76. ročník Matematické olympiády – 2026/2027

Řešení úloh domácího kola kategorie P

P-I-1 Skříň

Klíčové pozorování je, že $\text{nsd}(x, y) > 1$ platí právě tehdy, když čísla x a y mají společného prvočíselného dělitele. Látku x lze tedy do skříně uložit právě tehdy, když žádné prvočíslo dělící x nedělí žádnou z látek, které už uvnitř stojí.

Z pravidla pro ukládání navíc plyne, že obsah skříně je v každém okamžiku po dvou nesoudělný. Každé prvočíslo proto dělí nejvýše jednu látku ve skříni. Stačí si tedy pro každé prvočíslo pamatovat, jestli je právě teď zastoupeno ve skříni, nebo ne. Budeme si udržovat dvě pole: *obsazeno*[p] říká, zda je ve skříni nějaká látka dělitelná prvočíslem p, a *veSkrini*[x] říká, zda látka x zrovna ve skříni stojí.

Požadavky pak obsloužíme takto:

- *Uložení látky x.* Rozložíme x na prvočinitele. Je-li některý z nich obsazený, vypíšeme NE a nic dalšího neděláme. Jinak všechny prvočinitele čísla x označíme v poli *obsazeno* jako obsazené, látku x v poli *veSkrini* jako uloženou a vypíšeme ANO.
- *Vrácení látky x.* Není-li látka x ve skříni, neděláme nic. Jinak všechny prvočinitele čísla x označíme v poli *obsazeno* jako volné a látku x v poli *veSkrini* jako vrácenou.

Případ, kdy Honza zkouší uložit látku, která už ve skříni stojí, nemusíme ošetřovat zvlášť. Všechny její prvočinitele jsou obsazené právě jí samotnou a první bod tedy správně vypíše NE. Zároveň když odstraňujeme látku x, která už ve skříni je, uvolnění všech prvočísel dělících x nic nepokazí, protože jsou látky ve skříni vždy po dvou nesoudělné.

Zbývá umět rychle rozkládat na prvočinitele. Upravíme Eratosthenovo síto (viz knihovnu základních algoritmů MO-P, *https://mo.mff.cuni.cz/p/knihovna.html*) tak, aby si pro každé $n \leq X$, kde $X = 10^6$, zapamatovalo jeho nejmenší prvočíselný dělitel: když síto prochází násobky prvočísla p, zapíše p do všech dosud prázdných políček. Protože prvočísla procházíme vzestupně, zapíše se do každého políčka opravdu nejmenší prvočíselný dělitel. Síto běží v čase $\mathcal{O}(X \log \log X)$. Pro každé $x \leq 10^6$ pak rozklad získáme opakovaným dělením v čase $\mathcal{O}(\log x)$. Každé $x \leq 10^6$ má přitom nejvýše sedm různých prvočíselných dělitelů, protože $2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 = 510\,510$, kdežto přinásobením 19 bychom už milion překročili.

Celková časová složitost je $\mathcal{O}(X \log \log X + q \log X)$, paměťová $\mathcal{O}(X)$.

Síto přitom není nezbytné: rozkládat lze i postupným dělením prvočísly do $\sqrt{X} = 1000$, kterých je pouhých 168. To je sice asymptoticky horší, ale při daných mezích to bohatě stačí.

Ještě si všimněme, kolik látek může ve skříni stát najednou. Každá z nich potřebuje aspoň jedno vlastní prvočíslo, takže nejvýše $\pi(10^6) = 78\,498$ lahviček.

Pomalejší řešení

Nejpřímočařejší postup je držet si obsah skříně v poli a při každém uložení projít celý jeho obsah a počítat $\mathrm{nsd}(x, y)$. To zvládne první sadu, kde je $q \leq 1\,000$, takže i látek ve skříni je nejvýše tolik. Na plných datech je to ale až $q \cdot \pi(10^6)$, tedy zhruba $1{,}4 \cdot 10^{10}$ operací.

Ve druhé sadě je $x \leq 5\,000$, takže existuje jen 669 použitelných prvočísel. Pole *obsazeno* má tedy jen 5 001 položek a rozkládat stačí dělením prvočísly do 70 ($\sqrt{5000} < 71$); síto vůbec není potřeba. Ve třetí sadě se nic nevrací, takže pole *veSkrini* vůbec nepotřebujeme a s ním odpadá všechna režie s ním spojená.

