            if d in delitele:
                vloz = False
                break
        if vloz:
            print("ano")
            skrina.add(x)
            for d in rozklady[x]:
```

```
            delitele.add(d)
    else:
        print("nie")
```

A-I-2 Cap záhradníkom

Záhrada je celá pekná

Na rozohriatie vyriešme prvú sadu, v ktorej má každý chodník kladnú estetickosť. Všimnime si, že ak by zamestnanec firmy prešiel po ľubovoľnom chodníku, jeho estetickosť by zhoršil. Firmu sa nám teda neoplatí volať, respektíve ju môžeme poslať na cestu, ktorá neobsahuje žiadny chodník: vyberieme rovnakú začiatočnú aj koncovú lavičku.
Výsledná estetickosť záhrady tak bude rovnaká ako na začiatku, teda súčet všetkých e_i. Toto číslo si označme E a nazývajme ho *začiatočná estetickosť*.

Záhrada je aj škaredá

V zvyšných sadách sa nám však situácia komplikuje, keďže záhrada môže obsahovať cesty s negatívnou estetickosťou, ktoré sa nám oplatí opraviť.
Aby sa nám riešenie jednoduchšie opisovalo, budeme používať grafovú terminológiu: lavičky budeme nazývať *vrcholmi* a chodníky medzi nimi *hranami*. Dva vrcholy spojené hranou nazývame *susednými*.

Predtým, než začneme hľadať najlepšiu cestu na úpravu, zamyslime sa, aká bude výsledná estetickosť záhrady pre konkrétnu vybranú cestu.
Označme estetickosti hrán na vybranej ceste postupne $e_1, e_2, \ldots, e_\ell$. Po úprave sa ich estetickosti zmenia na $-e_1, -e_2, \ldots, -e_\ell$. Od začiatočnej estetickosti teda odpočítame pôvodné estetickosti týchto hrán a pripočítame ich nové estetickosti. Výsledná estatickosť záhrady bude

$$E - e_1 - e_2 - \ldots - e_\ell + ((-e_1) + (-e_2) + \cdots + (-e_\ell)) = E - 2(e_1 + e_2 + \cdots + e_\ell).$$

Inak povedané, výsledná estetickosť je začiatočná estetickosť záhrady mínus dvakrát súčet estetickostí na upravovanej ceste.
Aby sme dosiahli čo najvyššiu estetickosť, musíme nájsť cestu s čo najmenším súčtom estetickostí hrán. Pre jednoduchosť, súčet estetickostí hrán na ceste nazývame estetickosť cesty.

Riešenie hrubou silou

Využime to, že záhrada je grafový strom. Medzi každými dvoma vrcholmi existuje práve jedna cesta. Stačí nám teda nájsť dvojicu vrcholov takú, že cesta medzi nimi má najmenšiu možnú estetickosť. Takýchto dvojíc je iba $\frac{n(n-1)}{2}$.
Na hľadanie cesty, tak ako aj na počítanie jej estetickosti, použijeme *prehľadávanie do hĺbky*[1]. Predstavme si, že nás zaujíma cesta z a do b. Nájdeme ju rekurzívne: začneme v a a pozrieme sa do všetkých susedov a. Keďže existuje presne jedna cesta medzi a a b existuje presne jeden sused a, z ktorého sa do b vieme dostať bez toho, aby sme sa vrátili do a.
Na každého suseda a sa preto rekurzívne spýtame otázku: vieme nájsť cestu do b bez toho aby sme sa pozreli na vrchol, ktorý sme už videli? Ak áno, aká je estetickosť na ceste z aktuálneho vrchola do b?
Ak je aktuálny vrchol b, potom odpoveď na prvú otázku je zjavne áno, a na druhú 0. Ak aktuálny vrchol nie je b, spýtame sa túto otázku všetkých ešte nenavštívených susedov.
Ak sa z niektorého suseda, povedzme w vieme dostať do b, potom cesta z aktuálneho vrcholu do b existuje a jej estetickosť je estetickosť hrany z aktuálneho vrchola do w plus estetickosť cesty z w do b.

[1] Taktiež známe ako DFS.

Pre každý pár vrcholov zavoláme túto funkciu najviac raz pre každý vrchol, a teda časová zložitosť riešenia je $O(n^3)$. Implementovať to vieme napríklad nasledovne:

Listing programu (Python)

```
def najdi_cestu(v, ciel, prisli_sme_z, hrany):
    if v == ciel:
        return (True, 0)
    for w, e in hrany[v]:
        if w != prisli_sme_z:
            da_sa, esetickost = najdi_cestu(w, ciel, v, hrany)
            if da_sa:
                # nasli sme cestu
                return (da_sa, esetickost + e)
    return (False, 0)

n = int(input())
hrany = [[] for _ in range(n)]

