P-III-4 Barevný graf

V našem řešení budeme postupně procházet jednotlivé operace a stavět podle nich vhodný graf. V něm následně nalezneme nejkratší cestu mezi vrcholy u a v.

Stavba grafu

Nabízí se postupně projít jednotlivé operace a nějak na základě nich postavit totožný graf, jaký namaloval Jirka. Takový postup by byl nicméně, nehledě na implementaci, pro naše účely příliš pomalý. Uvažme, že i s ignorováním násobných hran jich mohl Jirka namalovat celkem $\mathcal{O}(n^2)$.

Proto budeme místo Jirkova grafu postupně stavět jiný graf, takový, který zachová délku hledané nejkratší cesty a navíc bude podporovat operace barvení a spojování v konstantním čase. Tento graf bude orientovaný a všechny hrany v něm získají délku buď 0 nebo 1. Dále bude obsahovat všechny vrcholy původního Jirkova grafu, které budou mít přiřazená stejná čísla a stejné barvy. Nově též bude mít každá barva v grafu svůj vlastní vstupní a výstupní vrchol. Pokud budeme chtít propojit vrcholy vybraných dvou barev, nikdy je nebudeme propojovat přímo, ale vždy s využitím vstupních a výstupních vrcholů příslušných barev. Protože budou některé barvy v průběhu stavby grafu své vstupní a výstupní vrcholy měnit, budeme si pro každou barvu udržovat v poli odkazy na její aktuální vstupní a výstupní vrchol.

Před začátkem stavby grafu povede ze vstupního vrcholu barvy i orientovaná hrana délky 0 do vrcholu i. Naopak do výstupního vrcholu barvy i povede orientovaná hrana délky 0 z vrcholu i.

Pojďme rozebrat, jak v takovém grafu provést potřebné operace přebarvení a spojování vrcholů, a dokázat, že po jejich provedení odpovídají délky nejkratších cest mezi všemi dvojicemi vrcholů z původního grafu těm skutečným.

Zkusme nejprve přebarvit vrcholy barvy i barvou j. Založíme barvě j nový vstupní vrchol a výstupní vrchol, ty původní v grafu nicméně ponecháme. Z nového vstupního vrcholu barvy j natáhneme orientované hrany délky 0 do původního vstupního vrcholu barvy j a vstupního vrcholu barvy i. Stejně tak do nového výstupního vrcholu barvy j natáhneme orientované hrany délky 0 z původního výstupního vrcholu barvy j a výstupního vrcholu barvy i.

Všechny původní hrany v grafu zůstaly, takže jsme zachovali všechny cesty mezi původními vrcholy a tedy i mezi vrcholy Jirkova grafu. Zároveň jsme nepropojili žádné vrcholy z Jirkova grafu, protože z nově přidaného výstupního vrcholu nevede žádná hrana a do nově přidaného vstupního vrcholu žádná hrana nevstupuje.

Přebarvení jsme zvládli, pojďme propojit všechny vrcholy barvy i se všemi vrcholy barvy j. To provedeme natáhnutím orientované hrany délky 1 z výstupního vrcholu barvy i do vstupního vrcholu barvy j a orientované hrany délky 1 z výstupního vrcholu barvy j do vstupního vrcholu barvy i.

Tím jsme docílili toho, že mezi každým vrcholem barvy i a každým vrcholem barvy j existuje v obou směrech orientovaná cesta s celkovým součtem délek rovným jedné, což je stejné, jako kdybychom tyto vrcholy propojili hranami. Zároveň platí, že jsme nepropojili nic, co být propojeno nemělo. Natáhli jsme sice nové hrany do vstupních vrcholů, nicméně z každého vstupního vrcholu se dá dostat jen do vrcholů jeho barvy.

Ukázali jsme, že tyto operace fungují tak, jak bylo zamýšleno. Proto bude délka nejkratší cesty v tomto grafu odpovídat té v Jirkově grafu.

Hledání nejkratší cesty

Zbývá nám nalézt nejkratší cestu mezi vrcholy u a v. Protože mají hrany různé délky, nabízí se použít Dijkstrův algoritmus. Vyhneme se nicméně příliš pomalé binární haldě a využijeme skutečnosti, že hrany v našem grafu nabývají jen délek 0 nebo 1.

Z toho plyne, že kdykoliv pomocí aktuálně zpracovávaného vrcholu relaxujeme sousedy, budou vzdálenosti sousedů od vrcholu u nejvýše o 1 větší, než vzdálenost aktuálně zpracovávaného vrcholu. Vrcholy takové vzdálenosti začneme z haldy odstraňovat, až když zpracujeme všechny vrcholy mající stejnou vzdálenost od u jako aktuální vrchol. Rozmysleme si, že celkový počet různých vzdáleností v haldě je potom nejvýše 2 a rozdíl těchto vzdáleností je 1.