Pozor na lákavé, ale stále příliš pomalé vylepšení: protože $1009 \cdot 1013 > 10^6$, má každé $x \leq 10^6$ nejvýše jednoho prvočíselného dělitele většího než 1000. Malých prvočísel je jen 168, u těch si obsazenost pamatujeme přímo, a hledat zbývá jediné velké prvočíslo. Kdo ho ale hledá průchodem přes obsah skříně, má pořád složitost $\mathcal{O}(q \cdot |S|)$, kde $|S|$ je počet látek ve skříni, a na plná data to nestačí. I toto poslední hledání je potřeba udělat indexací do pole (nebo aspoň vyhledáním v množině či hašovací tabulce).

Program (Python 3):

```
#!/usr/bin/env python3

import sys

# horní mez na evidenční čísla
X = 1000000

def main():
    cisla = map(int, sys.stdin.buffer.read().split())
    q = next(cisla)

    # Síto nejmenších prvočíselných dělitelů: pro každé prvočíslo projdeme
    # jeho násobky a u těch, které ještě žádného dělitele nemají, si ho
    # poznamenáme. Prvočíslo je právě takové číslo, na které při tomhle
    # postupu dojde řada s dosud prázdným políčkem.
    spf = [0] * (X + 1)
    for i in range(2, X + 1):
        if spf[i] == 0:
            for j in range(i, X + 1, i):
                if spf[j] == 0:
                    spf[j] = i

    obsazeno = [False] * (X + 1)
    veSkrini = [False] * (X + 1)
    vystup = []

    for _ in range(q):
        typ = next(cisla)
        x = next(cisla)

        if typ == 1:
            # rozkládáme x na prvočinitele a cestou kontrolujeme,
            # jestli některý z nich už není obsazený.
            prvocinitele = []
            lze = True
            y = x
```

```
            while y > 1:
                p = spf[y]
                if obsazeno[p]:
                    lze = False
                    break
                prvocinitele.append(p)
                while y % p == 0:
                    y = y // p

            if lze:
                for p in prvocinitele:
                    obsazeno[p] = True
                veSkrini[x] = True
                vystup.append("ANO\n")
            else:
                vystup.append("NE\n")

        else:
            # jen když látka ve skříni opravdu stojí, uvolníme
            # všechny její prvočinitele.
            if veSkrini[x]:
                veSkrini[x] = False
                y = x
                while y > 1:
                    p = spf[y]
                    obsazeno[p] = False
                    while y % p == 0:
                        y = y // p

    sys.stdout.write("".join(vystup))

main()
```

P-I-2 Bonsajová zahrada

Na vstupu je dán strom o N vrcholech s ohodnocenými hranami, jehož vrcholy tvoří jednotlivé rozcestníky. My si v něm vybereme nějakou cestu (jednoznačně danou počátečním a koncovým rozcestníkem) a na ní prohodíme ohodnocení. Byl-li tedy součet hran na této cestě s, nyní je $-s$, tedy o $2 \cdot (-s)$ vyšší. Tímto prohozením se tedy součet ohodnocení všech hran zvýší právě o $2 \cdot (-s)$. Naším cílem tedy bude najít cestu takovou, že její součet s je nejmenší možný (takové cestě budeme říkat *nejzápornější*).

Než se ale podíváme na samotné řešení, je potřeba si promyslet jeden implementační detail. Součet všech hran totiž může být až řádově 10^{14}. V některých jazycích je tedy potřeba použít 64 bitové proměnné, například *long long* v C++.

Všechny hrany mají kladnou hodnotu

Pokud mají všechny hrany kladnou hodnotu, je zjevně nejlepší vybrat cestu bez hran (tj. se stejným počátečním a koncovým rozcestníkem), se součtem $s = 0$.

Všechny hrany mají zápornou hodnotu

Představme si, že znegujeme všechny hrany. Tím úlohu převedeme na ekvivalentní úlohu: V ohodnoceném stromě chceme najít nejdelší cestu (hodnoty hran

interpretujeme jako jejich délky). Tuto úlohu lze vyřešit následovným trikem: Začneme v libovolném vrcholu v a pomocí prohledávání do hloubky[1] v čase $\mathcal{O}(N)$ nalezneme vrchol a, který je od něj nejvzdálenější. Přesněji řečeno, v průběhu prohledávání si udržujeme informaci o délce cesty mezi v a aktuálním vrcholem: když se posuneme po hraně dopředu, její délku přičteme, a když se vracíme, tak ji odečteme (nebo si prostě v zásobníku pamatujeme délku cesty do vrcholu, do kterého se vracíme). Průběžně si přitom udržujeme maximum z nalezených délek a číslo vrcholu, do nějž cesta této délky vede.

