A-I-1 Semafory

Toto je zjavne úloha o najkratších cestách v grafe, v riešení budeme preto používať grafovú terminológiu: križovatky sú vrcholy grafu a cesty sú orientované hrany medzi nimi.
Pred čítaním jej riešenia sa oboznám s prehľadávaním do šírky https://www.ksp.sk/kucharka/bfs/ (čo je základný algoritmus, ktorý môžeme použiť, keď majú všetky hrany rovnakú, resp. jednotkovú dĺžku) a s Dijkstrovym algoritmom https://www.ksp.sk/kucharka/dijkstra/ (ktorý môžeme použiť, keď majú hrany ľubovoľné nezáporné dĺžky). Naše riešenie súťažnej úlohy bude vhodne využívať a upravovať tieto algoritmy.

Práca so semaforom

Z parametrov semaforu a konkrétneho času vieme v konštantnom čase spočítať, aké svetlo na ňom svieti a tiež to, kedy bude najbližšie zelená.
Pripomeňme si, že semafor má štyri parametre: časy zelenej a červenej (z a c), počiatočný stav a čas udávajúci, kedy sa semafor najbližšie prepne (f).
Cyklus semaforu má zjavne dĺžku $z + c$. Dohodneme sa, že za začiatok cyklu budeme považovať vždy okamih, kedy naskočí zelená. V rámci každého cyklu najskôr z sekúnd svieti zelená a potom c sekúnd červená.
Kedy začínajú cykly konkrétneho semaforu? Ak na ňom na začiatku (v čase 0) svieti červená, tá sa v čase f prepne na zelenú a tým začne nový cyklus. Ak má semafor na začiatku zelenú, tá sa v čase f prepne na červenú a potom v čase $f + c$ naspäť na zelenú a máme začiatok cyklu.
Ak už poznáme jeden čas t_0, kedy začal cyklus, vieme určiť aj všetky ostatné – líšia sa od t_0 o celočíselné násobky $(r + c)$.
A teraz už sme pripravení na hlavné otázky, ktoré nás zaujímajú. Predstavme si, že sme ku semaforu prišli v (celočíselnom) čase t. Kde v jeho cykle sme? Pozrime sa na hodnotu $t - t_0$. Ak ide o násobok $r + c$, sme práve na začiatku cyklu – t sa od t_0 líši o dotyčný násobok $r + c$. Takže zvyšok, ktorý dáva táto hodnota po delení $r + c$, nám hovorí, kde v cykle semaforu sa práve nachádzame. Ak $(t - t_0) \bmod (r + c) < r$, ešte svieti zelená. Pre $(t - t_0) \bmod (r + c) = r$ práve naskočila červená a pre ešte väčšie hodnoty sme v červenom intervale.
Ak svieti zelená, môžeme cez semafor rovno prejsť. Ak nie, vieme, že najbližšie zelená zasvieti, keď bude pre nový čas t' platiť, že $t' - t_0$ je násobkom $r+c$. Inými slovami, musíme počkať $r+c-x$ sekúnd, kde $x = (t-t_0) \bmod (r+c)$ je aktuálny zvyšok.
(Technická poznámka na záver: v niektorých programovacích jazykoch, ako napr. C a C++, je operátor modulo implementovaný tak, že pre zápornú hodnotu delenca vracia záporný výstup. Ak chceme na výstupe vždy hodnotu od 0 po $r + c - 1$, treba si dať pozor, aby sme modulo používali len na nezáporné vstupy. To vieme dosiahnuť napr. tak, že namiesto hodnoty $t - t_0$ použijeme hodnotu $t - t_0 + (r + c)$, ktorá dáva rovnaký zvyšok a je zaručene kladná.)