Efektivnější haldu pro tento případ můžeme implementovat například pomocí obousměrné fronty. V té budeme uchovávat vrcholy seřazené sestupně podle jejich vzdálenosti od vrcholu u. Aktuální minimum se bude nacházet na konci fronty. Popišme, jak do této haldy vložit nový vrchol. Pokud je vzdálenost vkládaného vrcholu stejná jako aktuálně zpracovávaného vrcholu, je menší nebo rovná minimu ve frontě a můžeme proto vrchol bezpečně vložit na konec fronty. Pokud je naopak vzdálenost vkládaného vrcholu vyšší než vzdálenost aktuálně zpracovávaného vrcholu, musí být díky pozorování o počtu hodnot a jejich rozdílu v haldě jeho vzdálenost větší nebo rovná vzdálenosti vrcholu na začátku fronty. Můžeme jej proto v takovém případě vložit na začátek fronty.

Protože vkládání i odstranění vrcholu z fronty trvá $\mathcal{O}(1)$, nalezneme nejkratší cestu v grafu v lineárním čase vzhledem k jeho velikosti.

Časová a paměťová složitost

Pojďme rozebrat celkovou časovou i paměťovou složitost. Na počátku založíme graf o $\mathcal{O}(n)$ vrcholech a hranách a pomocné pole délky $\mathcal{O}(n)$. V každém z následujících q kroků provedeme $\mathcal{O}(1)$ operací a do grafu přidáme $\mathcal{O}(1)$ hran a vrcholů. Stavba grafu tak zabere $\mathcal{O}(n+q)$ času a prostoru a výsledný graf bude mít $\mathcal{O}(n+q)$ hran i vrcholů. Nakonec v $\mathcal{O}(n+q)$ nalezneme nejkratší cestu s využitím $\mathcal{O}(n+q)$ paměti.

Naše řešení má tedy celkovou časovou i paměťovou složitost rovnou $\mathcal{O}(n+q)$.

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

using namespace std;

struct Edge {
    int node;
    int weight;
};

int V, Q;

// Pole reprezentující graf. Pro každý vrchol obsahuje seznam sousedů
// a ohodnocení hrany, která je spojuje.
vector<vector<Edge>> graph;

// Pole ukládající pro každou barvu aktuální vstupní a výstupní vrchol
vector<int> in;
vector<int> out;

void connect(int u, int v) {
    // Propojíme vstupní a výstupní vrcholy hranami délky 1.
    graph[out[u]].push_back({ in[v], 1 });
    graph[out[v]].push_back({ in[u], 1 });
}

void paint(int u, int v) {
    // Přidáme nový vstupní vrchol vrcholu v a propojíme jej
    // s původními vstupními vrcholy u a v.
    graph.push_back({ {in[u], 0}, {in[v], 0} });
    in[v] = graph.size() - 1;

    // Přidáme nový výstupní vrchol vrcholu v a propojíme jej
    // s původními výstupními vrcholy u a v.
    graph.push_back({});
    int s = graph.size() - 1;
    graph[out[u]].push_back({ s, 0 });
    graph[out[v]].push_back({ s, 0 });
    out[v] = s;
}

int dijkstra(int source, int destination, vector<vector<Edge>> graph) {
    deque<Edge> heap;
    vector<int> distances(graph.size(), -1);

    heap.push_back({ source, 0 });
    while (!heap.empty()) {
        int currentNode = heap.front().node;
        int currentDist = heap.front().weight;
        // Vezmeme aktuálni prvek zepředu fronty.
        heap.pop_front();

        if (currentNode == destination) {
            return currentDist;
        }
        if (distances[currentNode] != -1) {
            continue;
        }

        distances[currentNode] = currentDist;
```

```
        for (auto edge : graph[currentNode]) {
            int newDist = currentDist + edge.weight;
            // Vzdálenost souseda je stejná jako aktuálního,
            // takže vložíme souseda na předek fronty.
            if (newDist == currentDist) {
                heap.push_front({ edge.node, newDist });
            } else {
            // Vzdálenost souseda je větší, takže vložíme souseda
            // na konec fronty.
                heap.push_back({ edge.node, newDist });
            }
        }
    }
    return -1;
}

int main() {
    cin >> V >> Q;
    // Uděláme si místo na vrcholy, včetně vstupních a výstupních.
    graph.resize(3 * V);
    in.resize(V);
    out.resize(V);

    int source, destination;
    cin >> source >> destination;

    // Propojíme vrcholy s odpovídajícími vstupními a výstupními vrcholy.
    for (int i = 0; i < V; i++) {
        // i + V = out
        // i + 2V = in
        graph[i].push_back({ i + V, 0 });
        out[i] = i + V;
        graph[i + 2 * V].push_back({ i, 0 });
        in[i] = i + 2 * V;
    }

    // Postavíme na základě instrukcí zbytek grafu.
    for (int i = 0; i < Q; i++) {
        char op;
        int u, v;
        cin >> op >> u >> v;

        if (op == 'p') {
            paint(u, v);
        } else {
            connect(u, v);
        }
    }

