P-I-1 Vlaky

Vstup je tvořen množinou nerovností ve tvaru $T_A \geq T_B + m$ udávajících, že vlak A má vyjet alespoň o m minut později než vlak B. Pro řešení této úlohy bude pohodlnější se na ně dívat jako na nerovnosti ve tvaru $T_A + \ell \geq T_B$, kde $\ell = -m$ (tedy znegovat všechny časy na vstupu). Taková nerovnost se totiž vyskytuje i v jiném kontextu: Jestliže vzdálenost odsud do místa A je T_A a z A do B vede cesta délky ℓ, pak vzdálenost odsud do místa B je nanejvýš $T_A + \ell$. To nám umožní si úlohu přirozeně převést na problém hledání vzdáleností v orientovaném grafu* s hranami ohodnocenými jejich délkou: Vrcholy grafu budou vlaky, a pro každou nerovnost $T_A + \ell \geq T_B$ si mezi časy odjezdů přidáme hranu z A do B délky ℓ.

Přidejme si navíc speciální vrchol S a hrany délky 0 z S do všech ostatních vrcholů; ekvivalentně přidáme nový vlak S a nerovnost $T_S \geq T_A$ pro každý jiný vlak A. To zjevně nezmění existenci řešení, protože všechny původní nerovnosti musí stále být splněny a nerovnosti pro T_S snadno splníme například tím, že vlak S odjede současně s nejpozdějším z ostatních vlaků. Tím se vyhneme technickým komplikacím, které by jinak mohly vzniknout, kdyby část vlaků byla zcela nezávislých na zbytku.

Pro každý vlak A označme jako d_A délku nejkratší cesty z S do A v tomto grafu. Připomeňme, že v cestě v grafu se (na rozdíl od sledu) nemohou opakovat vrcholy, v grafu s omezeným počtem vrcholů je tedy různých cest z S do A cest pouze konečný počet. Navíc alespoň jedna taková cesta existuje (tvořená hranou z S do A), d_A je tedy jednoznačně určeno. Tvrdíme, že existuje-li řešení pro zadaný vstup, pak to, kde každý vlak A odjede v čase d_A, je jedno z nich.

Uvažujme libovolnou ze zadaných nerovností $T_A + \ell \geq T_B$, odpovídající hraně z A do B délky ℓ. Nechť P_A je nejkratší cesta z S do A, délky d_A.

- Pokud B na této cestě neleží, pak přidáním B na konec P_A vznikne cesta z S do B délky $d_A + \ell$, a d_B je z definice nanejvýš délka této cesty, požadovaná nerovnost $d_A + \ell \geq d_B$ tedy platí.
- Jinak má cesta P_A koncový úsek $v_0 v_1 \ldots v_k$, kde $v_0 = B$, $v_k = A$, vrcholy $v_0, \ldots, v_k$ jsou navzájem různé, a pro $i = 1, \ldots, m$ je v grafu hrana $v_{i-1}v_i$ délky ℓ_i a na vstupu tedy nerovnost $T_{v_{i-1}} + \ell_i \geq T_{v_i}$. Předpokládali jsme, že existuje alespoň jedno řešení splňující všechny tyto nerovnosti a tedy i jejich součet $T_B + \ell_1 + \cdots + \ell_k \geq T_A$, spolu s uvažovanou nerovností $T_A + \ell \geq T_B$ tedy dostáváme $\ell_1 + \cdots + \ell_k + \ell \geq 0$.

  Nechť ℓ_0 je součet délek hran počátečního úseku cesty P_A z S do B; jelikož d_B je délka nejkratší cesty z S do B, máme $d_B \leq \ell_0$. Délka d_A cesty P_A je

* O grafech se můžete více dozvědět například zde:
https://ksp.mff.cuni.cz/kucharky/grafy/

rovna součtu délek jejích hran, tedy $d_A+\ell = \ell_0+\ell_1+\cdots+\ell_k+\ell \geq \ell_0 \geq d_B$, čímž opět dostáváme požadovanou nerovnost.