Stručne o pomalších riešeniach

Ak všetky semafory vždy svietia na zelenú, stačí nám priamo použiť Dijkstrov algoritmus a nájsť najrýchlejšiu cestu.
Ak sú všetky časy prechodov po hranách malé, vieme graf prerobiť na taký, v ktorom každú hranu vieme prejsť za 1 sekundu. (Napr. hranu dĺžky 3 si vždy vieme vložením pomocných vrcholov rozdeliť na tri jednotkové hrany.)
Ak je takto zostrojený graf dostatočne malý, môžeme úlohu vyriešiť priamočiarou simuláciou všetkých možností naraz: začínajúc od času 0 si postupne pre každú sekundu zostrojíme množinu vrcholov, v ktorých sa môžeme nachádzať. Keď už poznáme túto množinu pre čas i, pre čas $i + 1$ ju zostrojíme tak, že sa pre každý vrchol, v ktorom sme mohli byť, pozrieme na všetky hrany, ktoré z neho vedú, a pre každú zistíme, či po nej v danej chvíli smieme prejsť.
Pri tomto type riešenia si následne môžeme všimnúť, že akonáhle prvýkrát dosiahneme nejaký vrchol, vieme tam zostať ľubovoľne dlho čakať, a teda v ňom vieme byť aj hocikedy neskôr. Pre každý vrchol grafu nám preto stačí zistiť najskorší čas, kedy v ňom vieme byť. Toto si vieme v grafe s jednotkovými hranami vypočítať vhodne upraveným prehľadávaním do šírky.
Iné, podobne efektívne vylepšenie, je, že pre každý vrchol si môžeme pamätať, ktorými hranami sa nám už z neho podarilo na zelenú odísť, a keď sa nám to podarí pre všetky z nich, môžeme tento vrchol vo zvyšku riešenia

ignorovať.

Vzorové riešenie

Popíšeme si, ako upraviť Dijkstrov algoritmus, aby fungoval pre grafy so semaformi bez zhoršenia asymptotickej časovej zložitosti.
Inicializácia algoritmu: najlepší známy čas τ, v akom vieme byť v každom vrchole, nastavíme na nulu pre štart a na nekonečno pre všetky ostatné vrcholy. Žiaden vrchol ešte nie je označený ako hotový.
Odteraz algoritmus beží v cykle. Každá iterácia cyklu vyzerá nasledovne:

1. Spomedzi vrcholov, ktoré ešte nie sú hotové, vyberieme ten s najmenším časom. Vrchol si označíme v; najmenší čas, kedy doň vieme prísť, je τ_v.
2. Vrchol v označíme za hotový.
3. Pre každú hranu i vedúcu z vrcholu $a_i = v$ do nejakého iného vrcholu $b_i = w$ spravíme nasledovnú úvahu: Vypočítame si najbližší čas ξ_v väčší alebo rovný τ_v, kedy do tejto hrany budeme vedieť vojsť. K tomu pripočítame čas t_i potrebný na prejdenie po nej. Takto sme práve objavili nový spôsob, ako sa dostať do w v čase $\xi_v + t_i$. Ak je doterajšia hodnota t_w väčšia, zmenšíme ju na práve vypočítanú lepšiu.

Rovnako ako pri klasickom Dijkstrovom algoritme si vieme ešte nehotové vrcholy pamätať vo vhodnej efektívnej dátovej štruktúre (usporiadanej množine alebo prioritnej fronte), aby sme v kroku 1 cyklu vedeli efektívne nájsť ten z nich, ktorý máme v tomto cykle spracovať. V kroku 3 potom samozrejme potrebujeme v tejto dátovej štruktúre príslušne upraviť záznam o vrchole w, ak mu zmeníme hodnotu τ_w.

