74. ročník Matematické olympiády – 2024/2025

Řešení úloh krajského kola kategorie P

P-II-1 Největší čtverec

Nejjednodušší způsob, jak úlohu vyřešit, je procházet všechny čtvercové sekce chodníku a pamatovat si doposud největší takovou, která obsahuje k nebo méně rozbitých dlaždic. Například pro každou dlaždici můžeme zkontrolovat všechny čtverce, jejichž levým horním rohem je tato dlaždice a které se vlezou na chodník. Jejich velikost může být 1 až $\mathcal{O}(\min(n, m))$, takže celkem se jedná o $\mathcal{O}(nm \min(n, m))$ čtverců.

Pokud budeme počet rozbitých dlaždic v daném čtverci počítat po jednom, dostaneme správné řešení s časovou složitostí $\mathcal{O}(nm[\min(n, m)]^3)$, bohužel příliš pomalé pro vstupy s $n, m \geq 50$.

Na problémy typu „počet/součet v obdélníku“ existuje mnoho rychlých řešení. Pro naše potřeby postačí dvourozměrné prefixové součty. Pro každou dlaždici se souřadnicemi (x, y) předpočítáme, kolik rozbitých dlaždic se nachází v obdélníku, jehož levý horní roh je samotný levý horní roh chodníku a pravý dolní roh je dlaždice (x, y). Pak lze spočítat počet rozbitých dlaždic v libovolném obdélníku v konstantním čase.[1]

Nejdříve musíme prefixové součty spočítat, což trvá úměrně počtu zahrnutých dlaždic: $\mathcal{O}(nm)$. Díky těmto prefixovým součtům jsme následně schopni zpracovat každý čtverec v konstantním čase, což nám dává složitost úměrnou počtu čtvercových sekcí: $\mathcal{O}(nm \min(n, m))$. Celková časová složitost je tedy $\mathcal{O}(nm + nm \min(n, m)) = = \mathcal{O}(nm \min(n, m))$.

Lineární řešení

Čtvercovou sekci chodníku nazveme použitelnou, pokud obsahuje k nebo méně rozbitých dlaždic, a je tedy možným řešením.

Pro rychlejší řešení použijeme následující princip: Pokud najdeme použitelný čtverec velikosti a, nemá už cenu zkoušet čtverce velikosti a nebo menší, protože nemohou naše řešení dále vylepšit.

Nechť a je velikost strany největšího doposud nalezeného použitelného čtverce, na začátku 0. Opět budeme zkoušet všechny dlaždice chodníku a hledat co největší použitelnou čtvercovou sekci s levým horním rohem na dané dlaždici a velikostí strany alespoň $a + 1$: Zjistíme, kolik rozbitých dlaždic obsahuje. Pokud není použitelná, nemohou být použitelné ani větší, které ji zahrnují, a můžeme se posunout na další dlaždici. Pokud naopak je použitelná, můžeme a postupně zvyšovat o jedna a zkoumat sekce se stejným levým horním rohem, dokud nenarazíme na nějakou nepoužitelnou nebo na konec chodníku.

---

[1] Přesné vysvětlení (i s obrázky) najdete na *https://ksp.mff.cuni.cz/kucharky/zakladni-algoritmy/* v sekci *Prefixové součty.*

Všimněme si, že po každém výpočtu rozbitých dlaždic ve čtvercové sekci se buď přesuneme k další dlaždici, anebo zvětšíme a o jedna. Možných levých horních rohů je nm a a může být maximálně $\min(n, m)$, takže tento výpočet proběhne až $(nm + \min(n, m))$-krát.

V kombinaci s dvourozměrnými prefixovými součty, jejichž předpočítání zabere $\mathcal{O}(nm)$ času a dotazy lze zodpovědět v konstantním čase $\mathcal{O}(1)$, tak lze dosáhnout časové složitosti $\mathcal{O}(nm + nm + \min(n, m)) = \mathcal{O}(nm)$.

Program (C++):

```
#include <bits/stdc++.h>
using namespace std;

// ps[r][s] ... počet rozbitých dlaždic ve čtverci od [0, 0] do [r - 1, s - 1]
int ps[1001][1001];
int n, m, k;

int count_broken(int r, int s, int a) {
    // Pokud čtverec zasahuje mimo okraje, vrátíme k+1.
    // Tím zajistíme, že se nepoužije.
    if (r + a > n || s + a > m) return k + 1;
    return ps[r + a][s + a] - ps[r + a][s] - ps[r][s + a] + ps[r][s];
}

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);

    cin >> n >> m >> k;
    vector<string> pavement(n);
    for (int i = 0; i < n; ++i) {
        cin >> pavement[i];
    }

    // inicializace prefixových součtů
    for (int r = 0; r < n; ++r) {
        for (int s = 0; s < m; ++s) {
            ps[r + 1][s + 1] = (pavement[r][s] == '@') + ps[r][s + 1]
                             + ps[r + 1][s] - ps[r][s];
        }
    }

    int best_a = 0;
    int best_r = 0;
    int best_s = 0;
    for (int r = 0; r < n; ++r) {
        for (int s = 0; s < m; ++s) {
            while (count_broken(r, s, best_a + 1) <= k) {
                best_a += 1;
                best_r = r;
                best_s = s;
            }
        }
    }
    cout << best_r + 1 << ' ' << best_s + 1 << ' ' << best_a << '\n';
}
```