Poznamenejme, že z tohoto argumentu lze také vyvodit, že pokud přiřazení délek nejkratších cest není přípustné řešení, v grafu existuje cyklus $v_0 v_1 \ldots v_k$ záporné délky $\ell_1 + \cdots + \ell_k + \ell < 0$. V tomto případě zjevně žádné řešení nemůže existovat, jelikož tento cyklus odpovídá nerovnostem $T_{v_0} + \ell_1 \geq T_{v_1}$, $T_{v_1} + \ell_2 \geq T_{v_2}$, …, $T_{v_{k-1}} + \ell_k \geq T_k$, $T_k + \ell \geq T_{v_0}$, jejichž součtem je sporná nerovnost

$$\left(\sum_{i=0}^{k} T_{v_i}\right) + (\ell_1 + \cdots + \ell_k + \ell) \geq \sum_{i=0}^{k} T_{v_i}.$$

Jak ale délky všech nejkratších cest z S určit? Nastavme všem vrcholům A počáteční řešení $T_A = 0$ odpovídající tomu, že se do nich přímo dostaneme po hraně délky 0 z S. Zkusme nyní projít všechny nerovnosti a „opravit“ ty nesplněné: Je-li v grafu hrana z A do B délky ℓ a přitom platí opačná nerovnost $T_A + \ell < T_B$, snižme T_B na $T_A+\ell$, čímž zajistíme platnost této nerovnosti. Tvrdíme, že existuje-li řešení, pak tento postup zajišťuje platnost důležité nerovnosti $d_B \leq T_B$ pro každý vlak B: Počáteční řešení všechny tyto nerovnosti splňuje, jelikož v něm je $T_B = 0$ a d_B je zjevně nanejvýš délka hrany z S do B, tedy také 0. Předpokládejme, že tato nerovnost všude platí až dokud nesnížíme hodnotu T_B, a speciálně tedy platí $d_A \leq T_A$. Dle výše uvedené argumentace je přiřazení délek nejkratších cest jedním z řešení, splňuje tedy nerovnost $d_A+\ell \geq d_B$, a proto nová hodnota $T_A+\ell$ pro vlak B splňuje $T_A + \ell \geq d_A + \ell \geq d_B$.

Samozřejmě po jednom průchodu přes nerovnosti jsme mohli opravením jedné z nich pokazit nějakou jinou; proto budeme tento průchod několikrát opakovat. Tvrdíme ale, že to se není potřeba dělat mnohokrát. Konkrétně, jako *úroveň* vrcholu A označme nejmenší možný počet hran cesty délky d_A z S do A; úroveň každého vrcholu je zjevně nejvýše t. Tvrdíme, že pokud existuje řešení, pak po r průchodech mají všechny vrcholy B úrovně nejvýše r přiřazenou hodnotu $T_B = d_B$, a ta se díky výše zdůvodněné nerovnosti $d_B \leq T_B$ nebude dále měnit; stačí tedy provést nejvýše t průchodů. Proč je tomu tak? Předpokládejme, že tvrzení platí po $r-1$ průchodech. Dle definice úrovně existuje v grafu cesta P_B délky d_B s nejvýše r hranami z S do B. Nechť A je předchůdce B na této cestě a ℓ je délka hrany z A do B. Pak cesta $P_B - B$ z S do A má délku $d_B - \ell$. Jelikož přiřazení délek nejkratších cest je platné řešení, splňuje nerovnost $d_A + \ell \geq d_B$, a tedy délka $P_B - B$ je nejvýše d_A. Protože d_A je délka nejkratší cesty z S do A, cesta $P_B - B$ nemůže být kratší, a má tedy délku přesně d_A. Cesta $P_B - B$ s nanejvýše $r-1$ hranami tedy ukazuje, že úroveň A je nejvýše $r-1$, a protože tvrzení platí po $r-1$ průchodech, máme $T_A = d_A$. Po snížení T_B tak, aby platila nerovnost $T_A + \ell \geq T_B$, tedy máme $T_B \leq T_A + \ell = d_A + \ell = d_B$. Jak jsme ukázali výše, naopak platí i $d_B \leq T_B$, a proto pro každý vrchol B úrovně nejvýše r skutečně platí $d_B = T_B$.