Dôkaz správnosti bude vyzerať tiež veľmi podobne klasickému algoritmu. Prítomnosť semaforov nám síce v priebehu riešenia mení dĺžky hrán v našom grafe, ale na korektnosť algoritmu to nebude mať vplyv.
Tvrdíme nasledovné: Po každom počte iterácií cyklu (aj nula) platí, že pre každý hotový vrchol v je τ_v presne rovné najkratšiemu času, za ktorý sa vieme dostať do v, a pre ostatné vrcholy platí, že τ_v je najkratší čas, za ktorý sa vieme dostať do v, ak po ceste smieme navštíviť len hotové vrcholy.
Navyše platí, že pre ľubovoľný hotový vrchol f a ľubovoľný nie-hotový vrchol g platí $\tau_f \leq \tau_g$. (Z toho následne vyplýva, že keď si všetky vrcholy vypíšeme v poradí, v akom boli prehlásené za finálne, tak budú zároveň usporiadané podľa ich koncovej hodnoty τ.)
Tieto tvrdenia si dokážeme matematickou indukciou podľa počtu iterácií cyklu. Na začiatku to zjavne platí. Pozrime sa teraz na ľubovoľnú jednu iteráciu cyklu, pričom predpokladáme, že na jej začiatku platí vyššie uvedené tvrdenie. Ukážeme si, že potom bude platiť aj na jej konci.
V kroku 1 spomedzi vrcholov, ktoré ešte nie sú hotové, vyberieme v s najmenšou hodnotou τ_v. Táto hodnota skutočne už musí byť finálna. Z indukčného predpokladu vieme, že τ_v je najmenší čas, za ktorý do v vieme prísť len cez hotové vrcholy. No a žiadna iná lepšia cesta do v už nemôže existovať. Totiž keď zoberieme hocijakú inú cestu, na nej budú aj nejaké iné vrcholy, ktoré ešte nie sú hotové. Označme w prvý z nich. Potom už len úsek od štartu po w trvá (aspoň) τ_w, čiže aspoň toľko ako aktuálne τ_v. (To isté slovne: cez iný nie-hotový vrchol sa už nevieme rýchlejšie dostať do v, keďže už samotná cesta do toho vrcholu nám trvá aspoň rovnako dlho.)
Nájdený vrchol v teda skutočne v kroku 2 môžeme prehlásiť za hotový.
Teraz sa nám teda zmenila množina hotových vrcholov, čo ale znamená, že sa môžu zmeniť hodnoty τ pre vrcholy, ktoré ešte nie sú hotové – teraz už cesty do nich môžu prechádzať aj cez v.
Zjavne ale stačí uvažovať také cesty, na ktorých je v posledný hotový vrchol. Totiž ak by sme z v pokračovali ďalej do iného hotového vrcholu x, prišli by sme tam v čase väčšom ako τ_v. Lenže my sme x už vyhlásili za hotový skôr ako v, a teda nutne $\tau_x \leq \tau_v$. Inými slovami, existuje lepší spôsob, ako sa dostať do x a neísť pri tom cez v.
Pre každý nie-hotový vrchol w teda stačí uvažovať nové cesty, pri ktorých najskôr prídeme cez samé hotové vrcholy do v a potom prejdeme priamou hranou z v do w.
Tu si teraz zdôrazníme vlastnosť našej úlohy, vďaka ktorej Dijkstrov algoritmus (s našimi úpravami) stále funguje: **nikdy sa nám neoplatí prísť do nejakého vrcholu / prejsť po nejakej hrane naschvál neskôr**. Ak vieme prísť do vrcholu skôr, nič nepokazíme, keď to spravíme – najhoršie, čo sa nám môže stať, je,

že budeme potom dlhšie čakať. Ak vieme vôjsť na hranu, nemá zmysel čakať a vôjsť až neskôr – nikdy nemôže nastať situácia, kedy by sme na hranu neskôr vošli ale skôr z nej vyšli.
Pre každú hranu z práve hotového vrcholu v do iného, ešte nie hotového vrcholu w teda má zmysel uvažovať len jedinú možnosť: najrýchlejšie ako sa dá sa dostaneme do v a následne najskôr ako vieme použijeme hranu z v do w.

Pri implementácii s efektívnou dátovou štruktúrou má toto riešenie časovú zložitosť $O((n+m)\log n)$. Totiž za celý beh programu n-krát z dátovej štruktúry vyberieme ďalší vrchol na spracovanie a pre každú hranu vw sa raz pozrieme na to, či optimálna cesta do v nasledovaná hranou do w nezlepší τ_w. Ak sa tak stane, musíme do našej dátovej štruktúry vložiť nový záznam pre w. Dokopy teda máme nanajvýš $n+m$ operácií s našou dátovou štruktúrou a každú z nich vieme spraviť v čase $O(\log n)$.
Nižšie uvádzame ukážkovú implementáciu v Pythone pomocou prioritnej fronty (implementovanej ako binárna halda).