P-II-2 Vynášení odpadků

Jelikož robot může nést jen jeden odpadek, bude střídavě chodit mezi popelnicemi a odpadky. Robotova cesta je tedy určena pořadím, ve kterém vyhazuje odpadky. Pro dané pořadí existuje jediná nejkratší cesta, a to chodit přímou cestou. Samotná délka robotovy cesty se ale dá popsat něčím jednodušším, než je pořadí odpadků.

Každý odpadek přispěje dvěma úseky cesty – cesta od popelnice k odpadku a cesta od odpadku k popelnici. Druhá cesta je vždy stejně dlouhá bez ohledu na pořadí vynášení odpadků. U první cesty jsou dvě možnosti jak může vypadat – buď jdeme od papírové, nebo od plastové popelnice. Tyto popelnice pro daný odpadek nazveme startovní a cílové. Důležitou roli v řešení budou hrát odpadky, které mají jinou startovní a cílovou popelnici. Takové odpadky nazveme přechodné. Délka cesty záleží jen na tom, které odpadky jsou přechodné.

Mohli bychom sestavit řešení jen na základě volby přechodných odpadků? Za určitých podmínek ano. Všimněme si, že pokud neexistuje žádný plastový odpadek, tak nemůže existovat přechodný odpadek a délka cesty nejde změnit. Pokud ale máme aspoň jeden plastový odpadek, jeden plastový odpadek bude muset být přechodný. Pokud máme aspoň jeden přechodný odpadek, budou se nám střídat úseky, kdy chodíme k papírové a kdy k plastové popelnici. Nepřechodné odpadky můžeme vyhodit kdykoli, kdy chodíme k dané popelnici. Dále musí platit, že přechodných plastů je buď stejně nebo právě o jedna více, než přechodných papírů. Pokud splníme tyto požadavky, určitě existuje cesta na vynesení všech odpadků, která má právě zvolené přechodné odpadky.

Jak najít nejkratší cestu? Určitě chceme za přechodné odpadky zvolit ty, pro které je to nejvýhodnější. Pro každý odpadek si tedy spočítáme, jakou délkou přispěje, pokud ho navštívíme jako přechodný a jako nepřechodný odpadek (označíme po řadě p_i a q_i). Rozdíl těchto dvou hodnot bude určovat, jak moc nám „pomůže“, když daný odpadek vezmeme jako přechodný. Označíme $d_i = p_i - q_i$, toto číslo určuje o kolik se změní délka cesty, pokud daný odpadek vezmeme jako přechodný místo nepřechodného. Je zřejmé, že pro daný počet přechodných odpadků je nejlepší vybírat ty, které nám co nejvíce pomůžou – tedy mají d_i co nejmenší.

Podle d_i si tedy seřadíme všechny papíry a všechny plasty zvlášť. Začneme s cestou neobsahující žádné přechodné odpadky. Poté budeme postupně přičítat d_i odpadků od nejmenších po největší, střídavě plasty a papíry. Řešením bude nejmenší délka cesty, kterou takto dostaneme. Tento algoritmus má časovou složitost $\mathcal{O}((m+n)\log(m+n))$ – kvůli třídění, následný průchod je lineární.

Poznámka

Existuje i řešení s časovou složitostí $\mathcal{O}(m+n)$. Pokud bychom přechodné odpadky přidávali po dvou, délka postupně získávané délky cesty budou neklesající – rozdíl mezi dvěma po sobě jdoucími délkami je $d_i + d_{m+i}$, kde d_i patří papírovému odpadku s i-tou nejmenší hodnotou d a d_{m+i} patří plastovému odpadku s i-tou nejmenší hodnotou d. Rozdíl následujících dvou délek je $d_{i+1} + d_{m+i+1}$, a to určitě není menší.

Určitě tedy budeme chtít přechodné odpadky přidávat, dokud je hodnota $d_i + d_{m+i}$ kladná. Na takovýto úkol můžeme nasadit binární vyhledávání. Jediným problémem je, že nevíme, jak správně půlit interval, jelikož odpadky nemáme seřazené podle d_i. Naštěstí ale existuje algoritmus *Quickselect*, který umí vyhledávat k-tý nejmenší prvek v nesetříděném poli v lineárním čase. Pomocí tohoto algoritmu můžeme najít požadované d_i a d_{m+i} a rozdělit odpadky pro binární vyhledávání. Je-li $n' = \min(m, n)$, pak čas strávený binárním vyhledáváním je $n' + n'/2 + n'/4 + \ldots \in \mathcal{O}(n') \subseteq \mathcal{O}(m + n)$.