Výše popsaný postup samozřejmě nefunguje, pokud žádné řešení neexistuje. Jak tuto situaci detekovat? Stačí po t průchodech ověřit, zda jsou všechny nerovnosti splněné (a našli jsme tedy platné řešení) nebo ne (a pak dle rozboru výše nejdou

splnit). Jinak řečeno, pokud bychom měli začít průchod opakovat $(t+1)$-krát, odpověď je „nelze“.

Jelikož provedeme pouze t průchodů přes $t+o$ nerovností, tento algoritmus (kterému se říká Bellmanův-Fordův*) má časovou složitost $\mathcal{O}(t^2+to)$. Ukládat si v něm musíme graf a $t+1$ čísel tvořících aktuální řešení, paměťová složitost je tedy $\mathcal{O}(t+o)$.

Program (C++):

```
#include <cstdio>
#include <vector>

const int INF = 1e9 + 1;

/* Popis nerovnosti.  */

class nerovnost {
    int A, B;
    int ell;

  public:
    nerovnost(int _A, int _B, int _ell) : A(_A), B(_B), ell(_ell) {}

    /* Sniž hodnotu B tak, aby platila nerovnost HODNOTY[A] + ell >= HODNOTY[B].
       Vrací true, pokud hodnotu B bylo nutné měnit.  */
    bool vynut(std::vector<long long> &hodnoty) const {
        if (hodnoty[A] + ell >= hodnoty[B])
            return false;

        hodnoty[B] = hodnoty[A] + ell;
        return true;
    }
};

/* Aktuální vzdálenost.  */
static std::vector<long long> T;

/* Seznam nerovností.  */
static std::vector<nerovnost> nerovnosti;

/* Postupně vynutí platnost každé NEROVNOSTI (čímž ale může další nerovnosti
   pokazit), vrací true, pokud došlo k nějaké změně.  */
static bool pruchod(const std::vector<nerovnost> &nerovnosti,
                    std::vector<long long> &hodnoty) {
    bool ret = false;

    for (const nerovnost &n : nerovnosti)
        ret |= n.vynut(hodnoty);

    return ret;
}

/* Opakuje průchod, dokud se hodnoty mění, maximálně ale R-krát.
   Vrací true, pokud se v R-tém průchodu nic nezměnilo (nebo se
   průchodů provedlo méně).  */
static bool opakuj_do_stabilizace(const std::vector<nerovnost> &nerovnosti,
```

* Více o Bellmanově-Fordovu algoritmu se můžete dočíst zde: *https://pruvodce.ucw.cz/static/pruvodce.pdf#page=152*

```
                                    std::vector<long long> &hodnoty, int r) {
    bool zmeneno = true;
    int iter;
    for (iter = 0; zmeneno && iter < r; iter++)
        zmeneno = pruchod(nerovnosti, hodnoty);

    /* Buď jsme skončili s iter < r, ZMENENO pak musí být false.
       Nebo probehlo iter = r průchodů a v posledním z nich se nic
       nezměnilo.  */
    return !zmeneno;
}

int main() {
    int t, o;
    scanf("%d%d", &t, &o);
    T.resize(t + 1, 0);
    nerovnosti.reserve(o + t);

    for (int i = 0; i < o; i++) {
        int A, B, m;
        scanf("%d%d%d", &A, &B, &m);
        nerovnosti.emplace_back(A, B, -m);
    }

    /* Přidáme nerovnosti pro speciální vlak S = 0.  */
    for (int v = 1; v <= t; v++)
        nerovnosti.emplace_back(0, v, 0);

    if (opakuj_do_stabilizace(nerovnosti, T, t)) {
        const char *sep = "";
        for (int v = 1; v <= t; v++) {
            printf("%s%d", sep, (int)T[v]);
            sep = " ";
        }
        printf("\n");
    } else {
        printf("nelze\n");
    }

    return 0;
}
```

P-I-2 Správa skladu

První sada vstupů zaručuje, že všechny nabídky jsou před všemi poptávkami. V takovém případě stihne Quark pouze jeden prodej. Mohli bychom si tedy pro každý typ předmětu poznamenat nejlepší nákupní a prodejní cenu, nakonec zvolit ten nejvýhodnější obchod (případně neobchodovat vůbec).