Listing programu (Python)

```
from heapq import heappush, heappop

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

    def cas_v_cieli(self, cas_na_starte):
        ''' kedy najskor prideme na koniec hrany, ak vieme, kedy sme na jej zaciatku? '''
        zvysok = (cas_na_starte + self.offset) % (self.zelena + self.cervena)
        if zvysok < self.zelena:
            return cas_na_starte + self.dlzka
        else:
            return cas_na_starte - zvysok + self.zelena + self.cervena + self.dlzka

# nacitame vstup
n, m = [ int(_) for _ in input().split() ]
G = [ [] for _ in range(n) ]
for mm in range(m):
    tokens = input().split()
    start, ciel, dlzka, zelena, cervena = [ int(_) for _ in tokens[:5] ]
    start, ciel = start-1, ciel-1
    if tokens[5] == 'Z':
        offset = zelena - int(tokens[6])
    else:
        offset = zelena + cervena - int(tokens[6])
    G[start].append( Hrana( ciel=ciel, dlzka=dlzka, zelena=zelena, cervena=cervena, offset=offset ) )

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

# v cykle spracuvame vrcholy
while len(Q) > 0:
    d, kde = heappop(Q)
    # ak sme tento vrchol uz spracovali, len zahodime starsi (neoptimalny) zaznam o nom
    if dist[kde] < d: continue
    # prejdeme vsetky hrany iduce odtialto a prepocitame optimalne casy pre susedne vrcholy
    for hrana in G[kde]:
        kam, kedy = hrana.ciel, hrana.cas_v_cieli(d)
        if kedy < dist[kam]:
            dist[kam] = kedy
            heappush( Q, (kedy,kam) )

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

A-I-2 Pokazený robot

Vstup si predspracujeme tak, že každému číslu d_i dáme znamienko: kladné ak sa hýbeme na východ, záporné ak na západ. Vo všetkých nižšie uvedených riešeniach používame takto upravené d_i.

Riešenie za 2 body

Ak $k = 0$, robot nerobí žiadne chyby. Stačí nám teda pre každý p-prvkový úsek príkazov zistiť, o koľko robota posunie. To po našej úprave zodpovedá tomu, že chceme zistiť súčet p po sebe idúcich hodnôt postupnosti d a následne zobrať jeho absolútnu hodnotu. Toto vieme ľahko spraviť v lineárnom čase od n, teda v čase $O(n)$.
Jedno možné riešenie začne tým, že si v cykle spočíta súčet $d_1 + \cdots + d_p$, teda súčet prvého možného úseku príkazov. Následne k tejto hodnote pripočítame d_{p+1} a odpočítame d_1, čím dostaneme súčet $d_2 + \cdots + d_{p+1}$, teda súčet druhého z úsekov, ktoré nás zaujímajú, a tak ďalej.
Iné rovnako dobré riešenie dostaneme tak, že si spočítame prefixové súčty postupnosti d a z tých už vieme v konštantnom čase zistiť súčet ľubovoľného úseku.

Riešenie za 6 bodov

Do vyššie uvedeného riešenia teraz doplníme riešenie pre druhú a tretiu sadu vstupov.
Pre $n \leq 1000$ si môžeme dovoliť každý p-príkazový úsek zvlášť celý prejsť a rozumne efektívne spracovať. Stačí nám teda efektívne vyriešiť jednoduchšiu úlohu: ako najďalej vieme dostať robota, ak máme postupnosť p príkazov, ktoré všetky musí vykonať? Naďalej pritom platí, že nanajvýš k z nich môžeme obrátiť.
Ak chceme dostať robota čo najďalej od jeho štartovej pozície, máme na výber dve symetrické možnosti: buď ho dostaneme čo najďalej na východ, alebo čo najďalej na západ. Riešenie oboch je presne rovnaké a môžeme naň použiť ten istý algoritmus. Ukážeme si riešenie prvej možnosti. V tej máme postupnosť hodnôt $d_1, \ldots, d_p$, môžeme nanajvýš k jej členom zmeniť znamienko a chceme na konci dostať súčet s čo najväčšou (kladnou) hodnotou.
U takto sformulovanej úlohy by malo byť zjavné, že ju vždy vieme optimálne vyriešiť pažravo. V prvom rade sa zjavne neoplatí meniť kladné čísla na záporné (spravenie tejto operácie zníži súčet) a v druhom rade ak máme na výber viac ako k záporných čísel, oplatí sa na kladné zjavne zmeniť tie s najväčšou absolútnou hodnotou.
Stačí si teda postupnosť $d_1, \ldots, d_p$ usporiadať od najmenšieho prvku po najväčší a potom prvým k (alebo menej, ak sa nám minú záporné členy) zmeniť znamienko zo záporného na kladné.
Postupne vyriešime zvlášť každý z $n-p+1$ úsekov. Počet úsekov vieme teda zhora odhadnúť ako $O(n)$. Pre každý úsek nájdeme jeho optimálne riešenie v čase $O(p \log p)$, dokopy teda dostávame riešenie s časovou zložitosťou $O(np \log p)$.
(Logaritmu z časovej zložitosti sa vieme zbaviť drobnou optimalizáciou: namiesto usporiadania postupnosti d stačí použiť lineárny algoritmus na nájdenie k-teho najmenšieho prvku.)

Vzorové riešenie