Zbývá dodat, že přechodných odpadků může být i lichý počet. Toto ošetříme tím, že binární vyhledávání spustíme dvakrát a podruhé odstraníme plast s nejnižší hodnotou d_i a budeme ho rovnou počítat jako přechodný. Z dvou délek, co nám vyjdou, vybereme tu menší.

Program (C++):

```
#include <bits/stdc++.h>

using namespace std;

double dist(int x1, int y1, int x2, int y2) {
    return sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2));
}

int main() {
    int M,N,x1,y1,x2,y2;
    cin >> M >> N >> x1 >> y1 >> x2 >> y2;

    vector<pair<int, int>> paper(M), plastic(N);
    for (int i = 0; i < M; i++) {
        cin >> paper[i].first >> paper[i].second;
    }
    for (int i = 0; i < M; i++) {
        cin >> plastic[i].first >> plastic[i].second;
    }

    double total = 0;
    vector<double> paperDiff(M), plasticDiff(N);
    for (int i = 0; i < M; i++) {
        double d1 = dist(x1, y1, paper[i].first, paper[i].second);
        double d2 = dist(x2, y2, paper[i].first, paper[i].second);
        total += 2 * d1;
        paperDiff[i] = d2 - d1;
    }
    for (int i = 0; i < M; i++) {
        double d1 = dist(x1, y1, plastic[i].first, plastic[i].second);
        double d2 = dist(x2, y2, plastic[i].first, plastic[i].second);
        total += 2 * d2;
        plasticDiff[i] = d1 - d2;
    }

    sort(paperDiff.begin(), paperDiff.end());
    sort(plasticDiff.begin(), plasticDiff.end());

    double best = numeric_limits<double>::infinity();
    if (N == 0)
        best = total;
```

```
    for (int i = 0; i < N; i++) {
        total += plasticDiff[i];
        best = min(best, total);

        if (i >= M)
            break;

        total += paperDiff[i];
        best = min(best, total);
    }

    cout << fixed << setprecision(10) << best << endl;
}
```

P-II-3 Venčení

Máme zadaný strom a chceme najít takový vrchol, po jehož odstranění bude nejdelší cesta co nejkratší. Přímočarý postup je zkusit odstranit každý vrchol, spočítat pro něj nejdelší cestu ve výsledném grafu, a nakonec vybrat ten nejlepší.

Jak ale tuto délku určit? Triviálním řešením by bylo z každého vrcholu spustit prohledávání do šířky, a tím změřit vzdálenosti mezi všemi dvojicemi vrcholů. Takto bychom našli největší vzdálenost v čase $\mathcal{O}(N^2)$. Jelikož ji hledáme pro každý zkoušený vrchol, celková časová složitost bude $\mathcal{O}(N^3)$, což nám stačí na 2 body.

Rychlejší algoritmus

Když zkoušíme odebrat nějaký vrchol ze stromu, graf se nám rozpadne na několik podstromů – délku nejdelší cesty můžeme počítat pro každý zvlášť. V kontextu stromů se této délce přezdívá *průměr* a lze ho určit i rychleji, dokonce v lineárním čase. Způsobů je více, ten nejelegantnější si dokonce vystačí jen se dvěma prohledáními do šířky[1]. My si ukážeme přístup z pohledu dynamického programování (DP), jelikož se nám bude hodit i pro plné řešení.

Strom si libovolně zakořeníme a jeho průměr budeme chtít spočítat z průměrů jeho podstromů (a ty zase z průměrů jejich podstromů…). Můžeme k tomu použít prohledávání do hloubky – to nám zaručí, že kdykoliv odcházíme z nějakého vrcholu, určitě jsme již navštívili všechny jeho děti. Počítat budeme pro každý podstrom dvě informace: maximální hloubku a samotný průměr. Pro listy je obojí triviálně číslo 1. Pro ostatní vrcholy spočítáme pomocí informací od dětí:

- maximální hloubka bude o jedna vyšší než ta největší od dětí
- pro průměr můžeme uvažovat dva možné případy a zvolit ten nejvyšší:
  - cesta nevede skrz rodiče – použijeme nejvyšší průměr od dětí
  - cesta vede skrz rodiče – pak ta nejdelší povede z/do dvou nejhlubších větví (ty známe z maximálních hloubek)

Tímto způsobem postupně spočítáme průměry všech podstromů, až nakonec celého stromu. Stačí nám na to jedno prohledání do hloubky, které zabere vůči velikosti stromu lineárně mnoho času i prostoru. Jelikož tento algoritmus pouštíme dohromady jednou na celém grafu pro každý zkoušený vrchol, celková časová složitost bude $\mathcal{O}(N^2)$, za což dostaneme 5 bodů.

