75. ročník Matematické olympiády – 2025/2026

Řešení úloh domácího kola kategorie P

P-I-1 Semafory

Úlohu lze přirozeně interpretovat jako hledání nejkratších cest v grafu (s dalšími omezeními), proto v našem řešení použijeme grafovou terminologii: křižovatky jsou vrcholy grafu a cesty jsou směrové hrany mezi nimi. Než si přečtete řešení, doporučujeme, abyste se seznámili s prohledáváním do šířky (což je základní algoritmus, který můžeme použít, když jsou všechny hrany stejně dlouhé; viz *https://ksp.mff.cuni.cz/encyklopedie/grafy/*) a Dijkstrovým algoritmem (který se hodí, když hrany mají libovolné nezáporné délky; viz *https://ksp.mff.cuni.cz/encyklopedie/halda-a-cesty/*). Naše řešení soutěžního úkolu bude tyto algoritmy vhodně využívat a modifikovat.

Práce se semafory

Začněme tím, že si popíšeme, jak z parametrů semaforu a konkrétního času v konstantním čase určit, která světla svítí a kdy bude svítit další zelená:

Připomeňme si, že semafor má čtyři parametry: dobu zelené a červené (z a c), počáteční stav a čas označující, kdy se semafor příště změní (f). Cyklus semaforu má zjevně délku $z + c$. Dohodněme se, že za začátek cyklu budeme vždy považovat okamžik, kdy se rozsvítí zelená. V rámci každého cyklu svítí zelená z sekund, a poté svítí červená c sekund. Kdy začíná první cyklus konkrétního semaforu?

- Pokud je na začátku (v čase 0) rozsvícena červená, přepne se v čase f na zelenou a tím začne cyklus.
- Pokud je semafor na začátku zelený, přepne se v čase f na červenou, a poté v čase $f + c$ zpět na zelenou, čímž začne cyklus.

Tím jsme určili čas t_0, kdy začíná první cyklus. Další cykly začínají právě v časech, které se liší od t_0 o celočíselné násobky $(r + c)$.

Nyní jsme připraveni na hlavní otázky, které nás zajímají. Představme si, že jsme dorazili k semaforu v (celočíselném) čase t. Kde se nacházíme v jeho cyklu? Podívejme se na hodnotu $t - t_0$. Pokud je to násobek $r + c$, jsme právě na začátku cyklu. Obecně nám pak zbytek této hodnoty $t - t_0$ po dělení $r + c$ říká, kde se nacházíme v cyklu semaforu:

- Pokud $(t - t_0) \bmod (r + c) < r$, zelená světla stále svítí.
- Pro $(t - t_0) \bmod (r + c) = r$ právě naskočila červená a pro ještě větší hodnoty jsme v červeném intervalu.

Pokud svítí zelená, můžeme projet semaforem rovnou. Pokud ne, víme, že zelená nám naskočí v nejbližším následujícím čase t' takovém, že $t' - t_0$ je násobkem $r + c$. Jinými slovy, musíme počkat $r+c-x$ sekund, kde $x = (t-t_0) \bmod (r+c)$ je aktuální zbytek.

Technická poznámka: v některých programovacích jazycích, jako jsou C a C++, je operátor modulo implementován tak, že pro zápornou hodnotu dělence vrací záporný výstup. Jelikož chceme, aby výstup byl vždy v intervalu od 0 do $r + c - 1$, musíme proto dávat pozor, abychom modulo používali pouze na nezáporné vstupy. Toho můžeme dosáhnout například použitím hodnoty $t - t_0 + (r + c)$ namísto hodnoty $t - t_0$, která dává stejný zbytek a je zaručeně kladná.

Stručně o částečných a pomalejších řešeních

Pokud jsou všechny semafory vždy zelené, můžeme jednoduše použít Dijkstrův algoritmus k nalezení nejrychlejší trasy.

Pokud jsou všechny přechodové časy podél hran malé, můžeme graf transformovat na graf, ve kterém lze každou hranu překonat za 1 sekundu. Například hranu délky 3 lze vždy rozdělit na tři jednotkové hrany vložením pomocných vrcholů. Pokud je takto sestavený graf dostatečně malý, můžeme problém vyřešit přímou simulací všech možností najednou: Počínaje časem 0 postupně určujeme po každé sekundě množinu vrcholů, do kterých se v právě tomto čase dokážeme dostat. Jakmile známe tuto množinu pro čas i, sestavíme ji pro čas $i + 1$ tak, že se podíváme na všechny hrany vedoucí z každého vrcholu, ve kterém bychom mohli být, a pro každou z nich určíme, zda ji v daném okamžiku můžeme překročit.

Navíc si můžeme všimnout, že jakmile se poprvé dostaneme do určitého vrcholu, můžeme tam zůstat a čekat tak dlouho, jak chceme, a proto tam můžeme být také kdykoli později. Pro každý vrchol grafu tedy potřebujeme pouze najít nejdřívější čas, kdy tam můžeme dorazit. V grafu s jednotkovými délkami hran to můžeme udělat vhodnou modifikací prohledávání do šířky.

Je možné ještě drobné vylepšení: Můžeme si pamatovat, kterými hranami jsme už z daného vrcholu dokázali odejít. Poté, co se to povedlo pro všechny hrany z tohoto vrcholu, můžeme nadále tento vrchol ignorovat.

Vzorové řešení

Abychom získali efektivní řešení, upravíme Dijkstrův algoritmus tak, aby fungoval pro grafy se semafory bez snížení jeho asymptotické časové složitosti.