Skombinujeme teraz myšlienky z oboch vyššie uvedených riešení. Z úplne prvého si zoberieme myšlienku tzv. *sliding window* (posuvného okna) – budeme sa postupne posúvať po vstupnej postupnosti a optimálne riešenie pre nový úsek vždy prepočítame vhodnou úpravou toho predchádzajúceho. No a zo 6-bodového riešenia si zoberieme myšlienku pažravého algoritmu, ktorý nám vyrobí optimálne riešenie pre konkrétny úsek.
(Naďalej budeme riešiť len nájdenie spôsobu, ako dostať robota čo najviac na východ. V programe potom celé toto riešenie použijeme dvakrát, pričom pred druhým použitím celý vstup otočíme „hore nohami“.)

Čo presne potrebujeme vedieť o práve spracúvanom úseku, aby sme vedeli povedať, aké má optimálne riešenie?

- Aký by mal tento úsek súčet s, ak by sme nič nemenili?
- Aký hodnotu z má súčet k najmenších záporných prvkov v ňom (resp. súčet všetkých záporných prvkov, ak ich je menej)?

Keď budeme poznať tieto dve hodnoty, odpoveďou, ktorú pre tento úsek hľadáme, je zjavne $s - 2z$. (Napr. ak pohyb o -3 zmeníme na pohyb o $+3$, posunie sa tým naša záverečná poloha o 6 doprava.)
Hodnotu s si už vieme udržiavať, teda prepočítať ju, keď sa posunieme na najbližší ďalší úsek. Aby sme vedeli udržiavať aj hodnotu z, budeme si hodnoty v aktuálnom úseku pamätať v dvoch usporiadaných množinách. V množine A budú tie prvky, ktoré nás zaujímajú – k najmenších záporných, resp. všetky záporné, ak ich je menej ako k. V množine B budú všetky ostatné prvky. No a k tomu si ešte budeme explicitne pamätať samotnú

hodnotu z, teda súčet množiny A. Ten si upravíme vždy, keď zmeníme množinu A. Napr. ak do nej pridáme nový prvok, pripočítame následne jeho hodnotu aj ku zapamätanému súčtu z.
Aké zmeny potrebujeme spraviť, keď sme skončili spracovanie jedného úseku vstupu a chceme sa posunúť na ďalší?

- Hodnotu d_i, ktorá práve náš úsek opustila, odstránime z množiny, v ktorej sa nachádza.
- Ak sme hodnotu d_i odstránili z A a najmenší prvok v B je záporný, presunieme ho z B do A, aby naďalej platilo, že v A máme k najmenších záporných prvkov.
- Hodnotu d_{i+p}, ktorá práve do nášho úseku pribudla, pridáme do jednej z množín: ak je záporná, do A, ak nie je, do B.
- Ak má teraz A priveľa prvkov (presne $k+1$), najväčší z nich presunieme do B.

Každú z týchto operácií vieme spraviť s usporiadanou množinou v čase logaritmickom od počtu jej prvkov, čo v našom prípade vieme zhora odhadnúť $O(\log p)$. Dokopy teda v tomto čase spracujeme každý z $O(n)$ úsekov, čím dostávame celkovú časovú zložitosť $O(n \log p)$.

Listing programu (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;

// pridanie novej hodnoty x do spracuvaneho useku
void add(long long x) {
    // zvacsime si sucet useku
    s += x;
    // ak je hodnota kladna alebo 0, vlozime ju do mnoziny B a sme hotovi
    if (x >= 0) { B.insert(x); return; }
    // zapornu hodnotu vlozime do mnoziny A, zvacsime si jej sucet
    z += x;
    A.insert(x);
    // ak uz je mnozina A privelka, presunieme z nej najvacsiu ("najmenej zapornu") hodnotu do B
    if (int(A.size()) > k) {
        long long biggest = *prev(A.end());
        A.erase( prev(A.end()) );
        z -= biggest;
        B.insert(biggest);
    }
}

// odstranenie jednej hodnoty x zo spracuvaneho useku
void remove(long long x) {
    // zmensime si sucet useku
    s -= x;
    // ak mame takuto hodnotu v B, len ju zahodime a sme hotovi
    auto it = B.find(x);
    if (it != B.end()) { B.erase(it); return; }
    // inak ju urcite mame v A, odstranime ju odtial a prepocitame si sucet A
    z -= x;
    A.erase(A.find(x));
    // prave sme zmensili A
    // ak mame v B nejake zaporne cisla (to sa mohlo stat, len ak uz A bola "plna"),
    // presunieme najmensie ("najviac zaporne") cislo z B do A
    if (B.empty()) return;
    long long smallest = *B.begin();
    if (smallest >= 0) return;
    z += smallest;
    A.insert(smallest);
    B.erase( B.begin() );
}

// najdeme optimalne riesenie smerom na vychod pre aktualnu postupnost D[]
long long solve() {
    if (p == 0) return 0;
    s = 0;
    z = 0;
    A.clear();
    B.clear();
```

```
    // do spracuvaneho useku dame prvych p prvkov, potom sa pozrieme na jeho optimalne riesenie
    for (int i=0; i<p; ++i) add( D[i] );
    long long answer = s - 2*z;
    // a nasledne vzdy spracuvany usek posunieme o 1 doprava a pozrieme, ci nove riesenie nie je lepsie
    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() {
    // nacitame vstup, v D[] si vyrobime znamienka podla smerov
    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;
    }
    // dvakrat pustime solve(): raz na povodnu postupnost, raz na otocenu
    long long answer = solve();
    for (auto &d : D) d = -d;
    answer = max( answer, solve() );
    cout << answer << endl;
}
```