Z vrcholu a poté spustíme druhé prohledávání do hloubky a nalezneme nejvzdálenější vrchol b. Cesta mezi vrcholy a a b je pak zaručeně nejdelší v celém stromě. Toto tvrzení není úplně jednoduché dokázat, ale jelikož ho stejně v obecném řešení nevyužijeme, necháme vám ho k zamyšlení.

Pomalé řešení

Pomocí prohledávání do hloubky můžeme úlohu řešit i obecně: Prostě projdeme všechny dvojice vrcholů v a u a z v pustíme prohledávání do hloubky. Na rozdíl od předchozí podúlohy si neudržujeme (váženou) délku cesty, ale rovnou součet hodnot na hranách cesty z v do aktuálního vrcholu. Ve chvíli, kdy dorazíme do v, se tak dovíme součet hodnot na cestě mezi u a v. Z těchto hodnot pro všechny dvojice si pak stačí průběžně počítat minimum.

Jelikož dvojici vrcholů u a v můžeme vybrat $\mathcal{O}(N^2)$ způsoby a pro každou z nich provedeme prohledání s časovou složitostí $\mathcal{O}(N)$, výsledná časová složitost je $\mathcal{O}(N^3)$, což postačuje pro 3. podúlohu ($N \leq 100$).

Toto řešení lze snadno zrychlit: Během jednoho prohledání do šířky z vrcholu v si můžeme určit součty cest do všech ostatních vrcholů, nejen do jednoho předem vybraného. Prohledávání tedy stačí tedy spustit z každého vrcholu jen jednou. Toto řešení má časovou složitost $\mathcal{O}(N^2)$ a postačuje pro 4. podúlohu ($N \leq 2000$).

Obecné řešení

Začneme tím, že si strom zakořeníme; jeden vrchol si tedy vybereme jako kořen a představíme si, že je úplně nahoře, jeho sousedi jsou těsně pod ním, jejich sousedi ještě níže, atd. *Podstrom* vrcholu v se pak skládá z v a vrcholů, do kterých z v vede cesta, jejíž hrany všechny vedou směrem dolů.

Nyní krátká úvaha o tom, jak vypadá nejzápornější cesta C. Určitě má nějaký vrchol v, který je ve stromě nejvýš. Z něj pak vede po nejvýše dvou hranách dolů (možná jen po jedné, je-li v konec cesty C). Pro každého syna (tj. jeho soused, s kterým je spojen hranou vedoucí dolů) vrcholu v si můžeme spočítat nejzápornější cestu jdoucí z v do podstromu daného syna. Z takových cest potom stačí vybrat dvě nejmenší a jejich hodnoty sečíst, jelikož nemá smysl použít méně záporné cesty v vrcholu v dolů. Tento postup můžeme opakovat pro každý vrchol a pamatovat si, nejzápornější nalezenou cestu.

[1] O prohledávání do hloubky si můžete přečíst zde:
https://ksp.mff.cuni.cz/encyklopedie/grafy/

Úlohu tedy budeme řešit tzv. dynamickým programováním[2]. Pro každý vrchol v si budeme počítat dvě hodnoty $P[v]$ a $Q[v]$:

- $P[v]$ je hodnota nejzápornější cesty $C[v]$, která začíná ve vrcholu v a končí v některém z vrcholů v jeho podstromu (což zahrnuje případ, kdy končí ve vrcholu v a má tedy nulovou hodnotu).
- $Q[v]$ je hodnota nejzápornější cesty, která začíná ve vrcholu v, končí v některém z vrcholů v jeho podstromu, a nemá stejnou první hranu jako cesta $C[v]$ ($Q[v]$ opět může být nulové, například pokud z v dolů vede jen jedna hrana).

Pro každý list v zjevně platí $P[v] = Q[v] = 0$. Pro vnitřní vrchol v vypočítáme hodnoty následovně. Nechť $v_1, \ldots, v_k$ jsou synové vrcholu v, tj. jeho sousedi, s nimiž je spojen hranou vedoucí dolů, a nechť h_i je hodnota hrany mezi v a v_i. Nejzápornější cesta z v, která začíná hranou do v_i, má zjevně součet $h_i + P[v_i]$ – hranu vv_i musíme započítat a pak chceme pokračovat nejzáporněji, jak jen to jde. Z těchto hodnot pro $i = 1, \ldots, k$ stačí vybrat minimum, což nám dá hodnotu $P[v]$; přitom si stranou poznamenáme index i_0, pro který toto minimum dostáváme. Výjimkou je případ, kdy všechny tyto hodnoty jsou kladné, pak je nejlepší z v nikam nechodit a dostat tak $P[v] = 0$. Hodnotu $Q[v]$ spočítáme stejně, jen při výpočtu musíme vynechat syna v_{i_0}.