Optimalizace zkoušených vrcholů

Užitečné pozorování je, že nějaká správná odpověď jistě musí být na nejdelší cestě stromu ze zadání. Odstraníme-li totiž vrchol mimo takovou cestu, nejdelší cesta se nijak nezkrátí. Postačí tedy vyzkoušet jen vrcholy na nějaké nejdelší cestě. Najít ji můžeme třeba tak, že si v našem DP budeme udržovat i jaké listy průměr vymezují.

Nejdelší cesta může být v nejhorším případě velká jako celý strom. Zamysleme se však nad dodatečným předpokladem podúlohy za 7 bodů: pokud by měl zadaný

[1] Přečíst si o něm můžete třeba tady pod nadpisem „Průměr stromu podruhé“: *https://ksp.mff.cuni.cz/viz/29-1-7/reseni*

strom průměr větší než 199, tak i kdybychom jeho nejdelší cestu rozpůlili, jistě bude existovat cesta o délce alespoň 100; tudíž by byl předpoklad porušen. Odtud plyne, že na všech vstupech této podúlohy jsou průměry nízké a stihneme vyzkoušet odstranit všechny vrcholy na cestě.

Odpovědi podstromů

Abychom vyřešili celou úlohu, využijeme toho, že v rámci našeho dynamického programování nepočítáme průměr jen celého zakořeněného stromu, ale i všech jeho podstromů. Když zkoušíme odstranit nějaký vrchol, graf se nám rozpadá na různé podstromy, jejichž průměry si můžeme předpočítat. Konkrétně pokud budeme zkoušet pouze vrcholy na nějaké nejdelší cestě, postačí nám hodnoty DP spočítat dvakrát: jednou zakořeníme na začátku této cesty, jednou na jejím konci. Takto budeme znát průměry jak podstromů „vycházejících“ z nejdelší cesty, tak podstromů „na cestě“.

Shrňme si tedy finální řešení. Nejdříve v zadaném stromu najdeme nějakou nejdelší cestu. Strom pak zakořeníme na začátku této cesty a všem podstromům spočítáme průměr. Poté strom zakořeníme i na konci cesty a připíšeme si průměry těchto podstromů. Následně projdeme celou cestu a budeme zkoušet odstranit její vrcholy: podíváme se na průměry v rozpadeném stromě. Nakonec zvolíme takový vrchol, jenž minimalizuje největší průměr.

DP použijeme třikrát a spotřebujeme na to lineární množství času i prostoru. Procházet budeme sousedy všech vrcholů na nejdelší cestě; některé mohou mít mnoho sousedů, jiné méně, dohromady však řádově nejvýš lineárně mnoho. Celková časová i prostorová složitost je tedy $\mathcal{O}(N)$.

Program (C++):

```
#include <iostream>
#include <vector>
#include <utility>
#include <cassert>

using namespace std;

struct OrientovanaHrana {
    int soused;

    // DP hodnoty pro příslušný podstrom
    int prumer = -1;
    int max_hloubka = -1;
    int nejhlubsi_vrchol = -1;
    pair<int,int> hranice_prumeru = {-1, -1};

    OrientovanaHrana(int _soused): soused(_soused) {}
};

vector<vector<OrientovanaHrana>> graf; // ze vstupu

// vrací orientovanou hranu na sebe
OrientovanaHrana dp_dfs(int vrchol, int rodic = -1) {
    OrientovanaHrana vysledek(vrchol);
    vysledek.prumer = 1;
    vysledek.max_hloubka = 0;
```

```
    vysledek.nejhlubsi_vrchol = vrchol;
    vysledek.hranice_prumeru = {vrchol, vrchol};

    int druha_max_hloubka = -1;
    int druhy_nejhlubsi_vrchol = -1;

    for (OrientovanaHrana& hrana : graf[vrchol]) {
        int soused = hrana.soused;
        if (soused == rodic) continue;
        hrana = dp_dfs(soused, vrchol);

        // aktualizace dvou nejhlubších vrcholů
        if (hrana.max_hloubka >= vysledek.max_hloubka) {
            druha_max_hloubka = vysledek.max_hloubka;
            druhy_nejhlubsi_vrchol = vysledek.nejhlubsi_vrchol;
            vysledek.max_hloubka = hrana.max_hloubka;
            vysledek.nejhlubsi_vrchol = hrana.nejhlubsi_vrchol;
        } else if (hrana.max_hloubka > druha_max_hloubka) {
            druha_max_hloubka = hrana.max_hloubka;
            druhy_nejhlubsi_vrchol = hrana.nejhlubsi_vrchol;
        }

        // uvažujme cestu neprocházející vrcholem
        if (hrana.prumer > vysledek.prumer) {
            vysledek.prumer = hrana.prumer;
            vysledek.hranice_prumeru = hrana.hranice_prumeru;
        }
    }

    // uvažujme cestu procházející vrcholem
    int delka = vysledek.max_hloubka + 1 + druha_max_hloubka;
    if (delka > vysledek.prumer) {
        vysledek.prumer = delka;
        vysledek.hranice_prumeru = {vysledek.nejhlubsi_vrchol,
                                    druhy_nejhlubsi_vrchol};
    }

    vysledek.max_hloubka++; // započítáme sebe

    return vysledek;
}

// určí rodiče pro nějaké zakořenění
void rodice_dfs(vector<int>& rodice, int vrchol, int rodic = -1) {
    rodice[vrchol] = rodic;
    for (OrientovanaHrana& hrana : graf[vrchol]) {
        if (hrana.soused == rodic) continue;
        rodice_dfs(rodice, hrana.soused, vrchol);
    }
}

vector<int> cesta_z_hranic(pair<int,int> hranice) {
    vector<int> cesta;

    // zakořeníme na konci cesty
    vector<int> rodice(graf.size());
    rodice_dfs(rodice, hranice.second);

    // následujeme rodiče od začátku cesty
    int vrchol = hranice.first;
```