Pro každý vrchol si budeme počítat nejdřívější čas, kdy je do něj možné dorazit. V průběhu algoritmu budeme na tuto hodnotu vždy znát horní odhad a ten budeme postupně vylepšovat. Až si budeme jistí, že tento odhad je pro daný vrchol přesný, označíme tento vrchol jako dokončený. Na začátku si tento odhad τ_v pro vrchol v nastavíme na 0, pokud v je vrchol, kde začínáme, a na nekonečno pro všechny ostatní vrcholy; a žádný vrchol zatím není označený jako dokončený.

Od této chvíle algoritmus běží ve smyčce. Každá iterace smyčky vypadá takto:

1. Ze všech vrcholů, které ještě nejsou dokončené, vybereme vrchol v s nejmenším časem.
2. Tento vrchol v označíme za dokončený.
3. Pro každou hranu i vedoucí z vrcholu $a_i = v$ do nějakého jiného vrcholu $b_i = w$ provedeme následující úvahu: Vypočítáme nejbližší čas ξ_v větší

nebo rovný τ_v, kdy bude na příslušném semaforu zelená, a budeme tedy moci vstoupit na tuto hranu. K tomu přidáme čas t_i potřebný k přejetí přes tuto hranu. Tímto způsobem jsme právě objevili nový způsob, jak se dostat do w v čase $\xi_v + t_i$. Pokud je aktuální hodnota τ_w větší, snížíme ji na nově vypočítanou lepší hodnotu.

Stejně jako u klasického Dijkstrovy algoritmu můžeme uložit nedokončené vrcholy do vhodné efektivní datové struktury (uspořádané množiny nebo prioritní fronty) v pořadí dle jejich τ-hodnot. Tím v kroku 1 každého cyklu dokážeme efektivně najít vrchol, který potřebujeme v tomto cyklu zpracovat. V kroku 3 pak musíme upravit záznam v této datové struktuře pro každý vrchol w, jehož hodnotu τ_w změníme.

Důkaz správnosti je také velmi podobný klasickému algoritmu. Tvrdíme, že na konci každé iterace cyklu platí:

- Pro každý dokončený vrchol v je τ_v přesně rovno nejkratšímu času potřebnému k dosažení v, a pro nedokončené vrcholy v je τ_v nejkratším časem potřebným k dosažení v, pokud cestou smíme navštívit pouze dokončené vrcholy.
- Pro každý dokončený vrchol f a každý nedokončený vrchol g platí, že $\tau_f \leq \tau_g$. Z toho mimo jiné vyplývá, že pokud seřadíme všechny vrcholy v pořadí, v jakém byly prohlášeny za dokončené, budou také seřazeny podle jejich výsledné hodnoty τ.

Tato tvrzení můžeme dokázat matematickou indukcí dle počtu iterací cyklu. Na začátku obě tvrzení zjevně platí. Podívejme se nyní na libovolnou iteraci cyklu. Za předpokladu, že výše uvedená tvrzení platí na začátku, ukážeme, že budou platit i na konci.

V kroku 1 vybereme z vrcholů, které ještě nejsou dokončené, vrchol v s nejmenší hodnotou τ_v. Tato hodnota již musí být přesná a nejde dále zlepšit: Z indukčního předpokladu víme, že τ_v je nejmenší čas, za který se můžeme dostat do vrcholu v pouze přes již dokončené vrcholy. Pokud zvolíme jakoukoli jinou cestu, bude také obsahovat některé další nedokončené vrcholy. Označme první z nich jako w. Pak pouze úsek této cesty ze začátečního vrcholu do w trvá (alespoň) τ_w, tj. alespoň stejně dlouho jako aktuální τ_v (jinak řečeno, do v se nemůžeme dostat rychleji přes jiný nedokončený vrchol, protože cesta do tohoto nedokončeného vrcholu sama o sobě zabere alespoň čas τ_v).

Proto v kroku 2 skutečně smíme prohlásit nalezený vrchol v za dokončený. Tím se změní množina dokončených vrcholů, což znamená, že hodnoty τ pro vrcholy, které ještě nejsou dokončené, se mohou také změnit. Konkrétněji: nyní do nich můžeme dorazit také nějakou cestou procházející přes vrchol v.

Zřejmě ale stačí brát do úvahy cesty, v nichž je v předposlední vrchol. Pokud bychom totiž pokračovali z v do jiného dokončeného vrcholu x, dorazili bychom tam za dobu delší než τ_v; nicméně vrchol x byl dokončen před vrcholem v, a proto

$\tau_x \leq \tau_v$. Jinými slovy, do vrcholu x se můžeme dostat rychleji bez procházení přes vrchol v.

Pro každý nedokončený vrchol w tedy stačí zvážit cesty, v nichž se nejprve dostaneme do v přes ostatní dokončené vrcholy, načež jdeme přímo po hraně z v do w.

Zde zdůrazněme vlastnost této úlohy, díky které (upravený) Dijkstrův algoritmus stále funguje: *nikdy se nevyplatí dorazit do vrcholu či překročit hranu záměrně později*. Pokud dorazíme do vrcholu dříve, nic tím totiž nezkazíme – nejhorší, co se může stát, je, že budeme muset čekat déle. Obdobně pokud můžeme vstoupit na hranu, nemá smysl čekat a vstoupit na ni později – nikdy nemůže nastat situace, kdy bychom na hranu vstoupili později, ale opustili ji dříve.

Pro každou hranu z právě dokončeného vrcholu v do jiného, dosud nedokončeného vrcholu w, má proto smysl zvážit pouze jednu možnost: dostaneme se do v co nejrychleji, a pak použijeme hranu z v do w, jakmile to bude možné. To ukazuje, že po dokončení kroku 3 budou hodnoty τ splňovat popsané vlastnosti.