    // Nalezneme nejkratší cestu v postaveném grafu.
    cout << dijkstra(source, destination, graph) << endl;
    return 0;
}
```

P-III-5 Jednosměrná jednokolejka

Základní návrh

Základní, nezoptimalizovaný návrh algoritmu bude následující: Budeme postupovat od začátku dne, tedy od minuty 0, a v každé minutě pro všechny stavy jednokolejky (kdy a kde na ní něco je) a možnosti držených vlaků spočítáme, jakého nejmenšího zpoždění lze dosáhnout, když skončíme v daném stavu. Na začátku máme pouze stav, kdy žádné vlaky nečekají, a jednokolejka je prázdná, s nejlepší zpožděním 0.

Dále v každé další minutě n uděláme následující: Už máme všechny výsledky toho, jakého nejmenšího zpoždění lze dosáhnout v minutě $n-1$ s daným konečným stavem. Ke všem těmto možnostem přičteme zpoždění držených vlaků za uběhlou minutu. Pokud v minutě n přijely vlaky, tak je přidáme do všech stavů jako držené.

Následně pro každou možnost držených vlaků zkusíme vypustit nějaké z nich. Stav jednokolejky určí zpoždění těchto vlaků, a nový stav jednokolejky. Pokud výsledné zpoždění pro zbývající držené vlaky bude lepší, než doposud nejlepší, tak si ho zapamatujeme.

Tento algoritmus projde všechny možnosti, a tedy je korektní, ale není efektivní. Mnoho stavů, které prochází, nevedou k lepšímu řešení, a tedy je nemusíme zkoušet.

Ořezání stavového prostoru

Nejprve si všimneme, že si nemusíme pamatovat celý stav jednokolejky, ale stačí nám vědět čas t, kdy z ní odjede poslední vlak. Tento čas nám totiž stačí, abychom určovali zpoždění následujících vlaků: Řekněme, že na jednokolejku pustíme vlak. Pokud tento vlak nebude brzděn jiným vlakem, který již je na jednokolejce, tak na předchozích vlacích nezáleží. Pokud nás nějaký vlak bude brzdit, tak jednokolejku opustíme spolu s ním, což je právě v čas t.

Dále zlepšíme situaci s drženými vlaky: Prvním důležitým pozorováním je, že nikdy nemusíme držet expresy před jednokolejkou. Jelikož expres je vždy stejně rychlý nebo rychlejší než ostatní vlaky, nemůže nikdy způsobit zpoždění jinému vlaku. Jeho zdržení by tedy pouze mohlo způsobit zpoždění samotnému expresu, což se nikdy nevyplatí.

Druhým důležitým pozorováním je, že vlaky se vyplatí pouštět pouze v časech, když je nějaký vlak zrovna připraven k odjezdu. Pokud v minutě t není žádný vlak připraven k odjezdu, tak bychom mohli všechny vlaky, co nyní pouštíme, vypustit o minutu dříve, ve stejném pořadí, což může čas odjezdu posledního vlaku a zpoždění pouze zmenšit.

Třetím pozorováním je, že pokud vypustíme vlak daného typu, tak můžeme vypustit všechny vlaky tohoto typu: Pokud bychom některý z těchto vlaků vypustili později, tak musí také dojet později, a tedy mít vyšší zpoždění. Jelikož vypouštíme alespoň jeden vlak tohoto typu, tak další vlaky nezpůsobí žádné další zpoždění.

Čtvrtým pozorováním, které potřebujeme, je to, že pokud v dané minutě vypustíme nějaký vlak, tak před ním můžeme vypustit všechny vlaky, které jsou rychlejší,

než tento vlak. Tyto vlaky nemůžou způsobit zpoždění našemu vlaku, a jelikož pojedou před naším vlakem, tak ani žádnému dalšímu. Kdybychom je poslali až po našem vlaku, tak by dojeli do cíle stejně nebo později, takže není potřeba s nimi čekat.

Tyto pozorování exponenciálně sníží potřebný počet stavů, které budeme uvažovat. Dokonce nám jich bude stačit řádově n^2. Stačí nám pouze uvažovat stavy, kde si pamatujeme, kdy naposled odjely rychlíky, a kdy naposledy odjely osobáky. Podle prvního pozorování můžeme vynechat expresy. Navíc podle těchto časů odjezdu můžeme určit, kdy z jednokolejky odjede poslední vlak, což je poslední část stavu, kterou potřebujeme.

Rychlý algoritmus

Díky druhému pozorování nemusíme iterovat přes všechny časy t, ale přes jen časy, kdy se připraví nějaký vlak. Tyto časy nazveme p_i, číslované od 1 do n. Nechť $D[r][o]$ je tabulka velikosti $(n+1) \times (n+1)$, kde $D[r][o]$ je nejmenší možné zpoždění, pokud jsme naposledy vypustili rychlíky v p_r, a osobní vlaky v p_o. Případ $r = 0$ (resp. $o = 0$) bude znamenat, že jsme žádné rychlíky (resp. osobní vlaky) ještě nevypustili.

Začneme opět s jedním stavem: $D[0][0] = 0$. Zbytek nastavíme na ∞, neboť nemůžou nastat. Potom postupujeme přes všechny časy p_i.

Na začátku zpracovávání času p_i aktualizujeme tabulku D, aby popisovala zpoždění v čase p_i místo p_{i-1}. Ve stavu $D[r][o]$ každý čekající vlak nabude $p_i - p_{i-1}$ zpoždění. Ještě potřebujeme zjistit kolik takových vlaků je, tedy kolik rychlíků se připraví mezi časy r a p_i, a kolik osobních vlaků se připraví mezi o a p_i. Tohle lze předpočítat například pomocí prefixových součtů.*

Pokud v čase p_i byl k odjezdu připraven expres, tak jej ke každému stavu $D[r][o]$ rovnou přidáme. Jeho čas opuštění jednokolejky získáme jako maximum z $\{o+t_{\mathrm{Os}}, r+t_{\mathrm{R}}, p_i + t_{\mathrm{Ex}}\}$, a zpoždění je toto číslo bez $p_i + t_{\mathrm{Ex}}$. Pokud přijede jiný vlak, tak je vzhledem k způsobu reprezentace stavů držen před jednokolejkou.