E = 0

for _ in range(n - 1):
    a, b, e = map(int, input().split())
    a -= 1
    b -= 1
    hrany[a].append((b, e))
    hrany[b].append((a, e))
    E += e

min_estetickost = 0

for a in range(n):
    for b in range(a):
        min_estetickost = min(najdi_cestu(a, b, a, hrany)[1], min_estetickost)

print(E - 2 * min_estetickost)
```

Počítame toho priveľa, alebo lepšie riešenie

Toto riešenie nám stačí na dva body. Nie je ho však zložité zlepšiť tak, aby prešlo na zadaniach štvrtej sady. Všimnime si, že predchádzajúce prehľadávanie do hĺbky vlastne prechádza *všetky* cesty, ktoré začínajú vo vrchole a. Takže namiesto toho, aby sme hľadali súčet estetickostí hrán na ceste pre každý koniec b separátne, spýtame sa namiesto toho otázku: pre daný vrchol a, aký je najmenší súčet estetickosti hrán na ceste začínajúcej v a?
Aby sme si situáciu lepšie popísali, použijeme nasledujúcu grafovú terminológiu: ak sa prehľadávanie dostane do vrchola v potom *podstrom* v je tá časť stromu do ktorej sa dá dostať z v bez toho aby sme použili vrchol ktorým sme sa do v dostali (podstrom a je jednoducho celý strom). *Deti* vrcholu sú jeho susedia, ktorí sa nachádzajú v jeho podstrome.
Všimnime si, že ak sme sa v prehľadaní dostali do nejakého vrchola v a vrchol w je v podstrome v, potom estetickosť cesty z a do w je jednoducho súčet estetickostí ciest z a do v a z v do w (toto funguje, keďže existuje presne jedna takáto cesta, a táto cesta musí ísť cez v). Takže ak chceme nájsť minimálnu estetickosť cesty, ktorá začína v a a končí v podstrome v, stačí nám zistiť, aká je najmenšia estetickosť cesty začínajúcej vo v a končiacej v podstrome v, a pripočítať k nej estetickosť cesty z a do v.
Modifikujeme teda prehľadávanie do hĺbky, aby odpovedalo otázku: „Aká je minimálna estetickosť cesty ktorá začína v aktuálnehom vrchole a končí v jeho podstrome?“
Aby sme zistili odpoveď pre daný vrchol, položíme túto otázku každému dieťaťu nášho vrchola a pripočítame k jeho odpovedi estetickosť hrany, ktorá do suseda vedie. Následne vyberieme minimum, pričom ešte vezmeme do úvahy cestu s estetickosťou 0, ktorá začína aj končí v aktuálnom vrchole.
Pre každý vrchol spustíme celé prehľadávanie presne raz, a preto je výsledná časová zložitosť $O(n^2)$, čo stačí na vyriešenie štvrtej sady. Implementovať riešenie sa dá napríklad takto:

Listing programu (Python)

```
import sys
sys.setrecursionlimit(200000)

def najdi_najmensiu(v, prisli_sme_z, hrany):
    najmensia = 0
    for w, e in hrany[v]:
```

```
        if w != prisli_sme_z:
            najmensia = min(najmensia,
                            e + najdi_najmensiu(w, v, hrany))
    return najmensia

n = int(input())
hrany = [[] for _ in range(n)]

E = 0

for _ in range(n - 1):
    a, b, e = map(int, input().split())
    a -= 1
    b -= 1
    hrany[a].append((b, e))
    hrany[b].append((a, e))
    E += e

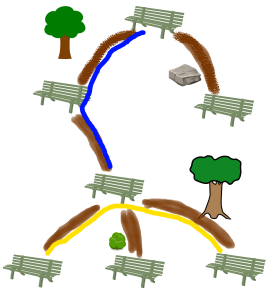
min_estetickost = min(najdi_najmensiu(i, i, hrany) for i in range(n))

print(E - 2 * min_estetickost)
```

Jedno prehľadávanie stačí, alebo vzorové riešenie

Toto riešenie je stále pomalé. Je to preto, že stále počítame niektoré veci viackrát. Napríklad, keď spustíme prehľadávanie z vrchola 1, tak vlastne zistíme aj informáciu o najmenšej estetickosti cesty, ktorá začína v nejakom susedovi 1 a končí v podstrome tohto suseda. Avšak, túto informáciu spočítame znova, keď spustíme prehľadávanie v tomto susedovi.
Na optimálne riešenie preto chceme aj tieto informácie nejako využiť. Predstavme si, že si prehľadávanie do hĺbky spustíme v nejakom fixnom vrchole. Cesty môžu byť dvoch typov, podľa toho, ako vyzerajú v strome. Tieto dva typy ciest vidíme na obrázku nižšie.

- Cesta sa začína v nejakom vrchole a končí v jeho podstrome. Táto cesta zodpovedá modrej ceste na obrázku.
- Cesta v nejakom vrchole „mení smer“: oba konce cesty sú v podstrome tohto vrchola. Navyše, oba konce patria do podstromov *rôznych detí* tohto vrchola. Tento typ cesty zodpovedá žltej ceste na obrázku.



Všimnime si, že cesty prvého typu už predchádzajúce prehľadávanie spracuje: pre každý vrchol počítame aká je cesta s najmenšou estetickosťou, ktorá začína v ňom a končí v jeho podstrome.
Zostáva nám teda spracovať cesty druhého typu. Všimnime si, že sa vlastne jedná o spojenie dvoch ciest prvého typu, ktoré obe začínajú v rovnakom vrchole, pričom z tohto vrchola idú do rôznych detí.
Takže pre každý vrchol sa nám zostáva pozrieť na dvojice detí a na cesty, ktoré končia v ich podstromoch. Avšak, týchto ciest môže byť stále veľmi veľa. Nám však stačí nájsť len cestu s najmenšou estetickosťou. Takže nám netreba prehľadávať všetky cesty ktoré končia v podstrome daného dieťaťa, iba tie, ktoré majú šancu patriť do cesty s najmenšou estetickosťou. Zjavne nás teda zaujíma najmenej estetická cesta ktorá v dietati končí.
Následne, cestu druhého typu s najmenšou estetickosťou ktorá mení smer v danom vrchole, nájdeme spojením dvoch najmenej estetických ciest (do ktorých estetickosti zahrnieme aj estetickosť hrany do dieťaťa), ktoré vedú do podstromov rôznych detí.
Takže, aby sme si to zhrnuli, robíme prehľadávanie do hĺbky, a pre každý vrchol spočítame:

- Najmenšiu estetickosť nejakej cesty (prvého či druhého typu), ktorá sa celá nachádza v jeho podstrome.
- Najmenšiu estetickosť cesty prvého typu, ktorá začína v tomto vrchole a končí v jeho podstrome.

Odpoveď na druhú otázku spočítame tak, ako v predchádzajúcom prehľadávaní.
Aby sme zistili odpoveď na prvú otázku, pozrieme sa na odpovede jednotlivých detí. Najskôr nájdeme na najmenšiu estetickosť cesty v podstrome, ktorá nejde cez aktuálny vrchol – to je len minimum odpovedí prvého typu. Ostáva nám zistiť najmenšiu estetickosť cesty, ktorá ide cez aktuálny vrchol.
Na to si zistíme dve minimálne estetickosti ciest ktoré začínajú v aktuálnom vrchole a končia v podstromoch *rôznych* detí. Toto sú vlastne dve najmenšie hodnoty z odpovede na druhú otázku plus estetickosť cesty do daného dieťaťa.
Na toto všetko nám stačí jedno prehľadávanie a riešenie má tak lineárnu časovú zložitosť, čo stačí na vyriešenie celej úlohy. Implementovať ho ide napríklad takto:

Listing programu (Python)