Stavy a přechody

V ostatních podúlohách je situace o něco komplikovanější. Pomůžeme si, když problém popíšeme pomocí stavů: ty se vyznačují jednak časem (které nabídce Quark zrovna čelí), druhak obsahem skladu (ten může být buď prázdný, nebo obsahovat konkrétní předmět).

Úlohu budeme řešit dynamickým programováním. Označme si $zisk[i][t]$ jakožto maximální možný zisk ve stavu po i-té nabídce/poptávce a předmětem t ve skladu.

Úplný začátek můžeme značit $i = 0$, prázdný sklad $t = 0$; potom $zisk[0][0]$ představuje počáteční stav (s nulovým výnosem). Na začátku není možné mít jakýkoliv předmět, příslušné stavy proto pro jednoduchost ohodnotíme ziskem $-\infty$. Abychom určili největší možný zisk i pro další stavy, musíme uvažovat všechny přechody a vždy zvolit ten nejlepší.

Každou nabídku/poptávku lze buď přijmout, nebo odmítnout. Pokud ji odmítneme, zisk ani obsah skladu se nezmění ($zisk[i][t] = zisk[i-1][t]$). Přijmout nabídku znamená zaplatit cenu a změnit obsah skladu ($zisk[i][t] = \max_{x=0}^{n}(zisk[i-1][x]) - c$). Přijmutím poptávky získáme odměnu, ale pouze výměnou za konkrétní předmět ($zisk[i][0] = zisk[i-1][t] + c$).

Projdeme časem od začátku do konce a postupně spočítáme maximální zisk pro všechny stavy – na konci zvolíme ten nejvyšší. Jak dlouho náš program poběží? Označme jako p počet typů předmětů. Stavů je celkem $\mathcal{O}(n \cdot p)$, mezi nimi $\mathcal{O}(p)$ přechodů za každou nabídku/poptávku, dohromady je tedy časová složitost $\mathcal{O}(n \cdot p)$. Řešení je tudíž efektivní jak pro podúlohu s jedním předmětem, tak pro sadu vstupů s nízkým n.

Optimalizace přechodů

Abychom řešení zrychlili, můžeme si všimnout, že se mezi sousedními časy mnoho zisků nemění: konkrétně pouze ty pro stavy s prázdným skladem a zrovna nabízeným předmětem. Stačí mít tedy uložený jen nejnovější zisk pro každý předmět ve skladu a aktualizovat ty relevantní.

Druhé užitečné pozorování je, že se nikdy nevyplatí zahazovat předmět. Každý plán, který nějaký předmět zahazuje, lze totiž vylepšit tím, že se tento předmět nekoupí. Odtud plyne, že při nákupu stačí uvažovat pouze přechod $zisk[i][t] = zisk[i-1][0] - c$.

Toto řešení potřebuje vyhodnotit jen dva přechody za každou nabídku i poptávku (jestli ji přijmeme, nebo ne), celkem si tedy vystačí s $\mathcal{O}(n)$ času. Potřebujeme načíst vstup a udržovat si nejlepší zisk pro každý možný obsah skladu, tudíž je i prostorová složitost $\mathcal{O}(n)$.

Program (C++):

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

using namespace std;

const int INF = 1e9 + 1;

vector<int> max_profit;
// max_profit[0] - nejvyšší možný zisk s prázdným skladem
// max_profit[i] - nejvyšší možný zisk s předmětem *i* ve skladu

int main() {
    int n;
    cin >> n;

    max_profit.assign(n + 1, -INF);
    max_profit[0] = 0; // začínáme s prázdným skladem a ziskem 0
```

```
    for (int i = 0; i < n; i++) {
        char X;
        int t, c;
        cin >> X >> t >> c;

        if (X == 'N') {
            max_profit[t] = max(max_profit[t], max_profit[0] - c);
        } else if (X == 'P') {
            max_profit[0] = max(max_profit[0], max_profit[t] + c);
        }
    }