Hodnoty P a Q můžeme spočítat v jednom průchodu do hloubky: Vždy těsně před tím, než bychom se z vrcholu v vrátili, provedeme popsaný výpočet. V tomto okamžiku již máme spočítané hodnoty $P[v_i]$ pro syny v, ve výpočtu je tedy můžeme bez problémů použít. Časová složitost tohoto výpočtu je $\mathcal{O}(N)$ – všimněme si, že v každém vrcholu strávíme tolik času, kolik z něj vede hran do synů, a strom má celkem pouze $N - 1$ hran.

Hodnotou nejzápornější cesty tedy určíme jako minimum z hodnot $P[v] + Q[v]$ pro všechny vrcholy v. Časová složitost tohoto řešení je $\mathcal{O}(N)$.

Program (C++):

```
#include <iostream>
#include <vector>

struct edge {
        int v; // číslo sousedního vrcholu
        int h; // hodnota hrany
};

struct vertex {
        long long p, q; // proměnné P[v] a Q[v]
        std::vector<edge> edges; // hrany vedoucí do sousedních vrcholů
};

std::vector<vertex> vertices;
```

[2] O problematice dynamického programování si můžete přečíst více zde: *https://ksp.mff.cuni.cz/encyklopedie/dynamicke-programovani/*

```
// rekurzivní prohledávání do hloubky
void dfs(int i, int from) {
        long long min = 0;
        int min_id = -1;
        long long min2 = 0;

        // hledání "prvního" minima, tj. P[v]
        for (const edge &e: vertices[i].edges) {
                // přeskočíme otce, tj. vrchol, z kterého jsme rekurzí přišli
                if (e.v == from)
                        continue;

                // rekurze do syna
                dfs(e.v, i);

                long long tmp = e.h + vertices[e.v].p;
                if (tmp < min) {
                        min = tmp;
                        min_id = e.v;
                }
        }

        // hledání "druhého" minima, tj. Q[v]
        for (const edge &e: vertices[i].edges) {
                // přeskočíme otce a syna, přes kterého
                // vede "první" nejzápornější cesta
                if (e.v == from || e.v == min_id)
                        continue;

                long long tmp = e.h + vertices[e.v].p;
                if (tmp < min2)
                        min2 = tmp;
        }

        vertices[i].p = min;
        vertices[i].q = min2;
}

int main() {
        int N;
        std::cin >> N;
        vertices.resize(N);

        long long sum = 0; // součet hodnot všech hran

        for (int i = 0; i < N - 1; i++) {
                int a, b, h;
                std::cin >> a >> b >> h;

                // zadání indexuje vrcholy od 1, ale v řešení
                // se nám hodí indexovat od 0
                a--;
                b--;

                vertices[a].edges.push_back({b, h});
                vertices[b].edges.push_back({a, h});
                sum += h;
        }

        // samotný výpočet P[v] a Q[v]
        dfs(0, 0);
```

```
        long long minimum = 0;
        for (int i = 0; i < N; i++) {
                long long tmp = vertices[i].p + vertices[i].q;
                if (tmp < minimum)
                        minimum = tmp;
        }

        std::cout << sum - 2 * minimum << std::endl;
        return 0;
}
```

P-I-3 Skluzavky

Na vstupu máme ohodnocený slabě souvislý orientovaný graf.[1]

Skluzavky tvoří strom

V tomto případě vede mezi sloupy 1 a N právě jedna cesta. Tu můžeme najít například prohledáním do hloubky. U skluzavek na této cestě, které vedou v protisměru, musí jít Erik po schodech. Zároveň nám tyto skluzavky rozsekají zbytek cesty na úseky vedoucí po směru. Každý z těchto úseků by měl Erik buďto celý sjet po skluzavkách, nebo celý vyjít po schodech (nedává smysl část cesty jít a část se klouzat – v takovém případě by Erik vynaložil energii X a ještě nějakou navíc, než kdyby se pouze klouzal). Energie na vyjití nějakého úseku po schodech je součet délek skluzavek v tomto úseku. Tu na konci každého úseku porovnáme s X a vybereme méně náročnou variantu.

Prohledání stromu nám zabere $\mathcal{O}(N)$ času, následné projití cesty a spočítání nutné energie také, tedy toto řešení má časovou složitost $\mathcal{O}(N)$. Paměťová složitost je také $\mathcal{O}(N)$.