```
import sys
sys.setrecursionlimit(200000)

def najdi_najmensiu(v, prisli_sme_z, hrany):
    najmensia_zo_mna = 0
    druha_najmensia = 0
    najmensia_celkovo = 0
    for w, e in hrany[v]:
        if w != prisli_sme_z:
            vysledok = najdi_najmensiu(w, v, hrany)
            najmensia_celkovo = min(najmensia_celkovo,
                                    vysledok[1])
            if najmensia_zo_mna > vysledok[0] + e:
                druha_najmensia = najmensia_zo_mna
                najmensia_zo_mna = vysledok[0] + e
            elif druha_najmensia > vysledok[0] + e:
                druha_najmensia = vysledok[0] + e

    najmensia_celkovo = min(najmensia_celkovo,
                            najmensia_zo_mna + druha_najmensia)
    return (najmensia_zo_mna, najmensia_celkovo)

n = int(input())
hrany = [[] for _ in range(n)]

E = 0

for _ in range(n - 1):
    a, b, e = map(int, input().split())
    a -= 1
    b -= 1
    hrany[a].append((b, e))
    hrany[b].append((a, e))
    E += e

min_estetickost = najdi_najmensiu(0, 0, hrany)[1]

print(E - 2 * min_estetickost)
```

A-I-3 Horské dráhy

Začnime riešením za 5 bodov. Ako vieme súťažnú úlohu vyriešiť, ak navyše oproti pôvodnému zadaniu dostaneme záruku, že chodníky popri horskej dráhe tvoria strom?
V tomto špeciálnom prípade platí, že od stanice 1 na stanicu n vedie práve jedna postupnosť chodníkov. (Tú vieme nájsť ľubovoľným prehľadaním vstupného grafu, napr. do hĺbky.)
Pre každý úsek horskej dráhy, ležiaci na tejto ceste, sa môžeme pozrieť, či je orientovaný v smere, ktorým chceme ísť. Ak nie, musíme tento úsek nutne ísť po chodníku. Ostávajú úseky horskej dráhy, ktoré sú orientované správnym smerom. Tie si vieme rozdeliť na navzájom nezávislé bloky – každý blok bude jedna skupina úsekov, ktoré na seba nadväzujú.
No a poslednou úvahou, ktorú potrebujeme spraviť je, že každý blok buď prejdeme celý naraz horskou dráhou (na jeden lístok), alebo ho pôjdeme celý pešo. Totiž ak už si kúpime lístok, zjavne nemá zmysel použiť ho len

na časť bloku a potom ešte musieť platiť niečo navyše za prejdenie zvyšku bloku. Pre každý blok zvlášť preto porovnáme cenu týchto dvoch možných riešení a vyberieme si lacnejšie z nich.
Keďže úsekov horskej dráhy je v tomto špeciálnom prípade len $n - 1$, vieme všetky kroky vyššie uvedeného riešenia implementovať v čase $O(n)$.

Pomalšie riešenie plnej úlohy

Vo všeobecnej verzii úlohy budeme hľadať nejakým spôsobom obmedzenú najkratšiu/najlacnejšiu cestu vo všeobecnom ohodnotenom grafe. Základným algoritmom pre takýto typ úloh je Dijkstrov algoritmus. Ak ho ešte nepoznáte, silne odporúčame prečítať si najskôr niečo o ňom, a až potom pokračovať v čítaní tohto vzorového riešenia. Jedno miesto, kde si o ňom môžete prečítať, je tu: https://school.ksp.sk/courses/graphs/graphs1-dijkstra/

Najkratšiu cestu nemôžeme hľadať priamo v grafe zo vstupu, tam nám to komplikuje to, že máme možnosť za jednu cenu prejsť po viacerých nadväzujúcich hranách.
Jednou z možností, ako tento problém odstrániť, je prerobiť vstupný graf na iný tak, aby v tom novom hrany a ich ceny ozaj zodpovedali tomu, ako môžu naši protagonisti cestovať.
Vrcholy nového grafu budú tie isté ako doteraz: n staníc. Hrany zostrojíme nasledovne: Pre každý chodník a každý smer pridáme orientovanú hranu zodpovedajúcu prejdeniu pešo po ňom. No a potom sa pre každú stanicu x pozrieme na to, pre ktoré iné stanice y sa dá dostať z x do y na jeden lístok horskou dráhou. Množinu všetkých takýchto y vieme zostrojiť obyčajným prehľadaním pôvodného vstupného grafu, pričom začneme v x a môžeme sa hýbať len po hranách predstavujúcich úseky horskej dráhy v smere jazdy. Pre každé takto nájdené y budeme mať v novom grafe orientovanú hranu $x \to y$ s cenou k (čo je cena jedného lístka).