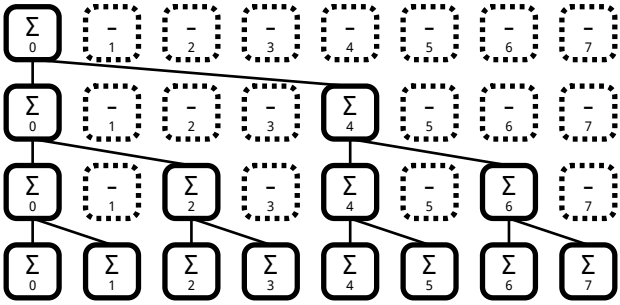
```
    while (vrchol != -1) {
        cesta.push_back(vrchol);
        vrchol = rodice[vrchol];
    }

    return cesta;
}

int reseni() {
    int nejlepsi_prumer = graf.size()+1; // nekonečno
    int odpoved = -1;

    // zakořeníme v libovolném vrcholu, abychom našli nejdelší cestu
    pair<int,int> hranice = dp_dfs(0).hranice_prumeru;
    vector<int> nejdelsi_cesta = cesta_z_hranic(hranice);

    // spočítáme DP se zakořeněním v začátku a konci
    dp_dfs(hranice.first);
    dp_dfs(hranice.second);

    // zkusíme odstranit všechny vrcholy na nejdelší cestě
    for (int vrchol : nejdelsi_cesta) {
        int nejvyssi_prumer = 0;
        for (OrientovanaHrana& hrana : graf[vrchol]) {
            assert(hrana.prumer != -1); // DP by již mělo být vypočítané
            nejvyssi_prumer = max(nejvyssi_prumer, hrana.prumer);
        }

        if (nejvyssi_prumer < nejlepsi_prumer) {
            nejlepsi_prumer = nejvyssi_prumer;
            odpoved = vrchol;
        }
    }

    return odpoved;
}

int main() {
    int n;
    cin >> n;
    graf.resize(n);

    for (int i = 0; i < n-1; i++) {
        int u, v;
        cin >> u >> v;
        u--; v--; // indexujme od nuly
        graf[u].push_back(OrientovanaHrana(v));
        graf[v].push_back(OrientovanaHrana(u));
    }

    // indexujme zase od jedničky
    cout << reseni()+1 << endl;
}
```

P-II-4 Počítáme s tříděním

Průměrnost

Průměr úzce souvisí se součtem, takže nejprve spočítáme součet všech vstupů. To jsme se naštěstí už naučili v domácím kole. Pokud si řešení úlohy Součet z domácího kola nepamatujete, pak vám doporučujeme si ho přečíst znovu. Tentokrát ale potřebujeme, aby se součet dozvěděla všechna jádra, nejen to s identifikačním číslem 0.

Hodnotu v jádru 0 tedy musíme do ostatních jader nějak zkopírovat. To uděláme přesně opačně oproti tomu, jak jsme hodnoty sčítali. Nejprve odsuneme na konec posloupnosti všechna jádra kromě těch s identifikačními čísly 0 a $N/2$. Jádro s číslem $N/2$ si pak součet zkopíruje od jádra s číslem 0. Poté mezi ně přidáme jádra s čísly $N/4$ a $3N/4$, která si zkopírují součet od jader nalevo od nich, a tak dále. Výpočet znázorňuje tento binární strom:



Strom kopírování.

Poté, co se všechna jádra dozví součet Σ, může každé snadno ověřit, jestli je jeho vstup podprůměrný, průměrný nebo nadprůměrný. Spočítat průměr přímo nemůžeme, protože nemusí být celé číslo. To naštěstí není potřeba, protože např. místo nerovnice $a < \Sigma/N$ můžeme vyhodnotit celočíselnou nerovnici $aN < \Sigma$.