Pomalé řešení

V tomto případě už je více možností, jak se dostat od sloupu 1 k N, pořád ale budeme hledat nejkratší cestu v ohodnoceném grafu, k čemuž můžeme použít Dijkstrův algoritmus.[2]

Nejkratší cestu ale nechceme hledat přímo v zadaném grafu ze skluzavek – tam nám totiž situaci komplikuje klouzání, které umožňuje projít několik hran za jednotnou cenu. Proto si vytvoříme nový graf, ve kterém bude mít každá hrana jednoznačně danou cenu. Vrcholy v novém grafu budou pořád odpovídat sloupům, ale hrany budou odpovídat pohybům, které Erik může udělat. Chození po schodech reprezentujeme jednoduše – mezi každými dvěma vrcholy, mezi kterými vede skluzavka, vytvoříme obousměrnou hranu s cenou délky skluzavky. Klouzání po skluzavkách je komplikovanější – musíme pro každý vrchol zjistit, kam všude se dá dostat jedním sklouznutím. Když to budeme vědět, můžeme z každého vrcholu vést hrany do všech

[1] O grafech se můžete více dozvědět například zde:
https://ksp.mff.cuni.cz/encyklopedie/grafy/

[2] Jak funguje Dijkstrův algoritmus, si můžete přečíst například zde:
https://ksp.mff.cuni.cz/encyklopedie/halda-a-cesty/

vrcholů, kam se z něj dá dostat sklouznutím, s cenou X. Takto vytvořený graf bude mít až $\mathcal{O}(N^2)$ hran, protože z každého vrcholu může vést hrana do všech ostatních vrcholů. Dijkstrův algoritmus v něm tedy najde nejkratší cestu z vrcholu 1 do N v čase $\mathcal{O}(N^2 \log N)$.

Zbývá zjistit, kam vedou všechna možná sklouznutí z každého vrcholu. To zjistíme pro každý vrchol zvlášť – spustíme z něj prohledávání (například do hloubky) a budeme sledovat, kam všude se dá dostat pouze po směru skluzavek. To nám pro každý vrchol zabere $\mathcal{O}(M)$ času, celkem to tedy bude trvat $\mathcal{O}(NM)$ času. Celková časová složitost tohoto řešení je proto $\mathcal{O}(NM + N^2 \log N)$. Paměťová složitost je $\mathcal{O}(N^2)$.

Rychlé řešení

Myšlenku tvorby nového grafu neopustíme, musíme ale zmenšit počet přidávaných hran. Nemáme totiž kapacitu si pamatovat každé možné sklouznutí z každého vrcholu. Volbu sklouznutí si proto rozdělíme na dvě části – rozhodnutí, kdy se začít klouzat, a rozhodnutí, kam se klouzat – obojí má totiž pouze $\mathcal{O}(N)$ možností. Abychom rozlišili, jestli Erik zrovna chodí po schodech, nebo se klouže, původní graf zduplikujeme. V jedné kopii (nazvěme ji schodovou) budou hrany odpovídat chůzi po schodech, v druhé kopii (nazvěme ji *klouzací*) budou hrany odpovídat klouzání po skluzavkách. Každý sloup má odpovídající vrchol v obou kopiích, a tyto dva vrcholy jsou navzájem propojené. Hrany v rámci obou kopií odpovídají skluzavkám původního grafu, ale v klouzací kopii vedou pouze po směru skluzavek, zatímco ve schodové kopii jsou obousměrné. Abychom reprezentovali, že klouzání stojí energii X, bude hrana z vrcholu ve schodové kopii do odpovídajícího vrcholu v klouzací kopii mít cenu X. Hrany v rámci klouzací kopie, stejně jako hrana z vrcholu v klouzací kopii do odpovídajícího vrcholu ve schodové kopii, budou mít cenu 0 (jelikož jakmile jsme za klouzání jednou zaplatili, je zadarmo, dokud se nevrátíme do schodové kopie). Hrany v rámci schodové kopie budou mít cenu délky skluzavky, stejně jako v pomalém řešení.

Tento nový graf má $2N$ vrcholů, M hran v klouzací kopii, $2M$ hran ve schodové kopii, a $2N$ hran mezi kopiemi. Jelikož $N < M + 2$ (původní graf je souvislý), má nový graf stále $\mathcal{O}(N)$ vrcholů a $\mathcal{O}(M)$ hran. Dijkstrův algoritmus tedy najde nejkratší cestu z vrcholu 1 do N v čase $\mathcal{O}(M \log N)$. Vytvoření nového grafu zabere $\mathcal{O}(N + M)$, takže celková časová složitost je $\mathcal{O}(M \log N)$. Paměťová složitost je $\mathcal{O}(N + M)$.