Vyššie popísané riešenie vieme implementovať aj v jazykoch, ktoré nemajú vo svojej štandardnej knižnici efektívnu implementáciu usporiadanej (multi)množiny. Stačí nám mať prvky množiny A uložené v maximovej a prvky množiny B v minimovej halde. Jediný netriviálny krok takejto implementácie je zmazanie prvku, ktorý odstraňujeme. Ak si robíme vlastnú implementáciu haldy, vieme to spraviť tak, že si ku každému prvku pamätáme pointer, kde a v ktorej halde sa práve nachádza. Ak používame knižničnú implementáciu haldy, môžeme toto mazanie spraviť lenivo. Ku každému prvku si budeme pamätať, či je v A alebo B, a ku množine A si ešte budeme pamätať jej skutočnú veľkosť. Ak mažeme prvok z množiny A, tak len toto počítadlo zmenšíme, a inak nerobíme nič. A kedykoľvek, keď sa nám na vrchu ľubovoľnej z háld zjaví záznam, ktorý už mal byť zmazaný, tak ho zahodíme.

A-I-3 Piškóty

Podúlohy A a B: konkrétne postupnosti piškót

Emmika vie prvú postupnosť piškót celú zjesť napríklad nasledovne:

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

(Pri tomto postupe vždy zjeme najľavejšiu piškótu, ktorá je čokoládou hore. Po prvom kroku dostaneme konfiguráciu, ktorá sa potom každé dva kroky zopakuje, len máme o dve piškóty menej.)
Nie je to však jediný možný postup. Ďalšia možnosť je napríklad táto:

- „CCCC̲CC“
- „CCC̲P-PC“
- „C̲P-C-PC“
- „-C-C-PC̲“
- „-C-C-C-“

(Pri tomto postupe najskôr v ľubovoľnom poradí zjeme piškóty na párnych indexoch, číslujúc od nuly, pričom každú zo zvyšných piškót preklopíme dvakrát. Následne už vieme zvyšné piškóty dojesť v ľubovoľnom poradí, keďže žiadne dve už priamo nesusedia.)

Ak sú piškóty na začiatku v konfigurácii „CPPPPPC“, tak ich Emmika zjesť všetky nevie. Prečo? Všimnime si, že na začiatku má dve voľby: môže zjesť len piškóty na koncoch postupnosti. Navyše, voľba je *symetrická*: nech zje ktorúkoľvek z nich, výsledná postupnosť nezjedených piškót bude „CPPPPC“. Tým sme sa dostali do podobnej situácie, avšak s 6 piškótami namiesto 7. Ľahko zovšeobecníme, že presne to isté sa stane v každom zo štyroch nasledujúcich krokov. Po nich sa teda nutne dostaneme k postupnosti „CC“. No v tomto prípade platí, že nech zje Emmika ktorúkoľvek z dvoch zostávajúcich piškót, tá druhá sa jej otočí piškótovou stranou nahor a Emmika ju podľa svojich pravidiel nebude smieť dojesť. Teda v tejto podúlohe Emmika nevie zjesť všetky piškóty.