V takto zostrojenom grafe už zjavne platí, že každá cesta má dĺžku rovnú cene, za ktorú tú cestu vieme v pôvodnej úlohe prejsť. A preto stačí spustiť na ňom Dijkstrov algoritmus a ním nájsť najkratšiu cestu zo stanice 1 do stanice n.
Pripomíname, že na vstupe sme dostali $m \leq n^2$ úsekov horskej dráhy. Každé jedno prehľadávanie tohto vstupného grafu bude mať časovú zložitosť $O(n + m)$.
Náš nový graf môže mať v najhoršom prípade až $\Theta(n^2)$ hrán – napr. ak sa dá horskou dráhou dostať úplne odvšadiaľ všade, budeme v ňom mať rádovo n^2 hrán.
Celé zostrojenie nového grafu nám teda bude trvať $O(nm + n^2)$. Následné spustenie Dijkstrovho algoritmu na ňom vieme implementovať tak, aby bežalo v čase $O(n^2)$, a teda celková časová zložitosť tohto riešenia ostane $O(nm + n^2)$.
(Ak by sme použili implementáciu Dijkstrovho algoritmu vhodnejšiu pre riedke grafy, pribudol by nám ešte jeden $\log n$ v časovej zložitosti.)

Efektívne riešenie plnej úlohy

Na predchádzajúcom riešení bol neefektívny len jeden krok – ten, že nový graf mohol mať rádovo viac hrán ako ten pôvodný, lebo sme v ňom spájali každé x s každým y. Efektívnejšie riešenie teraz dostaneme tak, že tento krok predchádzajúceho riešenia spravíme šikovnejšie.
Tentokrát budeme mať v našom novom grafe nie n, ale rovno $2n$ vrcholov. Pre každú stanicu budeme mať jeden vrchol zodpovedajúci tomu, že sme v nej *bez lístka* a jeden vrchol zodpovedajúci tomu, že tam sme *s lístkom*.
V každej stanici sa môžeme za cenu k dostať z vrcholu bez lístku do vrcholu s lístkom (kúpime si lístok) a za cenu 0 sa dostať naspäť (zahodíme lístok, ktorý už nechceme ďalej používať).
Ak sme v stanici bez lístku, môžeme sa pohnúť po ľubovoľnom chodníku z nej a zaplatíme za to príslušnú sumu. Pridáme teda, rovnako ako v pomalšom riešení, orientované hrany, ktoré zodpovedajú týmto pohybom.
A ak sme v stanici s lístkom, môžeme sa pohnúť po ľubovoľnej odtiaľto vedúcej horskej dráhe. Tento pohyb je už zadarmo (už máme lístok) a do cieľovej stanice opäť dorazíme do jej verzie s lístkom. Do nového grafu teda pridáme zodpovedajúcu orientovanú hranu s cenou 0.
Takto zostrojený graf má $2n$ vrcholov. Máme v ňom m hrán vedúcich medzi vrcholmi s lístkom a $2m$ hrán medzi vrcholmi bez lístka, teda dokopy stále len $O(m)$ hrán. Celý ho vieme vytvoriť v čase $O(n + m)$, teda lineárnom od veľkosti vstupu. Najpomalšou časťou riešenia bude následné spustenie Dijkstrovho algoritmu, to bude mať časovú zložitosť $O((n + m) \log n)$.

Listing programu (C++)

```cpp
#include <iostream>
#include <vector>
#include <queue>
using namespace std;
using pii = pair<int, int>; // usporiadaná dvojica celých čísel

int main(){
    int N, M, K;
    cin >> N >> M >> K;

    // stanici číslo i (od 1 po n) zodpovedajú vrcholy 2i-2 (bez lístka) a 2i-1 (s lístkom)
    vector<vector<pii>> nasledovnici(2*N);
    for(int i=0; i<M; i++){
        int X, Y, D;
        cin >> X >> Y >> D;
        // hrany zodpovedajúce pešiemu presunu po chodníku v každom smere
        nasledovnici[2*X-2].push_back({2*Y-2, D});
        nasledovnici[2*Y-2].push_back({2*X-2, D});
        // hrana zodpovedajúca jazde zadarmo s už kúpeným lístkom
        nasledovnici[2*X-1].push_back({2*Y-1, 0});
    }
    // kúpa a predaj lístka v každej stanici
    for(int i=1; i<=N; i++){
        nasledovnici[2*i-2].push_back({2*i-1, K});
        nasledovnici[2*i-1].push_back({2*i-2, 0});
    }