Program (C++):

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;
using pii = pair<int, int>; // pii se převede na dvojici intů

int main(){
    int N, M, X;
    cin >> N >> M >> X;
    // Sloupu s číslem i přiřadíme vrcholy 2*i-2 a 2*i-1 v novém grafu
```

```
    // Budeme pamatovat, do jakých vrcholů se můžeme z každého vrcholu dostat:
    // Prvni hodnota je číslo cílového vrcholu, druhá je délka hrany
    vector<vector<pii>> naslednici(2*N);
    for(int i=0; i<M; i++){
        int S, K, D;
        cin >> S >> K >> D;
        // Hrany v rámci schodové kopie:
        naslednici[2*S-2].push_back({2*K-2, D});
        naslednici[2*K-2].push_back({2*S-2, D});
        // Hrany v rámci klouzací kopie:
        naslednici[2*S-1].push_back({2*K-1, 0});
    }
    // Hrany mezi kopiemi:
    for(int i=1; i<=N; i++){
        naslednici[2*i-2].push_back({2*i-1, X});
        naslednici[2*i-1].push_back({2*i-2, 0});
    }

    // Dijkstrův algoritmus z vrcholu 0 do 2*N-2 (sloupy 1 a N ve schodové kopii)
    vector<bool> nalezeno(2*N, false); // Jestli jsme už daný vrchol objevili
    nalezeno[0] = true;
    vector<int> vzdalenosti(2*N);
    vzdalenosti[0] = 0;
    // Vrcholy čekající na zpracování (v minimové haldě)
    // (nejprve vzdálenost do vrcholu, poté číslo do vrcholu)
    priority_queue<pii, vector<pii>, greater<pii>> halda;
    halda.push({0, 0});
    while(!halda.empty()){
        int vzdalenost = halda.top().first;
        int vrchol = halda.top().second;
        halda.pop();
        if(vzdalenosti[vrchol] < vzdalenost) continue;
        if(vrchol == 2*N-2) break;
        for(pii hrana: naslednici[vrchol]){
            int soused = hrana.first;
            int nova_vzdalenost = vzdalenost + hrana.second;
            if((!nalezeno[soused]) or vzdalenosti[soused] > nova_vzdalenost){
                nalezeno[soused] = true;
                vzdalenosti[soused] = nova_vzdalenost;
                halda.push({nova_vzdalenost, soused});
            }
        }
    }

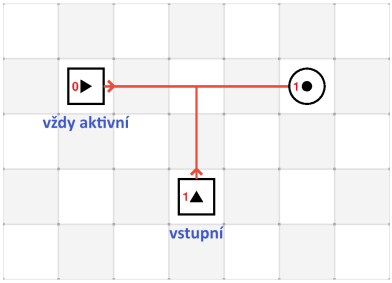
    cout << vzdalenosti[2*N-2] << endl;