Při implementaci s efektivní datovou strukturou má toto řešení časovou složitost $O((n+m)\log n)$. V průběhu algoritmu totiž:

- Každý vrchol vybereme pro zpracování (a dokončení) z této datové struktury nejvýše jednou.
- Pro každou hranu vw nejvýše jednou zkontrolujeme, zda optimální cesta do v následovaná hranou do w zlepšuje τ_w, a pokud ano, upravíme záznam pro w v naší datové struktuře.

Celkově tedy provedeme nejvýše $n+m$ operací s naší datovou strukturou a každá z nich nám zabere čas $O(\log n)$.

Níže je uveden příklad implementace v jazyce Python s použitím implementace prioritní fronty binární haldou.

Program (Python 3):

```
from heapq import heappush, heappop

class Hrana:
    def __init__(self, **kwargs):
        for key, value in kwargs.items():
            setattr(self, key, value)

    def cas_v_cili(self, cas_na_zacatku):
        # Kdy nejdříve můžeme dojít na konec hrany, když víme,
        # kdy jsme dorazili na její začátek?
        zbytek = (cas_na_zacatku + self.offset) % (self.zelena + self.cervena)
        if zbytek < self.zelena:
            return cas_na_zacatku + self.delka
        else:
            return cas_na_zacatku - zbytek + self.zelena + self.cervena + self.delka

# Načteme vstup
n, m = [ int(_) for _ in input().split() ]
G = [ [] for _ in range(n) ]
for mm in range(m):
```

```
    tokens = input().split()
    start, cil, delka, zelena, cervena = [ int(_) for _ in tokens[:5] ]
    start, cil = start-1, cil-1
    if tokens[5] == 'Z':
        offset = zelena - int(tokens[6])
    else:
        offset = zelena + cervena - int(tokens[6])
    G[start].append(
        Hrana(cil=cil, delka=delka,
              zelena=zelena,
              cervena=cervena, offset=offset))

# Inicializujeme Dijkstrův altoritmus
dist = [ 10**18 for _ in range(n) ]
dist[0] = 0
Q = []
heappush( Q, (0,0) )

# V cyklu zpracováváme vrcholy
while len(Q) > 0:
    d, kde = heappop(Q)
    # V haldě neměníme hodnoty, místo toho do ní při zlepšení hodnoty tau pro daný
    # vrchol v přidáme novou dvojici (tau, v).  Tím nám v haldě mohou zbývat
    # neaktuální dvojice se starým suboptimálním časem, ty prostě ignorujeme.
    if dist[kde] < d: continue
    # Projdeme všechny hrany z tohoto vrcholu a přepočítáme časy pro sousední vrcholy
    for hrana in G[kde]:
        kam, kdy = hrana.cil, hrana.cas_v_cili(d)
        if kdy < dist[kam]:
            dist[kam] = kdy
            heappush( Q, (kdy,kam) )

if dist[n-1] == 10**18:
    print(-1)
else:
    print(dist[n-1])
```

P-I-2 Rozbitý robot

Vstup si předzpracujeme tak, že každému číslu d_i přiřadíme kladné znaménko, pokud se pohybujeme na východ, a záporné znaménko, pokud se pohybujeme na západ. Ve všech níže uvedených řešeních používáme takto upravené hodnoty d_i.

Řešení za 2 body

Pokud $k = 0$, robot nedělá chyby. Pro každý p-prvkový úsek příkazů tedy stačí zjistit, jak daleko se robot posune. Po naší modifikaci to odpovídá nalezení součtu p po sobě jdoucích hodnot posloupnosti d a následnému výpočtu jeho absolutní hodnoty. To můžeme snadno provést v čase lineárním ve velikosti vstupu, tj. v čase $O(n)$, například jedním z následujících postupů.

Jedno možné řešení začíná vypočtením součtu první možné sekce příkazů, tj. $d_1 + \cdots + d_p$. Poté k této hodnotě přičteme d_{p+1} a odečteme d_1, což nám dá součet $d_2 + \cdots + d_{p+1}$, tj. součet druhého segmentu, který nás zajímá, a tak dále.

Jiná možnost je si předpočítat prefixové součty posloupnosti $d_1, d_2, \ldots$, s jejichž

použitím pak dokážeme určit součet libovolného souvislého úseku v konstantním čase.

Řešení za 6 bodů

Pro $n \leq 1000$ si můžeme dovolit projít každý úsek délky p zvlášť, zvládneme-li ho zpracovat rozumně efektivně. Stačí tedy vyřešit jednodušší problém: jak daleko můžeme robota dostat, pokud máme posloupnost p příkazů, které musí všechny provést? Stále platí, že můžeme obrátit nejvýše k z nich.

Chceme-li dostat robota co nejdále od jeho výchozí pozice, máme dvě symetrické možnosti: buď ho dostaneme co nejdále na východ, nebo co nejdále na západ. Na řešení obou zjevně můžeme použít stejný algoritmus, před řešením druhé varianty stačí všechny hodnoty d znegovat. Podívejme se tedy na řešení první možnosti.

V této možnosti máme posloupnost hodnot $d_1, \ldots, d_p$, můžeme změnit znaménko maximálně k jejích členů a chceme na konci získat součet s největší (kladnou) hodnotou. Zjevně se nevyplatí měnit kladná čísla na záporná (tím by se součet snížil), a máme-li na výběr více než k záporných čísel, musíme na kladná změnit ta s co největší absolutní hodnotou. Stačí tedy seřadit čísla v posloupnosti $d_1, \ldots, d_p$ podle velikosti, a poté změnit znaménko prvních (nejvýše) k z nich z záporného na kladné.