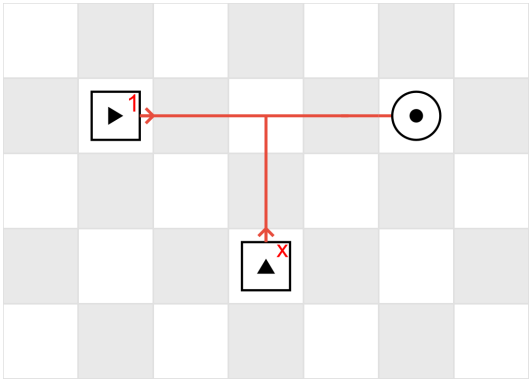
    // Dijkstrov algoritmus na nájdenie cesty z vrcholu 0 do vrcholu 2n-2
    vector<bool> objaveny(2*N, false);
    objaveny[0] = true;
    vector<int> vzdialenosti(2*N);
    vzdialenosti[0] = 0;
    // minimová halda so záznammi v tvare {vzdialenost', vrchol}
    priority_queue<pii, vector<pii>, greater<pii>> halda;
    halda.push({0, 0});
    while(!halda.empty()){
        int vzdialenost = halda.top().first;
        int vrchol = halda.top().second;
        halda.pop();
        if(vzdialenosti[vrchol] < vzdialenost) continue;
        if(vrchol == 2*N-2) break;
        for(pii hrana: nasledovnici[vrchol]){
            int sused = hrana.first;
            int nova_vzdialenost = vzdialenost + hrana.second;
            if((!objaveny[sused]) or vzdialenosti[sused] > nova_vzdialenost){
                objaveny[sused] = true;
                vzdialenosti[sused] = nova_vzdialenost;
                halda.push({nova_vzdialenost, sused});
            }
        }
    }