Dále budeme pouštět rychlíky a pak osobní vlaky, díky čtvrtému pozorování. Pouštění rychlíků znamená, že se pokusíme zlepšit $D[i][o]$, pomocí $D[r][o]$, pro všechna r a o. Jiné pouštění zkoušet nemusíme, díky třetímu pozorování. Tyto rychlíky, které vypustíme, dorazí v čase $\{o + t_{\mathrm{Os}}, p_i + t_{\mathrm{R}}\}$, z čehož získáme zpoždění každého z nich, a jejich počet získáme pomocí prefixových součtů.

Nakonec pustíme osobní vlaky. Díky čtvrtému pozorování je stačí vypouštět ze stavů, kdy už jsme vypustili všechny rychlíky. Budeme tedy zlepšovat $D[i][i]$ pomocí $D[i][o]$, pro všechna o. Osobní vlak nejde zdržet, takže stačí udělat $D[i][i] = \min(D[i][i], D[i][o])$.

Jelikož procházíme všechny stavy kromě těch, které jsme odargumentovali jako suboptimální, tak je algoritmus korektní.

---

* Více najdete na *https://ksp.mff.cuni.cz/encyklopedie/zakladni-algoritmy/* v sekci *Prefixové součty.*

Algoritmus potřebuje $\mathcal{O}(n^2)$ paměti na tabulku D a na zbytek jen $\mathcal{O}(n)$. Celková paměťová složitost je tedy $\mathcal{O}(n^2)$.

Časová složitost je $\mathcal{O}(n^3)$: Hlavní částí je iterace přes p_i, kterých je n. V každé iteraci děláme konečný počet operací pro každé políčko v tabulce D, tedy $\mathcal{O}(n^2)$.

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

using namespace std;

using ll = long long;

const ll INF = 1e18;

struct ArrivalTime {
    ll time;
    ll os_count;
    ll r_count;
    ll ex_count;
};

int main() {
    ll train_count, os_travel_time, r_travel_time, ex_travel_time;
    cin >> train_count >> os_travel_time >> r_travel_time >> ex_travel_time;

    vector<ArrivalTime> arrival_times;
    arrival_times.push_back({-INF, 0, 0, 0});

    for (ll i = 0; i < train_count; i++) {
        ll arrival_time; char type[3];
        cin >> arrival_time >> type;

        if (arrival_times.back().time < arrival_time) {
            auto [_, os_count, r_count, ex_count] = arrival_times.back();
            arrival_times.push_back({arrival_time, os_count, r_count, ex_count});
        }

        if (strcmp(type, "Os") == 0) {
            arrival_times.back().os_count += 1;
        } else if (strcmp(type, "R") == 0) {
            arrival_times.back().r_count += 1;
        } else if (strcmp(type, "Ex") == 0) {
            arrival_times.back().ex_count += 1;
        }
    }

    // dp[os][r] - arrival_times[os].time = maximální čas odjezdu posledního Os,
    // arrival_times[r].time = maximální čas odjezdu posledního R
    vector<vector<ll>> dp(arrival_times.size(),
                          vector<ll>(arrival_times.size(), INF));
    dp[0][0] = 0;

    for (int i = 1; i < arrival_times.size(); i++) {

        // Zpoždění čekajících vlaků
        for (int os = 0; os < i; os++) {
            for (int r = 0; r < i; r++) {
                ll waiting_train_count = arrival_times[i - 1].os_count
                                       - arrival_times[os].os_count;
                waiting_train_count += arrival_times[i - 1].r_count
                                     - arrival_times[r].r_count;
```

```
                dp[os][r] += waiting_train_count
                             * (arrival_times[i].time - arrival_times[i - 1].time);
            }
        }

        // zpoždění Ex
        for (int os = 0; os <= i; os++) {
            for (int r = 0; r <= i; r++) {
                ll last_train_finish_time =
                    max(arrival_times[os].time + os_travel_time,
                        arrival_times[r].time + r_travel_time);
                ll ex_trains_delay =
                    max(0LL, last_train_finish_time
                             - (arrival_times[i].time + ex_travel_time));
                dp[os][r] +=
                    (arrival_times[i].ex_count - arrival_times[i - 1].ex_count)
                    * ex_trains_delay;
            }
        }

        // zpoždění R
        for (int os = 0; os <= i; os++) {
            for (int r = 0; r < i; r++) {
                ll last_train_finish_time = max(0LL, arrival_times[os].time + os_travel_time)
                ll r_trains_delay =
                    max(0LL, last_train_finish_time
                             - (arrival_times[i].time + r_travel_time));
                ll r_depart_count =
                    arrival_times[i].r_count - arrival_times[r].r_count;
                dp[os][i] =
                    min(dp[os][i], dp[os][r] + r_depart_count * r_trains_delay);
            }
        }

        // zpoždění Os
        for (int os = 0; os < i; os++) {
            for (int r = 0; r <= i; r++) {
                dp[i][r] = min(dp[i][r], dp[os][r]);
            }
        }
    }

    cout << dp[arrival_times.size() - 1][arrival_times.size() - 1] << endl;
}
```