Toto řešení zpracovává každý z $n - p + 1 = O(n)$ segmentů samostatně. Posloupnost délky p zvládneme setřídit v čase $O(p \log p)$ a zbytek postupu pro každý úsek nám zabere pouze čas lineární v p, takže celkem dostáváme řešení s časovou složitostí $O(np \log p)$. Poznamenejme, že toto řešení jde drobně vylepšit a odstranit logaritmus z časové složitosti tak, že místo třídění použijeme algoritmus na nalezení k-tého nejmenšího prvku s lineární časovou složitostí.

Vzorové řešení

Nyní zkombinujeme myšlenky z obou výše uvedených řešení. Z prvního řešení vezmeme myšlenku tzv. *posuvného okna* – budeme se postupně pohybovat po posloupnosti a průběžně počítat optimální řešení pro aktuální úsek vhodnou úpravou řešení pro ten předchozí. Z 6-bodového řešení pak použijeme pozorování, že v každém úseku stačí měnit znaménko k nejmenších prvků (opět stačí řešit pouze variantu, kdy se snažíme dostat robota co nejdál na východ, a poté stejné řešení pustit na znegovanou posloupnost).

Co přesně potřebujeme vědět o aktuálně zpracovávaném úseku, abychom mohli určit optimální řešení?

- Jaký by byl součet s tohoto úseku, kdybychom nic neměnili?
- Jaká je hodnota z součtu k nejmenších záporných prvků (nebo součtu všech záporných prvků, pokud je jich méně) v tomto úseku?

Jakmile známe tyto dvě hodnoty, odpověď, kterou hledáme pro tuto sekci, je rovna $s - 2z$. Například pokud změníme pohyb o -3 na pohyb o $+3$, naše konečná pozice se posune o 6 doprava.

Z prvního řešení již víme, jak si udržovat hodnotu s, tj. jak ji přepočítat, když se posuneme na další úsek. Abychom uměli přepočítat i hodnotu z, uložíme si hodnoty v aktuálním úseku do dvou uspořádaných množin. Množina A bude obsahovat

prvky, které nás zajímají, tedy k nejmenších záporných čísel v aktuálním úseku, nebo všechna záporná čísla, pokud je jich méně než k. Množina B bude obsahovat všechny ostatní prvky. Kromě toho si také budeme separátně pamatovat hodnotu z, tj. součet množiny A. Hodnotu z upravíme pokaždé, když změníme množinu A. Například pokud do množiny A přidáme nový prvek, přičteme jeho hodnotu k zapamatované sumě z.

Jaké změny musíme provést, když dokončíme zpracování jednoho úseku vstupu a chceme přejít k následujícímu?

- Odstraníme hodnotu d_i, která právě opustila náš úsek, z množiny A či B, ve které se nachází (pokud se nachází v obou, tj. hodnota d_i se v úseku několikrát opakuje, je jedno, ze které ji odstraníme).
- Pokud jsme odstranili hodnotu d_i z A a nejmenší prvek v B je záporný, přesuneme jej z B do A, aby v A zůstalo stále k nejmenších záporných prvků.
- Hodnotu d_{i+p}, která byla právě přidána do našeho úseku, přidáme do jedné z množin: pokud je záporná, do A, pokud ne, do B.
- Pokud má A nyní příliš mnoho prvků (přesně $k+1$), přesuneme největší z nich do B.

Při implementaci uspořádaných množin A a B efektivní datovou strukturou (například nějakou variantou vyvažovaného vyhledávacího stromu) dokážeme každou z těchto operací provést v čase logaritmickém v počtu prvků těchto množin, tedy nejvýše $O(\log p)$. Celkově tedy zpracujeme každý z $O(n)$ úseků v tomto čase, což nám dává celkovou časovou složitost $O(n \log p)$.

Program (C++):

```
#include <algorithm>
#include <iostream>
#include <set>
#include <vector>
using namespace std;

int n, p, k;
vector<long long> D;
long long s, z;
multiset<long long> A, B;

// Přidání nové hodnoty x do aktuálního úseku
void add(long long x) {
    // Zvětšíme součet úseku
    s += x;
    // Je-li hodnota nezáporná, vložíme ji do B
    if (x >= 0) { B.insert(x); return; }
    // Zápornou hodnotu vložíme do množiny A a zvětšíme její součet
    z += x;
    A.insert(x);
    // Je-li v A k+1 prvků, největší ("nejméně záporný") z nich přesuneme do B
    if (int(A.size()) > k) {
        long long biggest = *prev(A.end());
```

```
        A.erase(prev(A.end()));
        z -= biggest;
        B.insert(biggest);
    }
}

// Odstranění hodnoty x z aktuálního úseku
void remove(long long x) {
    // Zmenšíme součet úseku
    s -= x;
    // Je-li x v B, stačí ho z B smazat
    auto it = B.find(x);
    if (it != B.end()) { B.erase(it); return; }
    // Jinak je x v A, odstraníme ho tedy a aktualizujeme součet A
    z -= x;
    A.erase(A.find(x));
    // V A je nyní méně než k prvků
    // Je-li v B nějaký záporný prvek, přesuneme nejmenší ("nejzápornější")
    // z nich do A
    if (B.empty()) return;
    long long smallest = *B.begin();
    if (smallest >= 0) return;
    z += smallest;
    A.insert(smallest);
    B.erase(B.begin());
}

// Najde optimální řešení směrem na východ pro posloupnost D[]
long long solve() {
    if (p == 0) return 0;
    s = 0;
    z = 0;
    A.clear();
    B.clear();
    // Do aktuálního úseku dáme prvních p prvků, a poté se podíváme
    // na nejlepší řešení pro tento úsek
    for (int i=0; i<p; ++i) add(D[i]);
    long long answer = s - 2*z;
    // Dále úsek postupně posunujeme doprava a kontrolujeme,
    // zda řešení pro aktuální úsek není lepší
    for (int x=1; x+p<=n; ++x) {
        add(D[x+p-1]);
        remove(D[x-1]);
        answer = max(answer, s - 2*z);
    }
    return answer;
}

int main() {
    // Načteme vstup a do D[] uložíme hodnoty se znaménky dle směru
    cin >> n >> p >> k;
    D.resize(n);
    for (int i=0; i<n; ++i) {
        string smer;
        cin >> smer >> D[i];
        if (smer == "Z") D[i] *= -1;
```

```
    }
    // Dvakrát pustíme solve(): jednou na původní posloupnost, jednou na znegovanou
    long long answer = solve();
    for (auto &d : D) d = -d;
    answer = max(answer, solve());
    cout << answer << endl;
}
```