    return 0;
}
```

P-I-4 Ben má nepořádek na stole

Podúloha a: Konstrukce hradla NOT

Označme indexem 0 emitor, který bude vždy aktivní, a indexem 1 vstupní emitor. Stačí nám docílit toho, aby emitor 0 zasáhl kolektor a aby ho emitor 1 vyrušil, bude-li aktivní. Jediné, na co je potřeba dát si pozor, je správné načasování srážky.

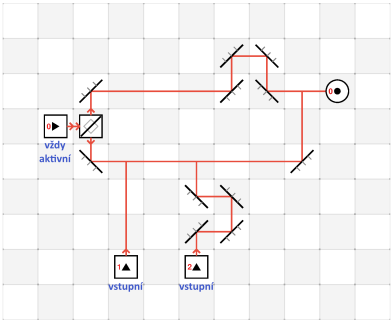


Konstrukce pro úlohu NOT

Podúloha b: Konstrukce hradla OR

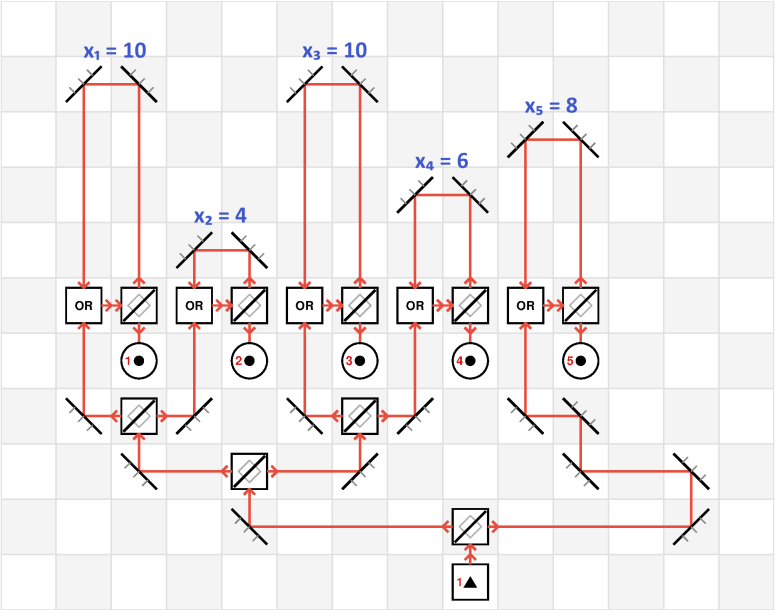
Při pokusu pouze mírně modifikovat konstrukci pro hradlo AND ze studijního textu nás bude limitovat počet vstupů kolektoru. Použijme vždy aktivní emitor 0 a vstupní emitory označme 1 a 2.

Rozdělme si laser z emitoru 0 na dva. Spodní větev zkusíme postupně vyrušit oběma vstupními emitory, aniž bychom vyrušili horní větev. Docílíme toho správným načasováním: spodní větev paprsku z emitoru 0 se střetne s paprsky ze vstupních emitorů v časech 3, respektive 5. Pokud ani jeden vstupní emitor není aktivní, spodní větev pokračuje dál a následně vyruší horní větev, takže do kolektoru nic nedorazí. Jinak horní větev do kolektoru doletí.



Konstrukce pro úlohu OR

Podúloha c: Rytmická blikání



Konstrukce pro úlohu Rytmická blikání s pěti čísly: $10, 4, 10, 6, 8$

Ukažme pro jedno x_i, jak zajistit, aby paprsek zasáhl daný kolektor každých x_i sekund. Vytvořme si jednoduchý obdélníkový cyklus o šířce 1 a výšce $\frac{x_i}{2} - 1$. Takže jeden oběh trvá x_i sekund. Takový cyklus dokážeme vytvořit, protože x_i je sudé a větší než 2, takže výška je kladné celé číslo.

Paprsek zavřený do cyklu dokážeme uchovat napořád, ale z cyklu musí každých x_i sekund paprsek vyletět. Toho docílíme tím, že do jednoho rohu cyklu umístíme rozdělovač. Jednu větev pošleme přímo do kolektoru a druhou necháme v cyklu. Obdobně vyřešíme začátek cyklu. Do druhého rohu umístíme OR, které pošle paprsek dál po cyklu bez ohledu na to, jestli už v cyklu byl, nebo přilétl zvenku. Bude se nám hodit mít tyto dva významné rohy vedle sebe.

Druhou částí problému je vyřešit, jak laser ze vstupu rozdělit, aby paprsky poprvé zasáhly všechny kolektory současně. Zvolme rozestup kolektorů jako 1 volné pole. O jedno pole diagonálně výše vlevo umístíme vstupní OR cyklu daného kolektoru. Takže i tyto ORy jsou v konfiguraci na přímce s rozestupy 1.

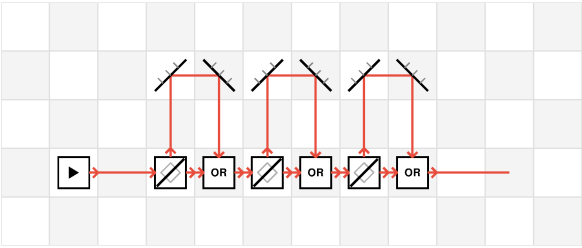
Konstruujme rozdělení zpětně, tedy paprsky spojujme. Díky lichým mezerám mezi paprsky můžeme spojení umístit pod středy dvojic tak, aby oba paprsky urazily stejnou vzdálenost. Vzdálenosti mezi nově spojenými paprsky se zdvojnásobí, takže mezery zůstanou liché. Postup opakujeme, dokud nezbude jediný paprsek, který připojíme k jistě aktivnímu emitoru.

V podstatě jsme postavili binární strom nad vstupními rohy do cyklů. Z jeho symetrie plyne, že všechny paprsky poletí stejně dlouho. Díky tomu musí existovat nějaké T splňující podmínku ze zadání.

Jediný detail nastane, pokud N není mocninou dvojky: občas nám zůstane paprsek bez protějšku, se kterým by se spojil. Pak musíme paprsek vést pomocí zrcadel stejně, jako kdybychom ho mohli s dalším paprskem spojit. Na obrázku je uveden případ, kdy $N = 5$ a $x_1, \ldots, x_5 = 10, 4, 10, 6, 8$.

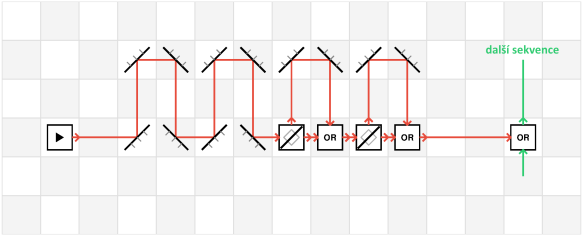
Podúloha d: Součet