Toto řešení má složitost $\mathcal{O}(\log N)$, stejně jako sčítání.

Program:

```
# Zadefinujeme pomocné konstanty
dvě:        const 2
jádra:      const N

## Sčítáme

# Nastavíme „proměnné" na jejich počáteční hodnotu
id:         id
mezisoučet: input 1
```

Pro všechna i od 0 do $\log_2 N - 1$:

```
    # K mezivýsledku přičteme mezivýsledek jádra napravo
    napravo:    right mezisoučet
    mezisoučet: + mezisoučet napravo
```

```
    # Jádra s lichými ID přesuneme na konec
    zbytek:     % id dvě
                sort zbytek

    # Pomyslné ID pro další operace vydělíme dvěma
    id:         / id dvě

součet:     copy mezisoučet

## Zkopírujeme součet

# Obnovíme původní hodnotu id
id:         id
```

Pro všechna i od $\log_2 N - 1$ do 0 (pozpátku):

```
    # Spočítáme podíl a zbytek po dělení ID číslem 2^i
    dělitel:    const 2^i
    podíl:      / id dělitel
    zbytek:     % id dělitel

    # Seřadíme jádra dle ID, poté ta nedělitelná 2^i přesuneme na konec
                sort id
                sort zbytek

    # Nová jádra si zkopírují součet od jejich levého souseda
    je_nové:    % podíl dvě
    nalevo:     left součet
    součet:     if je_nové nalevo součet

# Porovnáme N * vstup se součtem
vstup:      input 1
součin:     * vstup jádra
menší:      < součin součet
větší:      > součin součet
            - větší menší
```

Alternativní řešení

Při opravování řešení jsme zjistili, že někteří z vás vymysleli jednodušší způsob, jak spočítat součet všech vstupů a uložit ho do všech jader, a to pomocí jen jedné smyčky.

Jádra opět rozdělíme na sudá a lichá. Sousední sudá a lichá jádra si přičtou své mezisoučty navzájem, tedy sudé jádro si přičte součet jádra napravo, zatímco liché jádro si přičte (původní) součet jádra nalevo. Tím jsme dosáhli toho, že součet mezisoučtů všech lichých jader je rovný součtu všech vstupů, totéž platí o součtu mezisoučtů všech sudých jader. Problém jsme tedy rozdělili na dva podproblémy poloviční velikosti.

Jádra seřadíme podle jejich parity a jejich ID pomyslně vydělíme dvěma. Můžeme si všimnout, že pokud provedeme výše uvedenou proceduru znovu, tak bude na každou z polovin působit nezávisle, aniž by ji ovlivňovala ta druhá. Poloviční podproblémy se nám tedy rozdělí na čtvrtinové podproblémy. Budeme pokračovat $\log_2 N$-krát, dokud nezískáme podproblémy velikosti 1. Po každém kroku algoritmu platí, že součet mezisoučtů v každém podproblému je rovný součtu všech původních vstupů. Na konci proto bude mezisoučet každého jádra rovný správnému výsledku.

Program:

```
# Zadefinujeme proměnné a konstanty
id:         id
součet:     input 1
jádra:      const N
dvě:        const 2

# Sčítáme
```

Zopakujeme $\log_2 N$-krát:

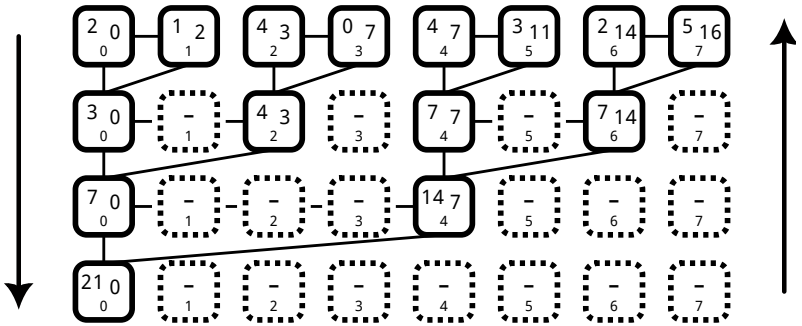
```
     liché:      % id dvě
     vlevo:      left součet
     vpravo:     right součet
     sčítanec:   if liché vlevo vpravo
     součet:     + součet sčítanec
                 sort liché
     id:         / id dvě

# Porovnáme N * vstup se součtem
vstup:      input 1
součin:     * vstup jádra
menší:      < součin součet
větší:      > součin součet
            - větší menší
```

Prefixový součet

Podobně jako Součet v domácím kole tuto úlohu vyřešíme metodou rozděl a panuj. Nejprve si ji ale trochu upravíme: Budeme počítat jen součet vstupů všech jader nalevo od daného jádra. Vstup jádra samotného k této hodnotě snadno přičteme na konci. Budeme se opět snažit najít způsob, jak N zredukovat na polovinu.