Poznámka na závěr: V jazycích, které ve své standardní knihovně nemají efektivní implementaci uspořádané (multi)množiny, je možné pro reprezentaci množiny B také použít haldu (viz *https://ksp.mff.cuni.cz/encyklopedie/halda-a-cesty/*), která je jednodušší na implementaci. Množinu A si nemusíme udržovat vůbec, stačí si udržovat její velikost a pro každý prvek aktuálního úseku si pamatovat, zda patří do A nebo do B. S použitím haldy dokážeme v čase $O(\log p)$ najít a odstranit nejmenší prvek z B, a vložit nový prvek je také možné se stejnou časovou složitostí. Musíme také umět z B odebrat prvek d_i, což lze také udělat v čase $O(\log p)$, pokud si navíc pamatujeme ukazatel na pozici, kde se d_i v haldě aktuálně nachází.

Popíšeme ještě jednou variantu, kterou lze využít i v případě, kdy pro haldu používáme nějakou knihovní implementaci, která operaci odebrání libovolného prvku nepodporuje. Tehdy budeme mazat prvky z B líně: Prvek d_i z B neodebíráme, pouze si vedle označíme, že už do B nepatří. Pokud na takový prvek narazíme při odebírání minima z B, ignorujeme ho a operaci opakujeme.

P-I-3 Sušenky

Podúlohy A a B: konkrétní postupnosti sušenek

Emička může sníst první posloupnost sušenek například takto:

C̲CCCCCC
-PC̲CCCC
-C̲-PCCC
---PC̲CC
---C̲-PC
-----PC̲
-----C̲-

V tomto postupu vždy jíme nejlevější sušenku, která je čokoládou nahoru. Po prvním kroku získáme konfiguraci, která se pak každé dva kroky opakuje, jen máme o dvě sušenky méně. Existují i jiné možnosti, například tato:

CCCCC̲CC
CCC̲P-PC
C̲P-C-PC
-C-C-PC̲
-C-C-C-

Při tomto postupu nejprve sníme sušenky na sudých pozicích (počítáno od nuly) v libovolném pořadí, čímž každou ze zbývajících sušenek dvakrát otočíme. Poté můžeme zbývající sušenky sníst v libovolném pořadí, protože žádné dvě nesousedí.

Pokud jsou sušenky na začátku v konfiguraci `CPPPPPC`, Emička je nemůže všechny sníst. Proč? Všimněte si, že na začátku má pouze dvě možnosti, sníst jednu ze sušenek na koncích řady. Na této volbě navíc ve skutečnosti vůbec nezáleží; ať sní kteroukoli, výsledná posloupnost nesnědených sušenek bude `CPPPPC`. To nás přivádí do podobné situace, ale s 6 sušenkami místo 7. Můžeme snadno zobecnit, že přesně to samé se stane v každém z následujících čtyř kroků. Po nich nutně dospějeme k posloupnosti `CC`. V dalším kroku nám nutně zůstane jedna sušenka otočená čokoládou dolů a Emička ji podle svých pravidel nebude moci sníst. V této podúloze tedy Emička nemůže sníst všechny sušenky.

Podúloha C: ověření, zda lze danou posloupnost sníst

V této podúloze jsme chtěli najít (snadno ověřitelnou) podmínku, která nám řekne, zda lze posloupnost sníst, nebo ne.

Pokud jste si s úkolem trochu pohráli, pravděpodobně jste přišli na to, co se stane v případě malého počtu sušenek: Pokud máme pouze jednu sušenku, můžeme ji sníst, pokud je čokoládovou stranou nahoru. Pokud máme dvě sousedící sušenky, můžeme je sníst pouze v případě, že jedna z nich je čokoládovou stranu nahoru. A co tři sušenky? Zde to funguje, pokud začneme s jednou nebo třemi sušenkami čokoládovou stranou nahoru. Z toho můžeme uhodnout, že je-li v posloupnosti lichý počet sušenek čokoládovou stranou nahoru, dokážeme celou posloupnost sníst, a jinak to nelze. Samozřejmě nyní musíme toto tvrzení řádně dokázat.

Uděláme to ve dvou krocích: Nejprve ukážeme, že můžeme sníst každou posloupnost, ve které je lichý počet sušenek čokoládou nahoru. Všimněme si, že jelikož otáčíme pouze sušenky, které sousedí v *původní* posloupnosti, jakmile je posloupnost rozdělena na samostatné části, konzumace sušenek z jedné z nich již nemá žádný mít vliv na ostatní části. Pokud tedy máme na stole posloupnost sušenek, z nichž některé už byly snědeny, Emička je může dojíst pouze v případě, že dokáže dojíst každý ze souvislých úseků oddělených snědenými sušenkami samostatně.

Nyní uvažme libovolný souvislý úsek, ve kterém je *lichý* počet sušenek čokoládovou stranou nahoru. Co se stane, když sníme *nejlevější* z těchto sušenek?