P-III-6 Počítáme s tříděním

Při psaní generátoru se hodí si nejprve nadefinovat pomocné funkce na vypisování instrukcí. Nejjednodušší je napsat jednu funkci `inst(...)`, která vypíše své argumenty jako instrukci a vrátí číslo této instrukce. To nám umožní zapsat výraz „jádro s ID 1 vrátí své oblíbené číslo, ostatní jádra oblíbené číslo jádra nalevo“ jako `inst("if", inst("=", inst("id"), inst("const", 1)), inst("input", 2), inst("left", inst("input", 2)))`. O něco přehlednější je nadefinovat funkci pro každou z instrukcí.

V mnoha jazycích je ale možné zajít ještě o krok dále a vytvořit třídu reprezentující výsledek instrukce a na ní nadefinovat operátory, které vypíšou příslušnou instrukci a vrátí její výsledek. V tomto řešení budeme pro ukázkové kusy kódu používat právě takovou třídu napsanou v Pythonu. Ta nám umožní výše uvedený výraz zapsat jako `(Result.id() == 1).switch(Result.input(2), Result.input(2).left())`. Kód této třídy najdete na konci řešení. Můžete si všimnout, že:

- Konstanty je možné psát přímo jako konstanty, instrukce `const` se vypíše automaticky.
- Instrukce `id` a `input` vytváří statické metody.
- Instrukce `left`, `right` a `sort` vytváří normální metody.
- Instrukci `if` vytváří metoda `switch`, protože `if` je v Pythonu klíčové slovo.

Pokud je N malé ($N \leq 2^7$)

Pro získání jednoho bodu stačí úlohu vyřešit v lineárním čase. To lze udělat například tak, že jádra N-krát pošlou uložené oblíbené číslo a ID doleva. Po každém kroku si jádro toto oblíbené číslo uloží, pokud je příslušné ID rovno ID jejich idolu.

Program (Python 3):

```
def main(cores: int) -> Result:
    idol = Result.input(1)
    id = Result.id()
    number = Result.input(2)
    answer = -1

    for _ in range(cores):
        answer = (id == idol).switch(number, answer)
        number = number.left()
        id = id.left()

    return answer
```

Pokud jádra mají stejné idoly ($d = \mathtt{S}$)

V sadě testů za další bod můžeme předpokládat, že všechna jádra mají stejný idol. Bude tedy existovat jedno jádro, které je svým vlastním idolem. Toto jádro předá své oblíbené číslo jádru s ID 0, podobně, jako jsme v domácím kole do jádra s ID 0 ukládali nejmenší číslo. Poté toto číslo zkopírujeme do všech jader, podobně jako jsme v krajském kole do všech jader kopírovali součet. Tím získáme řešení v $\mathcal{O}(\log N)$.

Program (Python 3):

```
# Jádro, co je svým vlastním idolem, předá oblíbené číslo jádru s ID 0.
def find_common_answer() -> Result:
    idol = Result.input(1)
    number = Result.input(2)

    is_correct = idol == Result.id()
    is_correct.sort()

    return is_correct.switch(number, number.left())

# Z jádra s ID 0 odpověď zkopírujeme do všech jader.
def distribute(answer: Result, cores: int) -> Result:
    id = Result.id()
    id.sort()

    correct_count = 1

    while correct_count < cores:
        (id % correct_count).sort()
        answer = (id < correct_count).switch(answer, answer.left())
        correct_count *= 2

    return answer

def main(cores: int) -> "Result":
    return distribute(find_common_answer(), cores)
```

Pokud jádra mají různé idoly a idoly mají opačnou paritu ($d = \texttt{R}, p = \texttt{P}$)

V dalších dvou řešeních budeme předpokládat, že žádná dvě jádra nemají stejný idol.