Podobně jako při sčítání spárujeme sousední sudá a lichá jádra. (Připomínáme, že první jádro má ID 0, a je tedy sudé.) Sudá jádra si ke svému vstupu na chvíli přičtou vstup lichého jádra, lichá jádra odsuneme na konec posloupnosti a úlohu rekurzivně vyřešíme pro $N \leftarrow N/2$. Poté lichá jádra vrátíme zpátky mezi sudá. Všimneme si, že sudá jádra v tu chvíli už mají prefixový součet spočítaný správně. Lichá jádra si mohou svůj prefixový součet snadno dopočítat tak, že k prefixovému součtu sudého jádra přičtou jeho (původní) vstup.

Tím jsme získali řešení se složitostí $\mathcal{O}(\log N)$. Průběh výpočtu můžeme opět nakreslit jako binární strom, který tentokrát projdeme dvakrát, jednou nahoru a jednou dolů:



Strom výpočtu. Jádra mají vstup vlevo, prefixový součet vpravo.

Program:

```
# Zadefinujeme pomocné konstanty
dvě:            const 2

## Opět sčítáme

# Nastavíme „proměnné" na jejich počáteční hodnotu
id:             id
mezisoučet_0:   input 1
```

Pro všechna i od 0 do $\log_2 N - 1$:

```
    # K mezivýsledku přičteme mezivýsledek jádra napravo
    napravo:        right mezisoučet_i
    mezisoučet_i+1: + mezisoučet_i napravo

    # Jádra s lichými ID přesuneme na konec
    zbytek:         % id dvě
                    sort zbytek

    # Pomyslné ID pro další operace vydělíme dvěma
    id:             / id dvě

## Spočítáme samotné prefixové součty

# Obnovíme původní hodnotu id
id:             id
# Prefixový součet prvního jádra je 0
prefix:         const 0
```

Pro všechna i od $\log_2 N - 1$ do 0 (pozpátku):

```
    # Spočítáme podíl a zbytek po dělení ID číslem 2^i
    dělitel:        const 2^i
    podíl:          / id dělitel
    zbytek:         % id dělitel

    # Seřadíme jádra dle ID, poté ta nedělitelná 2^i přesuneme na konec
                    sort id
                    sort zbytek

    # Nová jádra si spočítají prefixový součet
    je_nové:        % podíl dvě
    prefix_nalevo:  left prefix
```

```
součet_nalevo:  left mezisoučet_i
nový_prefix:    + prefix_nalevo součet_nalevo
prefix:         if je_nové nový_prefix prefix

# Nakonec přičteme vstup pro získání řešení
vstup:          input 1
                + prefix vstup
```

Alternativní řešení

I u této podúlohy jsme při opravování narazili na zajímavé, jednodušší alternativní řešení.

Náš algoritmus provede $\log_2 N$ fází. V i-té fázi (číslované od nuly) si k sobě každé jádro přičte mezisoučet jádra s o 2^i menším ID, pokud takové jádro existuje. Tím docílíme toho, že po provedení i fází bude každé jádro mít v sobě uložen součet sebe a $2^i - 1$ předchozích jader, nebo méně, pokud tolik jader s nižším ID neexistuje. Po $\log_2 N$ fázích tedy bude každé jádro znát součet až $2^{\log_2 N} = N$ jader, tedy celý prefixový součet.

Zbývá rozmyslet, jak implementovat jednu fázi. Potřebujeme zařídit, aby každé jádro sousedilo s jádrem s o 2^i menším ID. Toho dosáhneme tak, že jádra nejprve seřadíme podle jejich ID a pak podle zbytku dělení jejich ID konstantou 2^i. Nalevo od každého jádra pak bude nejbližší jádro, jehož ID má stejný zbytek po dělení 2^i, což bude přesně to jádro, které hledáme. Každé jádro si tedy ke svému mezisoučtu přičte mezisoučet jádra nalevo od něj, ale pouze pokud jádro nalevo od něj opravdu má o 2^i menší ID, protože takové vůbec jádro existovat nemusí. To můžeme snadno zkontrolovat přímo, ale v řešeních jsme narazili i na jednodušší, ekvivalentní podmínky. Můžeme například ověřit, jestli má jádro nalevo menší ID, nebo jestli je ID daného jádra alespoň 2^i.