První krok je vyřešit reprezentaci síly emitoru. Pro emitor se sílou p vytvoříme sekvenci p laserů letících za sebou, které budou mít vzdálenost 4 (vzdálenosti měříme mezi začátky laserů). Tuto konstantu odůvodníme později. Pro $p > 0$ použijeme $p - 1$ sériově zapojených duplikačních buněk. Tři takové buňky jsou zkonstruované na obrázku. Pokud už máme sekvenci délky x, pak průlet jednou duplikační buňkou nám tuto sekvenci prodlouží o 1. Každý paprsek se rozdělí, zpozdí o 4 a připojí zpět do sekvence. Paprsky se zpozdí přesně o vzdálenost mezi paprsky a díky OR splynou. Výjimkou je opožděná kopie posledního paprsku, která už nemá s čím splynout, a proto sekvenci prodlouží o 1. Celá konstrukce na obrázku vytvoří sekvenci délky 4. S touto konstrukcí nám aktivní emitor vrátí sekvenci délky p a neaktivní 0. Pro $p = 0$ případný paprsek pohltíme vhodně natočeným zrcadlem.



3 spojené duplikační buňky (pro $p = 4$)

Nyní musíme vymyslet, jak spojit vzniklé sekvence ze všech emitorů. Nejjednodušší bude rozmístit emitory na přímce s dostatečně velkým rozestupem dělitelným 4. Ke každému emitoru připojíme $P - 1$ duplikačních buněk, kde P je síla nejsilnějšího emitoru. U prvních $P - p_i$ buněk nahradíme rozdělovač a OR zrcadly. Pak budou mít všechny sekvence správnou délku a zároveň všechny poletí rovnoběžně. Navíc budou jejich poslední paprsky synchronizované, protože vždy projdou delší větví každé buňky. Díky tomu bude vzdálenost mezi posledním paprskem jedné sekvence a prvním paprskem té následující dělitelná 4. Sekvence spojíme pomocí OR na jedné přímce a díky libovolně velkým rozestupům emitorů můžeme zaručit, že při spojování v OR nedojde ke splynutí žádných dvou paprsků. Zapojení jednoho emitoru (a jeho sekvence) můžeme vidět na dalším obrázku.



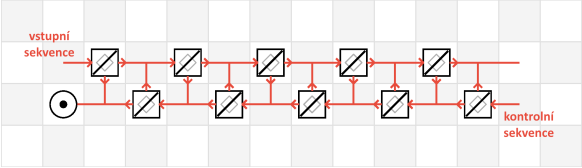
Zapojení emitoru s $p = 3$, $P = 5$

Pospojovaným sekvencím ze vstupních emitorů říkejme *vstupní sekvence*. Dále využijme jednoho vždy aktivního emitoru a vytvořme sekvenci délky K (stejným způsobem jako ze vstupních emitorů). Té budeme říkat *kontrolní sekvence*. Chceme ověřit, zda má kontrolní sekvence více paprsků než vstupní sekvence.

Obě sekvence necháme letět proti sobě po rovnoběžných drahách vzdálených jedno pole. Na našem obrázku níže přiletí kontrolní sekvence vpravo dole a vstupní sekvence vlevo nahoře.

Aby se protiletící paprsky vzájemně vyrušily, musíme správně načasovat jejich průchod rozdělovači. Označme R první rozdělovač v dolní řadě, kterým v čase t proletí první paprsek kontrolní sekvence. Dále označme S rozdělovač v horní řadě, kterým v témže čase t proletí první paprsek vstupní sekvence. Ke správnému načasování a následnému vyrušení pak dojde tehdy, leží-li mezi rozdělovači R a S v horní řadě sudý počet rozdělovačů. Tato podmínka je splněna například v případě, že by R a S byly krajní rozdělovače na obrázku. Díky rozestupům dělitelným 4 se pak i všechny následující paprsky potkají ve správný okamžik bez ohledu na to, které emitory byly na počátku aktivní.

Každé vyrušení odstraní jeden paprsek z obou sekvencí. Při dostatečné délce konstrukce tak v kontrolní sekvenci zbude paprsek právě tehdy, když K je větší než počet paprsků ve vstupní sekvenci. Případný zbývající paprsek zachytíme na výstupu.



Porovnání počtů paprsků