    // prázdný sklad má vždy nejvyšší zisk
    cout << max_profit[0] << endl;
    return 0;
}
```

P-I-3 Stavba

Nejdřív si všimněme, že pokud existuje řešení, které vyhoví K požadavkům, tak existuje i řešení, které vyhoví prvním K požadavkům s nejmenší výškou. Vezměme libovolné řešení, které to nesplňuje. Označme výšku největšího splněného požadavku h a výšku nejmenšího nesplněného požadavku ℓ ($\ell < h$).

Nejdřív vyhoďme z řešení domky, které jsou vyšší než h. (A doplňme domky výšky 1 na začátek.) To nám nevyrobí přílišný rozdíl výšek – každý úsek domků, které jsme vyhodili, měl z obou stran domky výšky h, které jsou nyní vedle sebe. Dále vyhoďme jeden domek h – ten může sousedit s domky výšek h a $h-1$, takže stále nemáme příliš velký rozdíl výšek. Nyní zdvojme domek výšky ℓ (ten už tam je, protože začínáme na 1 a musíme vystoupat do $h-1$).

Tím jsme vyrobili posloupnost, která splňuje alespoň tolik požadavků, co předtím. Nicméně místo x do výšky nejmenších požadavků splňujeme nyní $x+1$. Tento proces můžeme opakovat, až se dostaneme do stavu, kdy jsou splněné požadavky jsou co nejnižší.

Nyní se opět zamysleme, jak naše řešení vypadá. Existuje totiž optimální řešení v následujícím tvaru:

$$k_1, \ldots, k_i, \ell_1, \ldots, \ell_j$$

Kde platí:

$$k_1 \leq \cdots \leq k_i > \ell_1 > \cdots > \ell_j$$

Pokud optimální řešení $h_i, \ldots, h_n$ není v tomto tvaru, tak ho do něj převedeme: Seřaďme všechny domky vzestupně. Tím jsme ale porušili podmínku $h_n \leq 1$. Nicméně si všimněme, že od každé výšky h' (kromě té nejvyšší) musíme mít alespoň dva domky. Začínáme totiž na 1 a musíme vystoupat až k nejvyšší, a pak zase musíme klesnout na 1. Nyní tedy stačí od každé výšky až na nejvyšší vzít jeden domek, seřadit je sestupně, a to jsou hledaná $\ell_1, \ldots, \ell_j$. Zbylá vzestupně seřazená posloupnost potom tvoří $k_1, \ldots, k_i$.

Teď, když už víme, jak řešení má vypadat (uspokojuje K nejmenších požadavků a je neklesající, pak klesající), pojďme ho sestrojit. Začneme s počáteční výškou $h_c \leftarrow 1$:

(1) Postavme dům h_c, a pokud existuje k němu zákazník, označme jeho požadavek za splněný.

(2) Pokud nám zbývají alespoň dva zákazníci požadující výšku h_c, zůstaňme na ní. (Musíme uspokojit K nejmenších požadavků, a jen jeden z nich zvládneme na cestě zpátky.)

(3) Jinak všechny požadavky h_c nebo nižší uspokojíme na cestě zpátky, nemá smysl tedy stavět domy této nebo nižších velikostí. $h_c \leftarrow h_c + 1$

(4) Pokud bychom ale měli příště postavit příliš vysoký dům, abychom skončili na 1, omezme se: $h_c \leftarrow \min(h_c, z)$, kde z je počet domů zbývajících postavit.

Určitě splníme K nejmenších požadavků, více splnit nestíháme. Ještě ověřme, že posloupnost je opravdu v hledaném tvaru – Dokud se neaktivuje podmínka (3), tak h_c neklesá. Jakmile ji aktivujeme, z bude klesat vždy o 1, tedy bude i h_c klesat o 1.

Ještě dořešme, jak si udržovat nevyřízené požadavky zákazníků. Všimněme si, že nepostavíme dům větší než N, protože do takové výšky nemáme čas vystoupat. A požadavky menší než N si můžeme udržovat v poli, kde a_i je počet požadavků na výšku i.

Celkem tedy potřebujeme $\mathcal{O}(N)$ času na předzpracování požadavků zákazníků, a pak v $\mathcal{O}(N)$ krocích vždy postavíme dům, a upravíme h_c. Celkem potřebujeme $\mathcal{O}(N)$ času. Paměťová složitost bude $\mathcal{O}(N)$, tak velké je pole nevyřízených požadavků zákazníků.

Program (Python 3):