- Pokud je snědená sušenka úplně první v řadě, nic se nestane nalevo od ní. Podobně, pokud je to poslední sušenka v řadě, nic se nestane s těmi napravo od ní.
- Pokud jsou nějaké sušenky nalevo od ní, všechny jsou čokoládou dole a nejpravější z nich otočíme čokoládovou stranou nahoru. Tím vznikne kratší úsek, ve kterém je přesně jedna sušenka (tedy lichý počet) čokoládovou stranou nahoru.
- Pokud jsou nějaké sušenky vpravo od právě snědené sušenky, je mezi nimi sudý počet sušenek čokoládovou stranou nahoru a otočíme tu nejlevější z nich. Tím se počet sušenek čokoládovou stranou nahoru změní (zvětší či

zmenší) právě o jedna, bude jich tedy napravo od té snědené zbývat lichý počet.

Pokud vždy sníme nejlevější sušenku, která je čokoládou nahoru, v každém souvislém úseku odděleném snědenými sušenkami proto určitě vždy bude lichý (a tedy nenulový) počet sušenek čokoládovou stranou nahoru. Dokud nesníme vše, můžeme tedy vždy sníst další sušenku, a protože celkový počet nesnědených sušenek neustále klesá, celý proces musí skončit tím, že sníme všechno.

Stejný důkaz můžeme říct formálněji takto: Matematickou indukcí podle n dokazujeme tvrzení, že každou souvislou posloupnost n sušenek, ve které je lichý počet sušenek čokoládovou stranou nahoru, můžeme sníst pomocí postupu „vždy sněz nejlevější sušenku, která je čokoládou nahoru“. Při dokazování indukčního kroku provedeme výše uvedenou analýzu toho, co se děje v prvním kroku řešení, a poté z indukčního předpokladu vidíme, že touto strategií sníme obě vzniklé kratší souvislé posloupnosti sušenek.

Zbývá nám tedy uvažovat případ, kdy máme sudý počet sušenek čokoládovou stranou nahoru. Nyní musíme ukázat, že *žádný* postup nebude fungovat. K tomu stačí ukázat následující tvrzení: Ať už Emička sní libovolnou sušenku, jeden ze vzniklých neprázdných souvislých úseků sušenek bude obsahovat sudý počet sušenek čokoládovou stranou nahoru. Časem tedy nutně vytvoří neprázdný souvislý úsek s *nula* sušenkami čokoládou nahoru. (Kdykoliv Emička sní sušenku tak, že vzniknou dva nové neprázdné souvislé úseky, vybereme ten z nich, který má sudý počet sušenek čokoládou nahoru, a druhý ignorujeme. Toto opakujeme, dokud nedojdou sušenky čokoládovou stranou nahoru.) Tento úsek pak Emička nemůže sníst.

A proč výše uvedené tvrzení platí? Představme si, že máme souvislou posloupnost s $2k$ sušenkami s čokoládou nahoře a Emička se rozhodla sníst i-tou z nich. Před touto sušenkou je $i-1$ sušenek čokoládou nahoru a za ní je $2k-i$ takových sušenek. Jelikož součet těchto dvou hodnot je $2k-1$, což je lichý počet, přesně jedna z těchto hodnot je lichá. Odpovídající úsek je jistě neprázdný, jelikož obsahuje lichý (a tedy nenulový) počet sušenek čokoládou nahoru. Po snědení i-té sušenky Emička v tomto úseku otočí právě jednu sušenku, čímž změní počet těch čokoládou nahoru právě o jedna. Tento souvislý úsek tedy poté bude obsahovat sudý počet sušenek čokoládovou stranou nahoru.

Podúloha D: Všechny vítězné tahy

K vyřešení poslední podúlohy navážeme na tu předchozí. Již jsme dokázali, že Emička může sníst souvislou neprázdnou posloupnost sušenek právě když v ní je lichý počet sušenek čokoládovou stranou nahoru; a že pokud se posloupnost skládá z několika souvislých úseků oddělených snědenými sušenkami, umíme celou posloupnost sníst právě tehdy, když lze sníst každý z těchto úseků zvlášť (tedy když každý z nich obsahuje lichý počet sušenek čokoládou nahoru).

Algoritmus bude tedy vypadat následovně: Rozdělíme vstup na souvislé úseky sušenek a vypočítáme počet možných prvních snědených sušenek pro každý z nich zvlášť. Pokud je odpověď pro některý úsek 0, jako celkový výsledek vypíšeme 0, jinak vypíšeme součet všech odpovědí pro jednotlivé úseky.

Jak vypočítáme odpověď pro daný souvislý úsek? Jednou z možností je zkontrolovat samostatně pro každou možnou sušenku x otočenou čokoládovou stranou nahoru, co se stane, když ji sníme a otočíme její přímé sousedy. Musíme zjistit, zda úsek vlevo a úsek vpravo od sušenky x bude poté mít sudý nebo lichý počet sušenek čokoládou nahoru: Sušenku x můžeme sníst jako první pouze v případě, že oba tyto úseky jsou buď prázdné nebo obsahují lichý počet sušenek čokoládovou stranou nahoru. Toto řešení můžeme implementovat v lineárním čase, například pomocí techniky prefixových součtů.

Implementaci si ale můžeme zjednodušit, pokud se ještě trochu zamyslíme. Co se stane, když v daném souvislém úseku je $2k+1$ sušenek čokoládou nahoru a sníme i-tou z nich? V řešení části C jsme ukázali, že můžeme určitě sníst tu nejlevější ($i = 1$) a díky symetrii také tu nejpravější ($i = 2k+1$). Podívejme se nyní na ostatní i. Vlevo od vybrané sušenky je $i - 1$ a vpravo je $2k + 1 - i$ sušenek čokoládovou stranou nahoru. Protože $1 < i < 2k + 1$, oba tyto úseky jsou neprázdné. Když sníme i-tou sušenku, otočíme jednu sušenku v každém z nich, čímž změníme paritu počtu sušenek čokoládou nahoru. Pro lichá i tedy oba nové kratší úseky budou mít lichý počet sušenek čokoládou nahoru (a můžeme je tedy sníst), zatímco pro sudé i bude v obou kratších úsecích sudý počet sušenek čokoládou nahoře (a žádný z nich nemůžeme sníst).