Chtěli bychom jádra seřadit tak, aby každé jádro bylo vedle svého idolu. To ale není snadné – když jádra seřadíme podle ID jejich idolu, ale pak tento idol udělá totéž a přesune se jinam. Ve dvou sadách testů naštěstí platí poněkud zvláštní podmínka – jádra se sudým ID mají idol s lichým ID, a naopak. To nám umožní úlohu vyřešit nadvakrát. V první fázi se oblíbené číslo svého idolu dozví jádra se sudým ID, v druhé jádra s lichým ID.

Jádra si v každé fázi rozdělíme na ta, co číslo „vysílají“, a na ta, co ho „přijímají“. Jádra nejprve seřadíme podle jejich role, např. přijímající napravo. Poté jádra seřadíme podle vhodně zvoleného klíče. Klíčem vysílajících jader bude jejich ID, klíčem přijímajících jader bude ID jejich idolu. Tím docílíme toho, že přijímající jádra budou hned napravo od jejich idolu. Mohou si tedy snadno přečíst jeho oblíbené číslo. Toto řešení běží v $\mathcal{O}(1)$.

Program (Python 3):

```
def main(cores: int) -> Result:
    idol = Result.input(1)
    number = Result.input(2)

    # Provedeme jedno kolo vysílání.
    def transmit(receiver_parity: int, default: Operand) -> Result:
        is_receiver = Result.id() % 2 == receiver_parity
```

```
        # Jádra seřadíme, aby přijímající jádra byla hned napravo od jejich idolů.
        is_receiver.sort()
        key = is_receiver.switch(idol, Result.id())
        key.sort()

        return is_receiver.switch(number.left(), default)

    # Lichá jádra pošlou čísla sudým.
    result = transmit(0, -1)
    # Sudá jádra pošlou čísla lichým.
    result = transmit(1, result)

    return result
```

Pokud jádra mají různé idoly, poprvé ($d = \texttt{R}$)

Pro tuto podúlohu známe dvě pěkná řešení, ukážeme si obě. Obě vycházejí z předchozího řešení.

Předchozí řešení funguje v případě, že ID jádra a ID jeho idolu mají jiný nejméně významný bit. Jádra, která tuto podmínku nesplňují, odpoví špatně (což umí poznat), ale ostatní jádra nijak negativně neovlivní. Toto řešení můžeme snadno upravit, aby fungovalo, pokud se ID jádra a ID idolu liší v i-tém bitu, pro libovolné i. Spustíme ho tedy $\log N$-krát, pro každý bit v zápisu identifikačních čísel jednou. Tím jsme docílili toho, že jádro zná správnou odpověď, pokud se jeho ID a ID jeho idolu liší v alespoň jednom bitu. V opačném případě musí být svým vlastním idolem, tedy může odpovědět svým vlastním oblíbeným číslem. Tím získáme řešení v $\mathcal{O}(\log N)$.

Program (Python 3):

```
def main(cores: int) -> Result:
    idol = Result.input(1)
    number = Result.input(2)

    def transmit(bit: int, receiver_parity: int, default: Operand) -> Result:
        parity = (Result.id() & 1 << bit) != 0
        is_receiver = parity == receiver_parity

        is_receiver.sort()
        is_receiver.switch(idol, Result.id()).sort()

        return (Result.id().left() == idol).switch(number.left(), default)

    # Tato hodnota se použije, pokud se ID jádra a jeho idolu neliší v žádném bitu.
    result = number

    for i in range(cores.bit_length()):
        # Jádra, co mají v i-tém bitu ID 0, přijmou čísla od těch, co tam mají 1.
        result = transmit(i, 0, result)
        # Jádra, co mají v i-tém bitu ID 1, přijmou čísla od těch, co tam mají 0.
        result = transmit(i, 1, result)

    return result
```

Pokud jádra mají různé idoly, podruhé ($d = \texttt{R}$)

Předchozí řešení jádra dělí na vysílající a přijímající. Problém je v tom, že každé jádro musí jak vysílat, tak přijímat, takže takové rozdělení není možné. Tento

problém vyřešíme pomocí síly přátelství. Jádra rozdělíme do dvojic – nulté s prvním, druhé s třetím, a tak dále. Kamarádi si vymění ID, oblíbená čísla a ID idolů. Sudé jádro ve dvojici bude mít za úkol za obě jádra přijímat, zatímco liché jádro bude mít za úkol za obě jádra vysílat.

Vysílající jádra tedy mají za úkol vysílat dvě čísla, obdobně přijímající jádra mají dvě čísla přijmout. Toho můžeme snadno dosáhnout ve čtyřech vysílacích fázích. Nejprve jádra vysílají a přijímají své první číslo, potom vysílají druhé a přijímají první* a tak dále. Tím získáme řešení v $\mathcal{O}(1)$.