```
def get_counts(arr, max_val):
    counts = [0] * max_val
    for val in arr:
        if val < len(counts):
            counts[val] += 1
    return counts

n = int(input())
heights = list(map(int, input().split()))

counts = get_counts(heights, n + 1)
result = []

current_value = 1
while len(result) < n:
    result.append(current_value)
    counts[current_value] -= 1

    # 1 domek prodáme na cestě zpátky
    if counts[current_value] <= 1:
        current_value += 1

    # Omezíme aktuální velikost, abychom nepostavili příliš vysoký domek
    current_value = min(current_value, n - len(result))

print(" ".join(map(str, result)))
```

P-I-4 Počítáme s tříděním

Minimum

První krok je poměrně přímočarý: jádra seřadíme podle jejich vstupu. Jádro s nejmenším vstupem tím dostaneme na začátek posloupnosti.

Poté přesuneme jádro s ID 0 na začátek posloupnosti, aby si mohlo onen nejmenší vstup přečíst.* Toho dosáhneme tím, že jádru s ID 0 přiřadíme klíč 0, zatímco ostatním jádrům přiřadíme klíč 1, a podle tohoto klíče jádra seřadíme. Třídění je stabilní, takže si všechna jádra s klíčem 1 zachovají pořadí, zatímco jádro s ID 0 se přesune na začátek.

Dosažené uspořádání jader. Vstup jádra je nahoře, ID dole.

Teď se nabízí, aby si každé jádro prostě přečetlo vstup jádra napravo od něj, protože napravo od jádra s ID 0 bude jádro s nejmenším vstupem. To ale nebude fungovat v případě, že mělo nejmenší vstup zrovna jádro s ID 0. Každé jádro proto spočítá minimum ze svého vstupu a vstupu jádra napravo od něj, což bude jeho výstup.

Program:

```
# Seřadíme jádra podle jejich vstupu
vstup:        input 1
              sort vstup

# Jádro s ID 0 přesuneme na začátek
id:           id
nula:         const 0
nemá_id_0:    != id nula
              sort nemá_id_0

# Spočítáme minimum ze vstupů jádra s ID 0 a jádra napravo od něj
vstup_vpravo: right vstup
vpravo_menší: < vstup_vpravo vstup
              if vpravo_menší vstup_vpravo vstup
```

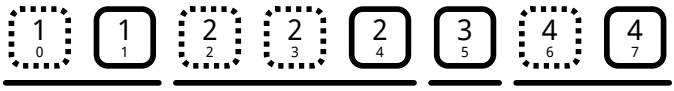
Nejčetnější číslo

Díky tomu, že jsou čísla na vstupu v neklesajícím pořadí, zabírá každé číslo nějaký souvislý interval jader. Stačí nám pro každý z nich nějak spočítat, jak dlouhý je. Poté můžeme snadno najít nejdelší z nich, obdobně, jako jsme hledali nejmenší číslo v předchozí úloze.

Nabízí se, aby každé jádro nějak spočítalo délku intervalu, ve kterém se nachází. To je ale zbytečně těžké, místo toho z každého intervalu vybereme jedno jádro, které

* Také bychom jej mohli přesunout na konec posloupnosti, protože čtení probíhá cyklicky.

spočítáním délky intervalu pověříme. Konkrétně vybereme poslední jádro v intervalu a nazveme jej *zástupcem* intervalu. Zástupce se pozná snadno – jádro napravo od něj má jiný vstup než on.



Intervaly a zástupci. Zástupci mají plný rámeček, ostatní jádra přerušovaný.

Podívejme se na identifikační čísla zástupců. Můžeme si všimnout, že rozdíl ID zástupců dvou sousedních intervalů bude rovný délce pravého intervalu. Všechny zástupce proto přesuneme na začátek posloupnosti, kde si může každý přečíst ID jeho levého souseda a dopočítat délku svého intervalu. Jedinou výjimku tvoří zástupce úplně prvního intervalu, který si musí všimnout, že nemá na levé straně zástupce a spočítat délku svého intervalu jako svoje ID + 1.

Ještě si musíme rozmyslet jeden okrajový případ, a to když jsou všechna čísla na vstupu stejná. V tom případě bude mít každé jádro napravo jádro se stejným vstupem, a proto se nebude považovat za zástupce. Naštěstí je těžké napsat řešení, které v tomto případě nebude fungovat. Pokud prostě napíšeme řešení tak, že vždy vrátí vstup některého z jader, pak v tomto případě zaručeně odpoví správně.

Program:

```
# Zadefinujeme užitečné konstanty
nula:               const 0
minus_jedna:        const -1

# Zjistíme, jestli je jádro zástupcem intervalu (tj. na konci)
vstup:              input 1
vstup_napravo:      right vstup
není_zástupce:      = vstup_napravo vstup

# Všechna ostatní jádra přesuneme na konec řady
                    sort není_zástupce

# Zjistíme ID zástupce předchozího intervalu
id:                 id
předchozí_id:       left id

# Před první interval si domyslíme jádro s ID -1
je_první_interval: left není_zástupce
předchozí_id:       if je_první_interval minus_jedna předchozí_id

# Spočítáme délku intervalu
délka_intervalu:    - id předchozí_id

# Uvažujeme jen výsledky zástupců, ostatní jádra mohou spočítat zvláštní hodnoty
délka_intervalu:    if není_zástupce nula délka_intervalu

# Najdeme maximální délku intervalu, obdobně jako při počítání minima
                    sort délka_intervalu
má_id_0:            != id nula
                    sort má_id_0
délka_vlevo:        left délka_intervalu
```

```
vstup_vlevo:       left vstup
je_vlevo_delší:    > délka_vlevo délka_intervalu

# Místo délky intervalu vracíme jeho hodnotu
                   if je_vlevo_delší vstup_vlevo vstup
```

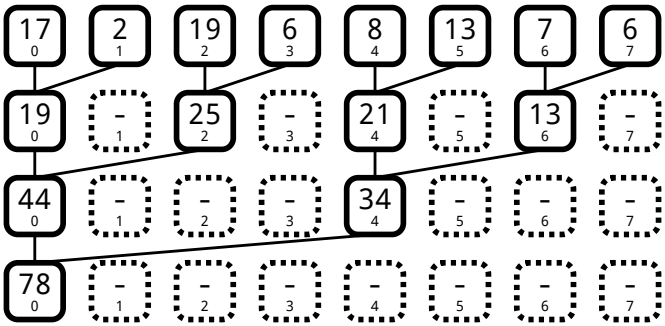
Součet

V této úloze si už bohužel nevystačíme s programem konstantní délky vzhledem k počtu jader. Zároveň ale máme k dispozici méně instrukcí než je jader, takže si nemůžeme dovolit čísla sčítat jedno po druhém. Ukážeme si tedy program, který úlohu vyřeší v čase $\mathcal{O}(\log N)$.

Jádra si nejprve rozdělíme podle jejich ID na lichá a sudá. (První jádro má ID 0, tedy je sudé.) Sudá jádra si ke svému vstupu přičtou vstup jádra napravo od nich. Lichá jádra už dále nebudeme potřebovat, proto je přesuneme na konec posloupnosti. Od této chvíle už nic dělat nebudou. Nakonec si sudá jádra (pomyslně) vydělí ID dvěma.

Tím jsme získali původní problém, jen s polovičním počtem jader. Stačí tedy tento postup opakovat $\log_2 N = 11$-krát. Pak nám zbude jen jedno jádro, ve kterém bude součet všech původních vstupů, a to dokonce to s ID 0.

Schéma výpočtu bude mít tvar binárního stromu:



Strom výpočtu

Program:

```
# Zadefinujeme pomocnou konstantu
dvě:        const 2

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

Předchozí blok devětkrát zkopírujeme.

```
# Nakonec zbudou nesečtená jen dvě jádra, která sečteme
napravo:    right mezisoučet
mezisoučet: + mezisoučet napravo
```