    cout << vzdialenosti[2*N-2] << endl;
}
```

A-I-4 Optické hradlá

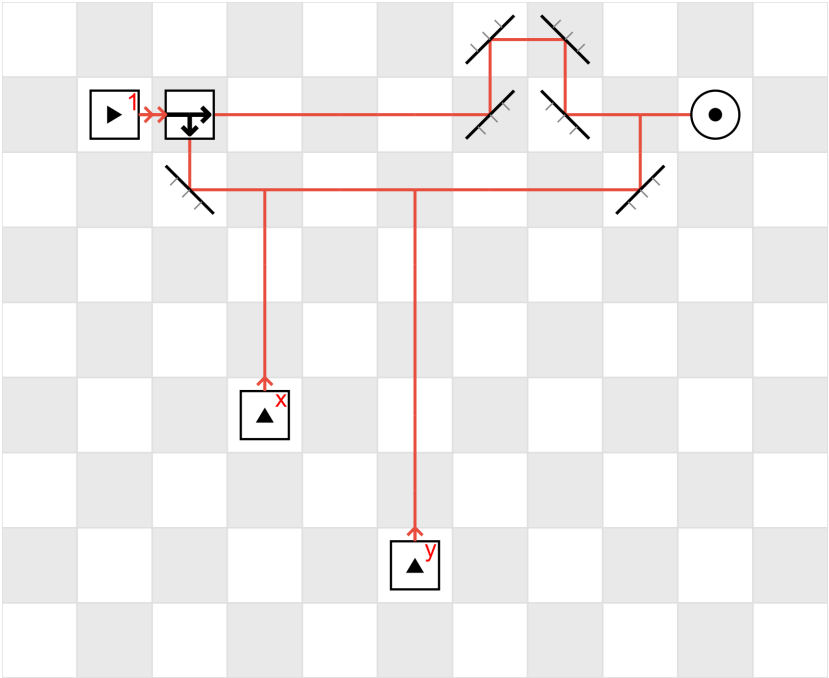
Podúloha A (1 bod): konštrukcia hradla NOT

Emitory namierime tak, aby sa ich lúče zrazili. Kolektor umiestnime ďalej do dráhy lúča z emitoru 1. Takáto konfigurácia sa bude správať ako NOT: Ak je emitor x aktívny, lúče sa zrazia a do kolektora nič nepriletí. A naopak, ak je x neaktívny, lúč z emitora 1 zasiahne kolektor.



Podúloha B (2 body): konštrukcia hradla OR

Z lúča z emitora 1 si rozdeľovačom spravíme dve kópie.
Ku dráhe jednej kópie umiestnime emitory x a y tak, aby sa ich lúč, ak bude vystrelený, s touto kópiou zrazil. Ak teda aspoň jeden z emitorov x a y bude aktívny, táto kópia zanikne.
Následne nasmerujeme na to isté miesto obe kópie lúča z emitora 1. Ak ešte stále obe existujú, zrazia sa a zaniknú. Ak nie, prvá kópia bude pokračovať ďalej. Tam jej do dráhy postavíme kolektor. Takto dosiahneme, že kolektor bude zasiahnutý práve vtedy, ak sa dve kópie lúča nezrazili – teda ak aspoň jeden z emitorov x a y bol aktívny.

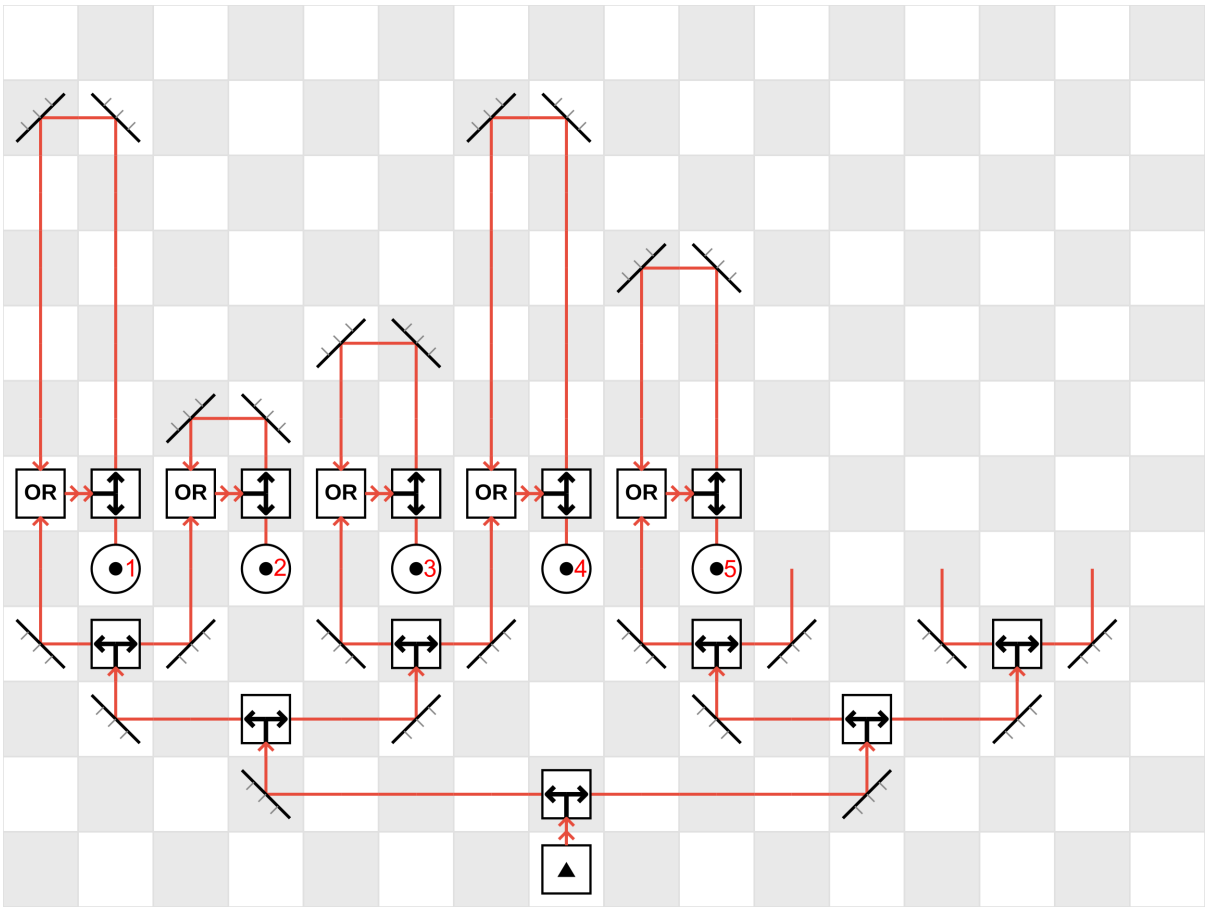


Všimnite si, že emitory x a y sú umiestnené tak, aby ich lúče prišli na miesto potenciálnej zrážky naraz s kópiou lúča z emitoru 1. (Emitor y sme tiež mohli umiestniť o dva riadky vyššie a potom jeho lúč spomaliť pomocou zrkadiel.)
A ešte podotkneme, že ak sú aj x aj y aktívne, tak sa lúč z emitora y nemá s čím zraziť, a teda preletí ďalej. Toto nám nevadí, lebo horná kópia lúča z emitoru 1 je tou dobou už ďalej vpravo.

Podúloha C (3 body): rytmické blikanie

Základnou myšlienkou toho, ako spraviť blikanie s periódou $2p$, je spraviť si cyklus tvorený $2p$ políčkami, v ktorom bude dokola lietať laserový lúč. Potom stačí na jedno miesto v tomto cykle vložiť rozdeľovač. Ten vždy,

keď ním lúč prejde, vyrobí druhú kópiu, ktorá vyletí von (a tam zasiahne príslušný kolektor).
Pre každý kolektor budeme mať jeden takýto cyklus. A už máme len jeden problém: na začiatku v nich samozrejme žiaden laserový lúč nelieta, potrebujeme do nich teda nejaký dostať. Zjavne teda potrebujeme rozkopírovať lúč vystrelený z emitora a doviesť do každého cyklu jednu kópiu. Ak to navyše zvládneme spraviť tak, aby do všetkých cyklov prišiel lúč v tom istom okamihu a aby ony následne všetky naraz prvýkrát vystrelili lúč smerom von, budeme už mať všetko ako má byť.
Výroba k kópií jedného lúča sa dá robiť postupne: vyrobíme druhý lúč, z neho tretí, z neho štvrtý, ..., a potom už len každý cyklus umiestnime tak ďaleko, aby k nim všetky lúče prišli naraz.
Ukážeme si ale pre zaujímavosť efektívnejšiu metódu, ktorá je aj rýchlejšia, aj potrebuje výrazne menšiu podložku: kopírovať môžeme aj paralelne, konštrukciou pripomínajúcou binárny strom.