Program (Python 3):

```
def is_receiver() -> Result:
    return Result.id() % 2 == 0

def is_sender() -> Result:
    return Result.id() % 2 == 1

# Provedeme jedno kolo vysílání.
def transmit(idol: Operand, id: Operand, num: Operand, default: Operand) -> Result:
    is_receiver().sort()
    is_receiver().switch(idol, id).sort()

    return (id.left() == idol).switch(num.left(), default)

def main(cores: int) -> Result:
    idol = Result.input(1)
    number = Result.input(2)

    # Pro přijímající jádra:
    idol_1 = idol
    idol_2 = idol.right()

    # Pro vysílající jádra:
    id_1 = Result.id()
    id_2 = id_1 - 1
    number_1 = number
    number_2 = number.left()

    result_1 = transmit(idol_1, id_1, number_1, -1)
    result_1 = transmit(idol_1, id_2, number_2, result_1)
    result_2 = transmit(idol_2, id_1, number_1, -1)
    result_2 = transmit(idol_2, id_2, number_2, result_2)

    Result.id().sort()
    result = is_sender().switch(result_2.left(), result_1)

    return result
```

Obecné řešení

Pokud bude mít více jader stejný idol, tak se předchozí řešení rozbijí, protože v jedné fázi může napravo od jednoho vysílajícího jádra skončit více přijímajících,

* Tyto dvě fáze by bylo možné sloučit do jedné – vysílající jádro zastupuje jádra se sousedními ID, takže stačí jádra seřadit tak, aby přijímající jádra se sudým a lichým idolem skončila na různých stranách vysílajícího jádra.

a to až $N/2$. Není jednoduché zařídit, aby se každé z nich dozvědělo číslo příslušného vysílajícího jádra.

Použijeme tedy trik ze zástupci, který jsme používali v úloze *Nejčetnější číslo* z domácího kola. Jádra si na začátku seřadíme primárně podle jejich idolu a sekundárně podle jejich ID. Z každého intervalu jader se stejným idolem vybereme jedno, tentokrát to poslední, a nazveme jej *zástupcem*. Zástupce bude mít za úkol zjistit oblíbené číslo svého idolu a nakonec ho předat všem jádrům se stejným idolem. Pokud v nějaké fázi nějaké přijímající jádro není zástupce, pak se přesune na jeden z konců posloupnosti, aby zástupcům ve čtení nepřekáželo.* Toho můžeme dosáhnout tak, že mu ID idola dočasně změníme na -1.

Poté, co se zástupci dozví oblíbené číslo svého idola pomocí jednoho z předchozích algoritmů, tak jádra znovu seřadíme nejprve podle jejich ID, pak podle ID idola. Zbývá nějak řešení zkopírovat z každého zástupce na všechna jádra v jejich intervalu. Nabízí se v rámci každého intervalu použít algoritmus na kopírování čísel, který jsme použili v řešení druhé sady vstupů. To bohužel nefunguje – v rámci intervalu některá ID jader chybí, s čímž algoritmus nepočítá. Použijeme tedy stejný trik jako v úloze *Pořadí* z krajského kola a budeme v rekurzivním kroku odstraňovat jádra jen tehdy, když opravdu mají druhé jádro do dvojice.** Přesněji:

1. Seřadíme jádra podle skutečného ID, pak podle ID idola, pak přesuneme vyřazená na konec.
2. Pokud má jádro co není zástupce liché ID a jádro nalevo od něj má o jedna menší ID, pak jej vyřadíme.
3. Všechna ID pomyslně vydělíme dvěma.
4. Rekurzivně vyřešíme zbylá jádra.
5. Změny provedené v 2. a 3. kroku vrátíme zpátky, jádra seřadíme jako v 1. kroku.
6. Jádra, co jsme právě „odvyřadili“, si zkopírují hodnotu od jádra nalevo od nich.

Těchto rekurzivních kroků provedeme $\log N$. Pak bude virtuální ID všech jader 0, a tedy bude v každém intervalu jen jedno jádro, a to zástupce, který už řešení zná.

Tím získáme řešení celé úlohy v $\mathcal{O}(\log N)$.

Také je možné obdobným způsobem číslo zkopírovat v rámci každé fázi z vysílajícího jádra na všechna příslušná přijímající. Pokud tímto způsobem upravíme druhé z řešení pro $d = \mathtt{R}$, získáme také algoritmus v $\mathcal{O}(\log N)$. Pokud takto upravíme první řešení, tak získáme algoritmus v $\mathcal{O}(\log^2 N)$, což stačí alespoň na předposlední sadu testů.

---

* Můžete si rozmyslet, že to není potřeba, pokud před každou fází jádra seřadíme podle jejich ID.