Program:

```
# Zadefinujeme proměnné a konstanty
id:             id
součet:         input 1
```

Pro všechna i od 0 do $\log_2 N - 1$:

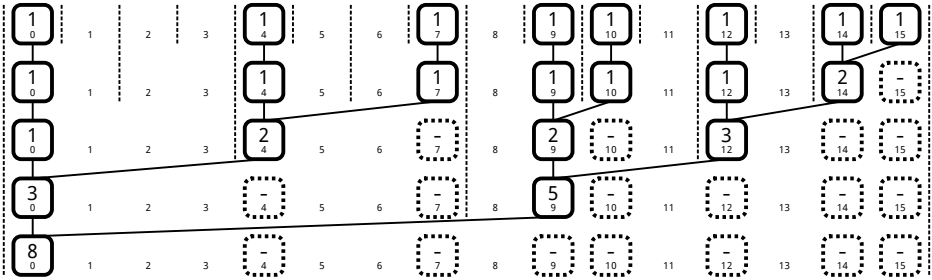
```
    # Seřadíme jádra
    dělitel:        const 2^i
    zbytek:         % id dělitel
                    sort id
                    sort zbytek

    # Spočítáme součet mezisoučtů jádra a jádra nalevo
    součet_vlevo:   left součet
    nový_součet:    + součet součet_vlevo

    # Součet použijeme, pokud existuje jádro s o 2^i menším ID
    má_přičítat:    >= id dělitel
    součet:         if má_přičítat nový_součet součet
```

Pořadí

Stejně jako v předchozích dvou úlohách opět použijeme metodu rozděl a panuj. Tentokrát ale budeme místo N redukovat na polovinu K, jak už napovídá žádaná složitost $\mathcal{O}(\log K)$.

Jádra seřadíme podle jejich vstupu. Tento vstup budeme chtít vydělit dvěma, abychom problém zmenšili. Pravděpodobně se ale stane, že dvě jádra skončí se stejným vstupem. To se stane každé dvojici sousedních jader, která mají vstupy ve tvaru $2i$ a $2i + 1$, pro nějaké i. Každou takovou dvojici jader proto sloučíme do jednoho jádra. Každé takové levé jádro si zapamatuje, že ve skutečnosti zastupuje dvě jádra (nebo více), a pravé jádro odsuneme na konec posloupnosti. Jádra, která do takové dvojice nepatří, necháme být.

Tím získáme problém s poloviční hodnotou K, který rekurzivně vyřešíme. Tuto půlku algoritmu můžeme opět znázornit stromem:



Strom výpočtu.
Jádra mají dole (původní) vstup, nahoře počet jader, která zastupují.

Můžeme si všimnout, že se tento algoritmus chová podobně jako algoritmus pro prefixový součet, jen některá jádra chybí. Tato jádra někdy nepotřebujeme, protože by si stejně pamatovala jen samé nuly, jindy je musí zastoupit nejbližší jádro napravo od nich. Druhá půlka algoritmu bude také téměř stejná jako u prefixového součtu. Odsunutá jádra budeme postupně vracet, každé vrácené jádro si spočítá pořadí jako součet pořadí jádra nalevo od něj a počtu jader, které toto jádro zastupuje.

Program:

```
# Zadefinujeme pomocné konstanty
jedna:              const 1
dvě:                const 2

## Jdeme po stromu směrem ke kořeni

# Nastavíme „proměnné" na jejich počáteční hodnotu
vstup:              input 1
je_vyřazené_0:      const 0
násobnost_0:        const 1
```

Pro všechna i od 0 do $\lceil \log_2(K+1) \rceil - 1$:

```
    # Seřadíme jádra dle vstupu, vyřazená dáme na konec
                        sort vstup
                        sort je_vyřazené_i

    # Zjistíme, jestli je jádro levé, respektive pravé ve dvojici
    je_liché:           % vstup dvě
    vstup_vlevo:        left vstup
    očekávaný_vstup:    - vstup jedna
    má_vlevo_očekávaný: = vstup_vlevo očekávaný_vstup
    je_pravé_v_dvojici: & je_liché má_vlevo_očekávaný
    je_levé_v_dvojici:  right je_pravé_v_dvojici

    # K násobnosti levých jader přičteme násobnost pravých
    násobnost_vpravo:   right násobnost_i
    součet_násobností:  + násobnost_i násobnost_vpravo
    násobnost_i+1:      if je_levé_v_dvojici součet_násobností násobnost_i

    # Pravá jádra vyřadíme a zmenšíme vstup
    je_vyřazené_i+1:    | je_vyřazené_i je_pravé_v_dvojici
    vstup:              / vstup dvě
```

```
## Jdeme po stromu zpátky od kořene

# Obnovíme vstup
vstup:              input 1
# Pořadí prvního jádra je 0
pořadí:             const 0
```

Pro všechna i od $\lceil \log_2(K+1) \rceil - 1$ do 0 (pozpátku):

```
    # Seřadíme jádra dle vstupu, vyřazená dáme na konec
                        sort vstup
                        sort je_vyřazené_i

    # Nově přidaná jádra si spočítají pořadí
    je_nové:            = je_vyřazené_i+1
    pořadí_nalevo:      left pořadí
    násobnost_nalevo:   left násobnost_i
    nové_pořadí:        + pořadí_nalevo násobnost_nalevo
    pořadí:             if je_nové nové_pořadí pořadí
```