V úseku s $2k + 1$ sušenkami čokoládou nahoru tedy Emička může začít tím, že sní první, třetí, pátou, ..., až $(2k + 1)$ z těchto sušenek. Má tedy přesně $k + 1$ možností, ze kterých si může vybrat.

Stačí nám tedy pro každý souvislý úsek spočítat, kolik obsahuje sušenek čokoládou nahoře. Je-li jich sudý počet, vypíšeme výsledek 0 a skončíme; je-li lichý, k průběžnému výsledku přičteme polovinu tohoto počtu zaokrouhlenou nahoru. Implementace tohoto řešení v Pythonu může vypadat například takto:

Program (Python 3):

```
n = int(input())

susenky = input()
useky = susenky.split('-')

res = 0

for usek in useky:
    if len(usek) > 0:
        c = usek.count('C')
        if c % 2 == 0:
            print(0)
            exit()
        res += 1 + (c // 2)

print(res)
```

P-I-4 Řešič to vyřeší

Podúlohy A–E: investice

Pro každý projekt se musíme se rozhodnout, zda ho podpoříme, nebo ne. To lze přirozeně modelovat pomocí binární proměnné x_i pro každý projekt i, která udává, zda jej podpoříme, nebo ne: 0 (nepravda) znamená ne, 1 (pravda) znamená ano.

Pokud projekt i podpoříme, dáme mu s_i peněz a později obdržíme výnos v_i, takže na konci budeme mít o $(v_i - s_i)$ peněz více než na začátku. Náš konečný zůstatek na účtu lze proto vyjádřit jako jeho počáteční hodnota (e) plus součet všech výrazů ve tvaru $(v_i - s_i)x_i$; nepodporované projekty nemění zůstatek na účtu (vynásobíme nulou), podporované projekty jej mění o svůj čistý zisk. Tuto hodnotu chceme maximalizovat.

V podúlohách A a D nemáme žádné konflikty ani závislosti, musíme tedy dodržet pouze jediné omezení: musíme mít dostatek peněz na podporu všech vybraných projektů. Jinými slovy, součet $s_i x_i$ pro všechna i nesmí překročit e.

Jak modelovat konflikty? Podmínku „z projektů s a t můžete podpořit nanejvýš jeden“ lze matematicky zapsat takto: $x_s + x_t \leq 1$. Jedná se o přímé omezení v povolené formě, takže jej prostě přidáme do našeho ILP a máme hotovo. Poznamenejme, že obecnější konflikty zahrnující více než dva projekty by bylo možné modelovat přesně stejným způsobem.

A jak modelovat závislosti? Pokud projekt u závisí na projektu v, nemůže zároveň platit $x_u = 1$ a $x_v = 0$. Ekvivalentně: hodnota x_u musí být vždy menší nebo rovna hodnotě x_v. A $x_u \leq x_v$ je opět podmínka v povolené formě, takže jsme hotovi.

Pro příklad ze zadání dostáváme následující ILP ve formátu pro `lp_solve`:

Program (ILP):

```
max: 100000 + 50000 * x0 + 45000 * x1 + 6000 * x2 + 860000 * x3;

50000 * x0 + 50000 * x1 + 20000 * x2 + 40000 * x3 <= 100000;
x0 + x1 <= 1;
x3 <= x2;
x2 <= x1;

bin  x0, x1, x2, x3;
```

Optimální řešení pro vstup D má zisk 242 559 804 a pro vstup E je optimální zisk 578 053 900. Následující program v jazyce Python řeší vstupy pomocí knihovny PuLP.

Program (Python 3):

```
import pulp

def hodnota(vyraz): return int(round(pulp.value(vyraz)))

e, n = [ int(_) for _ in input().split() ]
S = [ int(_) for _ in input().split() ]
V = [ int(_) for _ in input().split() ]
k = int(input())
C = [ [ int(_)-1 for _ in input().split() ] for __ in range(k) ]
```

```
# -1 neboť interně číslujeme od 0
z = int(input())
D = [ [ int(_)-1 for _ in input().split() ] for __ in range(z) ]

problem = pulp.LpProblem('Investice', pulp.LpMaximize)
project = [ pulp.LpVariable(f'project{i}', cat='Binary') for i in range(n) ]

problem += e + sum( project[i] * (V[i] - S[i]) for i in range(n) )
problem += sum( project[i] * S[i] for i in range(n) ) <= e
for conflict in C:
    problem += sum( project[i] for i in conflict ) <= 1
for dependency in D:
    problem += project[dependency[0]] <= project[dependency[1]]

status = problem.solve()
assert pulp.LpStatus[status] == 'Optimal'
print(hodnota(problem.objective))
for x in project: print(x.name, '=', hodnota(x))
```

Podúlohy F–J: chovatelé prasat

Tentokrát použijeme pro modelování celočíselné proměnné. Stačí ty, které popisují dopravu: Pro každý druh plodů t, každou sýpku i a každého farmáře j budeme mít proměnnou $x_{t,i,j}$, která představuje počet kilogramů tohoto druhu, který farmář j obdrží a zaplatí v balíku zaslaném ze sýpky i.