Podúloha C: overenie, či sa daná postupnosť dá zjesť

V tejto podúlohe sme chceli nájsť (ľahko overiteľnú) podmienku ktorá nám povie, či sa nejaká postupnosť dá alebo nedá zjesť.
Ak ste sa s úlohou trochu pohrali, pravdepodobne ste si rozmysleli, čo sa deje pre malé počty piškót: Ak máme len jednu piškótu, tak ju vieme zjesť práve ak je čokoládou stranou hore. Ak máme dve susediace piškóty, vieme ich dojesť práve ak len jedna z nich je čokoládovou stranou hore (ak sú hore čokoládovou stranou dve alebo žiadna tak to zjavne nejde). Čo pri troch piškótach? Zas to ide ak začíname s jednou alebo troma piškótami čokoládou hore. Inak povedané, zatiaľ to vyzerá, že pri *nepárnych* počtoch čokoládou hore otočených piškót vieme postupnosť dojesť a inak to nejde.
Máme teda hypotézu, že postupnosti s nepárnymi počtami čokoládou hore otočených piškót zjesť idú a inak to nejde. Samozrejme, teraz si musíme naše tvrdenie skúsiť poriadne dokázať.
Tento dôkaz spravíme v dvoch krokoch: Najskôr si ukážeme, že každý úsek piškót, v ktorom je čokoládou hore nepárny počet piškót, zjesť vieme. No a následne si ukážeme dôvod, pre ktorý nepôjde zjesť žiaden zo zvyšných úsekov.
Ako prvý krok si všimnime, že keďže obraciame iba piškóty, ktoré boli susedmi v originálnom rade, akonáhle sa nám rad rozdelí na samostatné úseky, jedenie piškót z jedného úseku už neovplyvní iné úseky. Inými slovami, rozdelené postupnosti piškót spolu *neinteragujú*. Teda ak máme nejakú postupnosť piškót na stole (z ktorej už nejaké môžu chýbať), Emmika ich vie dojesť práve ak vie samostatne dojesť každý z jej súvislých úsekov.
Majme teraz súvislý úsek, v ktorom je *nepárny* počet piškót čokoládou hore. Čo sa stane, keď zjeme *prvú (najľavejšiu)* z týchto piškót?

- Ak je zjedená piškóta úplne prvá v rade, naľavo od nej sa nič nestane. Podobne ak je v rade posledná sa nič nestane napravo od nej.
- Ak sú nejaké piškóty naľavo od nej, sú všetky piškótovou stranou hore. Najpravejšiu z nich otočíme čokoládou hore. Takto nám vznikne kratší úsek, v ktorom je práve jedna piškóta čokoládou hore.
- Ak sú nejaké piškóty napravo od práve zjedenej, bol medzi nimi párny počet piškót čokoládou hore. Najľavejšiu z týchto piškót otočíme. Bez ohľadu na to, ktorou stranou hore bola táto piškóta, sa tým zmení parita počtu piškót čokoládou hore (buď jedna ubudne alebo jedna pribudne). Teda aj napravo od zjedenej piškóty nám vždy vznikne úsek s nepárnym počtom piškót čokoládou hore.

A tým sme vlastne už vyhrali. Keď vždy zjeme najľavejšiu piškótu, ktorá je čokoládou hore, budeme mať v každom okamihu len samé úseky, ktoré majú takýchto piškót nepárny počet. Kým sa nám všetko neminie, vždy vieme zjesť ďalšiu piškótu. A keďže celkový počet nezjedených piškót neustále klesá, musí celý postup skončiť až tým, že dojeme všetko.
(Iný spôsob ako poriadnejšie sformulovať tento istý dôkaz by bol matematickou indukciou podľa n dokázať tvrdenie, že každý úsek n piškót, v ktorom je ich nepárny počet čokoládou hore, vieme zjesť postupom „vždy zjedz najľavejšiu piškótu, ktorú môžeš“. Pri dôkaze indukčného kroku spravíme vyššie uvedený rozbor, čo sa stane v prvom kroku riešenia, a následne uzavrieme, že z indukčného predpokladu už vieme, že oba kratšie úseky piškót vieme celé dojesť.)