** Také bychom mohli použít řešení úlohy *Pořadí* přímo a přečíslovat ID, aby souvislá byla.

Program (Python 3):

```
# Zjistíme, kdo je zástupce.
def elect_representatives(idol: Result) -> Result:
    Result.id().sort()
    idol.sort()

    # Zástupce je prvním jádrem v intervalu.
    # Může se stát, že mají všichni stejný idol,
    # proto bude jádro s ID 0 vždy zástupce.
    is_representative = (idol.left() != idol) | (Result.id() == 0)
    return is_representative

# Zkopírujeme řešení ze zástupců do ostatních jader.
def distribute_result(value: Result, idol: Result, cores: int):
    is_representative = elect_representatives(idol)

    def inner(id: Result, gone: Operand, bits: int) -> Result:
        # Pokud jsme provedli log N kroků, pak zbyl jen zástupce.
        if bits == 0:
            return value

        def sort():
            Result.id().sort()
            idol.sort()
            Result.operand(gone).sort()

        # Zjistíme, jestli se jádro právě stává vyřazeným.
        sort()
        is_going_away = id // 2 == id.left() // 2
        is_going_away &= is_representative ^ 1
        is_going_away &= gone ^ 1

        # Zbylá jádra se dozví řešení rekurzivně.
        result = inner(id // 2, gone | is_going_away, bits - 1)

        # Vyřazená jádra si zkopírují výsledek od jádra nalevo.
        sort()
        result = is_going_away.switch(result.left(), result)

        return result

    bits = cores.bit_length() - 1
    return inner(Result.id(), 0, bits)

NO_IDOL = -1

def main(cores: int) -> Result:
    idol = Result.input(1)
    optional_idol = elect_representatives(idol).switch(idol, NO_IDOL)

    Result.id().sort()
    # Sem je potřeba doplnit jedno z předchozích dvou řešení.
    result = main_for_unique_idols(cores, idol=optional_idol)

    return distribute_result(result, idol, cores)
```

Pomocná třída (Python 3):

```
class Result:
    last_id = 0

    def __init__(self, number: int):
        self.number = number

    def __str__(self) -> str:
        return str(self.number)

    @staticmethod
    def raw_inst(name: str, *operands: list[int]) -> "Result":
        print(name, *operands)
        Result.last_id += 1
        return Result(Result.last_id)

    @staticmethod
    def operand(operand: "Operand") -> "Result":
        if isinstance(operand, int):
            return Result.const(operand)
        else:
            return operand

    @staticmethod
    def inst(name: str, *operands: list["Operand"]) -> "Result":
        return Result.raw_inst(name, *(Result.operand(v) for v in operands))

    @staticmethod
    def const(value: int) -> "Result": return Result.raw_inst("const", value)
    @staticmethod
    def input(index: int) -> "Result": return Result.raw_inst("input", index)
    @staticmethod
    def id() -> "Result": return Result.raw_inst("id")

    def left(self) -> "Result": return Result.inst("left", self)
    def right(self) -> "Result": return Result.inst("right", self)
    def copy(self) -> "Result": return Result.inst("copy", self)
    def sort(self) -> "Result": return Result.inst("sort", self)
    def switch(self, true: "Operand", false: "Operand") -> "Result":
        return Result.inst("if", self, true, false)

    def __add__(self, other: "Operand") -> "Result": return Result.inst("+", self, other)
    def __sub__(self, other: "Operand") -> "Result": return Result.inst("-", self, other)
    def __mul__(self, other: "Operand") -> "Result": return Result.inst("*", self, other)
    def __floordiv__(self, other: "Operand") -> "Result": return Result.inst("/", self, other)
    def __mod__(self, other: "Operand") -> "Result": return Result.inst("%", self, other)
    def __and__(self, other: "Operand") -> "Result": return Result.inst("&", self, other)
    def __or__(self, other: "Operand") -> "Result": return Result.inst("|", self, other)
    def __xor__(self, other: "Operand") -> "Result": return Result.inst("^", self, other)

    def __radd__(self, other: "Operand") -> "Result": return Result.inst("+", other, self)
    def __rsub__(self, other: "Operand") -> "Result": return Result.inst("-", other, self)
    def __rmul__(self, other: "Operand") -> "Result": return Result.inst("*", other, self)
    def __rfloordiv__(self, other: "Operand") -> "Result": return Result.inst("/", other, self)
    def __rmod__(self, other: "Operand") -> "Result": return Result.inst("%", other, self)
    def __rand__(self, other: "Operand") -> "Result": return Result.inst("&", other, self)
    def __ror__(self, other: "Operand") -> "Result": return Result.inst("|", other, self)
    def __rxor__(self, other: "Operand") -> "Result": return Result.inst("^", other, self)

    def __eq__(self, other: "Operand") -> "Result": return Result.inst("=", self, other)
    def __ne__(self, other: "Operand") -> "Result": return Result.inst("!=", self, other)
    def __lt__(self, other: "Operand") -> "Result": return Result.inst("<", self, other)
    def __gt__(self, other: "Operand") -> "Result": return Result.inst(">", self, other)
    def __le__(self, other: "Operand") -> "Result": return Result.inst("<=", self, other)
    def __ge__(self, other: "Operand") -> "Result": return Result.inst(">=", self, other)
```

```
Operand = Result | int

if __name__ == "__main__":
    cores = int(input().split()[0])
    main(cores).copy()
```