Na obrázku je príklad kompletnej konštrukcie pre $n = 5$ a dĺžky cyklov 12, 4, 6, 12 a 8.



Opäť ešte pridáme pár technických detailov.
Náš pôvodný lúč sme si rozkopírovali rovno na 8 a potom sme pravé tri lúče nechali len tak voľne odletieť. Komu sa to nepáči, môže v pravej časti stromu niektoré rozdeľovače zmeniť na zrkadlá odrážajúce len doľava a vyrobiť si lúčov naozaj len päť.
Všimnite si ešte použitie hradla OR na mieste, kde prvýkrát lúč vstúpi do cyklu. Vtedy ho hradlo OR dostane na vstupe zdola a prepošle ho doprava. Po každom ďalšom obehu cyklu už tento lúč do hradla vstúpi zhora.

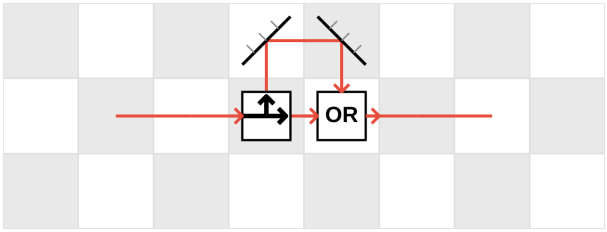
Podúloha D (4 body): súčet

Základná myšlienka nášho riešenia bude nasledovná:

1. Pre každý vstup i si vyrobíme obvod, ktorý jeho jeden vystrelený lúč nakopíruje na s_i lúčov.
2. Zo vždy-aktívneho emitoru si nakopírujeme k lúčov.

3. Všetky lúče prvého typu pošleme naproti všetkým lúčom druhého typu.
4. Ak nám prežijú nejaké lúče druhého typu, znamená to, že k bolo ostro väčšie ako súčet s_i aktívnych vstupov. Ak žiadne lúče neprežijú, tak k bolo menšie alebo rovné ako aktívny súčet.

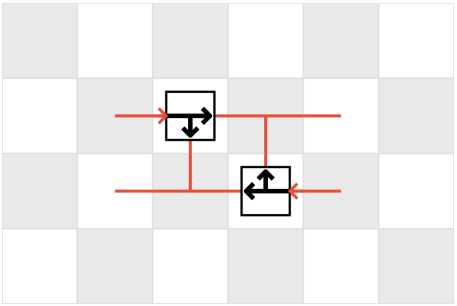
Začneme kopírovacím obvodom, ktorý z jedného lúča vyrobí x. Presnejšia vlastnosť, ktorú od neho budeme chcieť, je, aby všetkých x výsledných lúčov letelo postupne za sebou po tej istej polpriamke.
Existuje nespočetné množstvo možností, ako spraviť takúto konštrukciu. Napr. v predošlej úlohe sme ukázali, ako si rozkopírovať jeden lúč na x rovnobežne letiacich. Potom už stačí každý z nich pomocou zrkadiel ináč spomaliť a následne ich opäť pospájať na jednu priamku (to bude vyzerať rovnako ako rozdeľovanie, len namiesto rozdeľovačov použijeme na spájanie hradlá OR).
Ukážeme si ešte jednu „minimalistickú“ konštrukciu takéhoto obvodu:



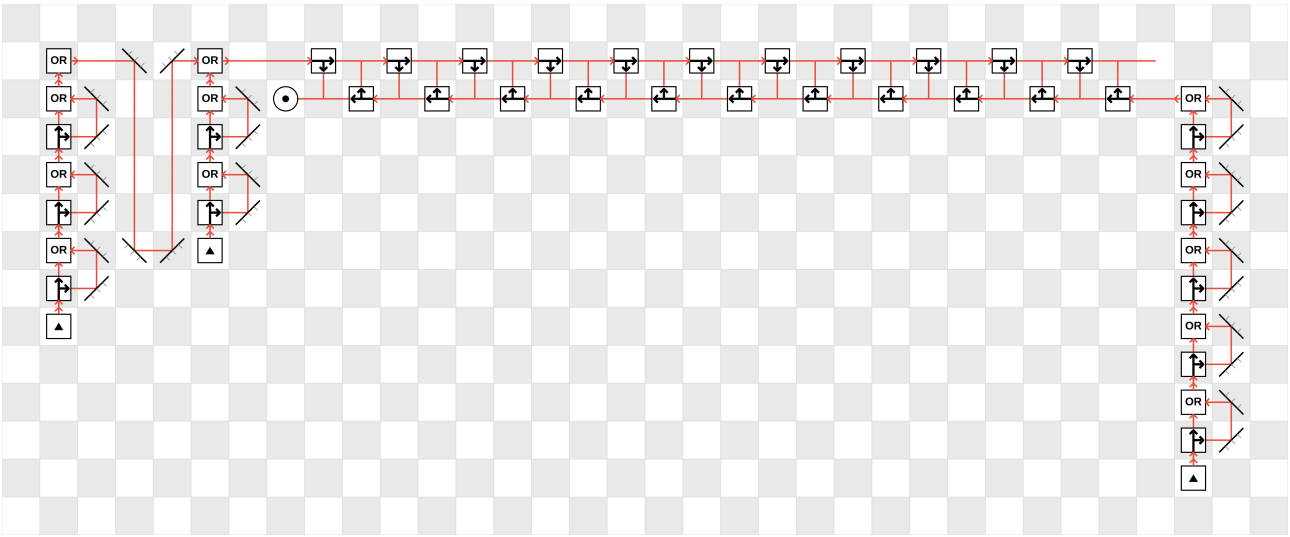
Tento maličký obvod pridá jednu novú kópiu lúča. Ak zľava priletí jeden lúč, doprava odletia dva (pričom medzi nimi bude jeden tik pauza). Ak ich zľava priletí postupne 10 (s tým, že medzi po sebe idúcimi je vždy tik pauza), doprava ich odletí 11 – každý z nich skopírujeme, ale až na úplný začiatok a koniec nám hradlo OR zase naspäť zlúči kópiu jedného lúča s rovno idúcim originálom nasledujúceho.
Stačí teda za emitor, z ktorého chceme mať x lúčov, navkladať $x - 1$ kópií tohoto jednoduchého obvodu. Ak tento emitor bude aktívny, vznikne nám z jeho jedného lúča x po sebe letiacich lúčov. A ak aktívny nebude, nikde nič nevznikne, a teda ďalej poletí lúčov nula.

Bloky lúčov potenciálne vychádzajúce z takýchto obvodov pre jednotlivé vstupné emitory si samozrejme vieme všetky pospájať na jednu polpriamku. Stačí výstupy jednotlivých obvodov spomaliť tak, aby aj keby boli všetky aktívne, tvorili na výslednej polpriamke ich lúče navzájom disjunktné, po sebe idúce úseky.