Omezení odpovídající zadání nyní vypadají následovně:

- Pro každou sýpku je součet všeho, co z ní odchází, menší nebo roven její kapacitě. Je zřejmé, že nemá smysl sbírat více, než můžeme odeslat.
- Pro každou dvojici (sýpka, farmář) je součet žaludů a bukvic odeslaných v balíku menší nebo roven w.
- Pro každou dvojici (druh plodů, farmář) je celkové množství dodaných plodů tohoto druhu nejvýše rovno odpovídající hodnotě g, tj. limitu určujícímu, kolik je tento farmář ochoten koupit.

A cíl, tedy náš zisk? Máme konstantní ztrátu 5000 za sklizeň, tedy „zisk“ -5000. Balík odeslaný ze sýpky i farmáři j má hmotnost $(x_{1,i,j} + x_{2,i,j})$, takže za něj zaplatíme $\ell_{i,j} \cdot (x_{1,i,j} + x_{2,i,j})$. Za prodej plodů typu t farmáři j naopak dostaneme částku $h_{j,t} \cdot (x_{t,1,j} + \cdots + x_{t,i,j})$. Sečtením všech těchto hodnot získáme jeden velký lineární výraz, jehož hodnotu chceme maximalizovat.

Generovaný ILP ve formátu pro `lp_solve` pro příklad ze zadání tedy vypadá takto.

Program (ILP):

```
max: -5000
     -  9 * x_1_1_1 -  9 * x_2_1_1 -  15 * x_1_1_2 - 15 * x_2_1_2 - 23 * x_1_1_3 - 15 * x_2_1_3
     - 11 * x_1_2_1 - 11 * x_2_2_1 -  28 * x_1_2_2 - 28 * x_2_2_2 - 12 * x_1_2_3 - 12 * x_2_2_3
     + 10 * x_1_1_1 + 10 * x_2_1_1 + 100 * x_1_1_2 + 30 * x_2_1_2 + 50 * x_1_1_3 + 60 * x_2_1_3
     + 10 * x_1_2_1 + 10 * x_2_2_1 + 100 * x_1_2_2 + 30 * x_2_2_2 + 50 * x_1_2_3 + 60 * x_2_2_3;

x_1_1_1 + x_2_1_1 + x_1_1_2 + x_2_1_2 + x_1_1_3 + x_2_1_3 <= 100;
x_1_2_1 + x_2_2_1 + x_1_2_2 + x_2_2_2 + x_1_2_3 + x_2_2_3 <= 200;

x_1_1_1 + x_2_1_1 <= 80;  x_1_1_2 + x_2_1_2 <= 80;  x_1_1_3 + x_2_1_3 <= 80;
x_1_2_1 + x_2_2_1 <= 80;  x_1_2_2 + x_2_2_2 <= 80;  x_1_2_3 + x_2_2_3 <= 80;
```

```
x_1_1_1 + x_1_2_1 <= 1000;  x_2_1_1 + x_2_2_1 <= 1000;
x_1_1_2 + x_1_2_2 <=  120;  x_2_1_2 + x_2_2_2 <=   70;
x_1_1_3 + x_1_2_3 <=   20;  x_2_1_3 + x_2_2_3 <=   30;

int x_1_1_1, x_2_1_1, x_1_1_2, x_2_1_2, x_1_1_3, x_2_1_3,
    x_1_2_1, x_2_2_1, x_1_2_2, x_2_2_2, x_1_2_3, x_2_2_3;
```

Optimální řešení pro H je 2 842 033, pro I je 1 219 206 a pro J je 2 287 290. Následující program v jazyce Python řeší vstupy pomocí knihovny PuLP.

Program (Python 3):

```
import pulp

def hodnota(vyraz): return int(round(pulp.value(vyraz)))

s, c = [ int(_) for _ in input().split() ]
K = [ int(_) for _ in input().split() ]
G, H = [], []
for i in range(c):
    g1, g2, h1, h2 = [ int(_) for _ in input().split() ]
    G.append((g1, g2))
    H.append((h1, h2))
w = int( input() )
L = [ [ int(_) for _ in input().split() ] for __ in range(s) ]

problem = pulp.LpProblem('Chovatele', pulp.LpMaximize)

package_sizes = [
  [ [ pulp.LpVariable(f'x_{t}_{i}_{j}', cat='Integer') for j in range(c) ]
        for i in range(s) ] for t in range(2)
]

cil  = sum( H[j][t] * package_sizes[t][i][j] for t in range(2)
            for i in range(s) for j in range(c) )
cil -= sum( L[i][j] * ( package_sizes[0][i][j] + package_sizes[1][i][j] )
            for i in range(s) for j in range(c) )
cil -= 5000
problem += cil

for t in range(2):
    for i in range(s):
        for j in range(c):
            problem += package_sizes[t][i][j] >= 0

for i in range(s):
    problem += sum( package_sizes[t][i][j] for t in range(2)
                                           for j in range(c) ) <= K[i]

for i in range(s):
    for j in range(c):
        problem += package_sizes[0][i][j] + package_sizes[1][i][j] <= w

for t in range(2):
    for j in range(c):
        problem += sum( package_sizes[t][i][j] for i in range(s) ) <= G[j][t]

status = problem.solve()
assert pulp.LpStatus[status] == 'Optimal'
print(hodnota(problem.objective))
```

Poznámka na závěr: Úlohu o chovatelích je možné modelovat a efektivně řešit i pomocí jiného formalismu, jako nalezení největšího toku ve vhodně konstruované síti nebo nejtěžšího párování ve vhodně konstruovaném váženém bipartitním grafu. Složitější problémy podobného typu však obecně tuto vlastnost nemají – optimalizační problémy, které zahrnují více než jeden typ zboží a v nichž tyto typy nejsou nezávislé, jsou obecně velmi těžké.