Teraz je načase rozmyslieť si zrážanie lúčov. Jedna možnosť by bola polpriamku so vstupnými lúčmi umiestniť kolmo na polpriamku s k lúčmi. Pri takomto riešení by sme však ešte potrebovali dosiahnuť, aby všetky vstupné lúče prišli na miesto zrážky bezprostredne po sebe a navyše aj vždy v tom istom čase, bez ohľadu na to, ktoré vstupy sú aktívne. To zatiaľ nevieme spraviť a pôsobí to nepríjemne.
Omnoho nádejnejšie vyzerá myšlienka poslať si obe sady lúčov naproti. Tam nám bude úplne jedno, aké sú medzi nimi medzery. Prvý zľava sa niekedy niekde zrazí s prvým sprava, potom druhý s druhým, a tak ďalej, až kým sa nám jedna strana neminie.
Takýto obvod by sme už vedeli spraviť, ale je tu ešte jeden problém. Totiž, ak pošleme vstupné lúče zľava a k kontrolných lúčov sprava, potrebujeme vedieť zistiť, či po všetkých zrážkach nejaký lúč pokračoval doľava. Tento lúč ale pôjde naspäť presne tou dráhou, ktorou prileteli vstupné lúče, a tam mu už nemáme ako do cesty vložiť kolektor tak, aby ho netrafili už vstupné lúče.
Potrebujeme teda ďalší trik. A jedným z nápadov, ktoré zafungujú, je neposlať oba lúče priamo proti sebe, ale jeden z nich posunúť o riadok dodola. Potom môžeme zrážky protiidúcich lúčov robiť nasledovným malým obvodom:



Kým sa nezjavia žiadne lúče, nič sa nedeje. Ak priletí lúč len zľava (v hornom riadku), rozdvojí sa, jedna kópia pokračuje plynule ďalej a druhá odletí dodola. (Aby nám inde nespravila nechcené veci, oplatí sa jej následne do cesty postaviť zarážku – napr. ďalší rozdeľovač, ale so vstupom z inej strany.) V princípe to isté sa stane, ak priletí lúč len sprava (v dolnom riadku). Ale ak priletia oba naraz, oba sa naraz rozdelia, potom na oboch križovatkách nastane zrážka a ďalej už nepokračuje ani jedným smerom nič.
Takýchto obvodov naskladáme do riadku za seba dostatočne veľa (dosť na to, aby všetky lúče z oboch strán vošli dnu skôr, ako by sa hociktorý z nich mohol dostať von).



Na obrázku je príklad celého počítača pre nasledovné parametre: $n = 2$, $k = 6$ a sily vstupov $s_1 = 4$ a $s_2 = 3$. V ľavej časti máme dva vstupné emitory. Ich obvody sú zapojené za seba tak, aby sa ich lúče určite nepomiešali. V strednej časti máme obvody robiace zrážky oproti letiacich lúčov. V pravej časti máme jeden vždy aktívny emitor a nad ním obvod, ktorý nám vyrobí k za sebou idúcich lúčov. Na konci riadku, ktorým potom letia tieto lúče doľava, je kolektor kontrolujúci, či sa k nemu nejaký z nich dostane.

ŠTYRIDSIATY DRUHÝ ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Michal Forišek, Paulína Smolárová
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: NIVAM – Národný inštitút vzdelávania a mládeže, Bratislava 2026