Čo teraz s prípadom, že máme párny počet čokoládou hore otočených piškót? Teraz musíme ukázať, že *žiadny* postup nebude fungovať. Ako to ukážeme? Po prvé si všimnime, že ak máme neprázdny úsek s *nula* piškótami čokoládou hore, tak žiadnu z nich nevieme zjesť, a teda tento úsek ostane nezjedený.
Ukážeme, že nech robí Emmika čokoľvek, ostane jej časom aspoň jeden neprázdny úsek takéhoto typu. Použijeme na to nasledovný invariant: ukážeme, že ak máme neprázdny úsek s *párnym* počtom čokoládou hore otočených piškót, nech Emmika zje akúkoľvek piškótu, vždy skončí s nejakým *kratším ale stále neprázdnym* úsekom s párnym počtom čokoládou hore otočených piškót.
Keď toto dokážeme, bude z toho už vyplývať, že Emmika nemôže všetko dojesť. Predstavme si totiž, že sledujeme len jeden konkrétny takýto úsek. (Vždy, keď Emmika zje piškótu tak, že vzniknú z jedného úseku dva nové, vyberieme si jeden z nich, ktorý má párny počet piškót čokoládou hore, a ten druhý odignorujeme.) V každom kroku sa sledovaný úsek skráti, takže krokov, ktoré ho zmenia, bude len konečne veľa. Po nich teda nutne dostaneme situáciu, kedy máme úsek piškót, ktorý je ešte stále neprázdny, ale Emmika už z neho nič ďalšie nevie zjesť – čiže všetky piškóty v ňom sú piškótovou stranou hore.
A prečo teda vyššie uvedené tvrdenie platí? Predstavme si, že máme neprázdny úsek s $2k$ čokoládou hore otočenými piškótami a Emmika sa rozhodla zjesť i-tu z nich (číslujeme $1 \leq i \leq 2k$). Pred touto piškótou je $i-1$ piškót čokoládou hore a za ňou je $2k-i$ piškót čokoládou hore. Keďže súčet týchto dvoch hodnôt je $2k-1$, čo je nepárne číslo, práve jedna z týchto hodnôt je nepárna.
Bez ujmy na všeobecnosti nech je nepárny počet čokoládou hore otočených piškót *naľavo* od piškóty, ktorú sa Emmika rozhodla zjesť. Tento úsek piškót je zaručene neprázdny (keďže obsahuje aspoň jednu piškótu čokoládou hore). A keď Emmika zje svoju vybranú piškótu, v tomto úseku následne jednu piškótu otočí a tým zmení paritu počtu piškót čokoládou hore z nepárnej na párnu. Tým sme teda naozaj dostali to, čo sme sľubovali – nový kratší úsek, ktorý je neprázdny a v ktorom naďalej platí náš invariant.

Podúloha D: Všetky víťazné ťahy

Pre vyriešenie poslednej podúlohy použijeme myšlienku z riešenia tej predchádzajúcej. Už sme si dokázali, že Emmika vie zjesť súvislú neprázdnu postupnosť piškót práve vtedy, ak je v nej nepárny počet čokoládou hore otočených piškót. A ak má piškótová postupnosť viacero samostatných úsekov (lebo niektoré piškóty už boli zjedené), tak sa dajú dojesť všetky piškóty práve vtedy, ak sa dá dojesť každý úsek zvlášť – teda ak je *v každom súvislom úseku* nepárny počet piškót čokoládou hore.
Algoritmus bude teda vyzerať nasledovne: rozdelíme si vstup na súvislé úseky piškót a spočítame si hodnotu pre každý z nich separátne. Ak je pre niektorý z nich odpoveďou nula, vypíšeme nulu. Inak vypíšeme súčet možností.
Ako spočítame hodnotu pre daný súvislý úsek? Jedna možnosť je skontrolovať, zvlášť pre každú možnú čokoládovú piškótu X, čo sa stane, ak ju zjeme a otočíme jej priamych susedov. Aj o úseku naľavo aj o úseku napravo od nej potrebujeme zistiť, či bude mať párny alebo nepárny počet piškót čokoládou hore. Piškótu X smieme zjesť ako prvú len vtedy, ak je pre oba tieto úseky počet čokoládových piškót nepárny (alebo, samozrejme, ak už sú prázdne). Toto riešenie vieme implementovať v lineárnom čase, napríklad pomocou prefixových súčtov.

Jednoduchší spôsob je však sa ešte trochu zamyslieť pred tým, než niečo začneme strmhlavo implementovať. Čo sa stane, ak máme v súvislom úseku $2k+1$ čokoládou hore otočených piškót a zjeme i-tú z nich? V riešení časti C sme si ukázali, že určite môžeme zjesť najľavejšiu z nich ($i = 1$) a zo symetrie takisto aj najpravejšiu z nich ($i = 2k+1$).
Pozrime sa teraz na ostatné i. Naľavo od vybranej piškóty je $i-1$ a napravo $2k+1-i$ čokoládou hore otočených piškót. Keďže $1 < i < 2k+1$, oba tieto úseky piškót sú neprázdne. Keď zjeme vybranú piškótu, v každom úseku jednu piškótu otočíme a tým mu zmeníme paritu počtu piškót čokoládou hore. Teda pre nepárne i platí, že v každom z nových dvoch kratších úsekov bude *nepárny* počet piškót čokoládou hore (a teda ich dojesť vieme), zatiaľ čo pre párne i bude v oboch kratších úsekoch *párny* počet piškót čokoládou hore (a teda nevieme dojesť ani jeden z nich).
Teda v každom úseku s $2k+1$ piškótami čokoládou hore platí, že Emmika smie zjesť prvú, tretiu, piatu, ..., až $(2k+1)$. spomedzi týchto piškót. Má teda na výber práve $k+1$ možností.
Implementovať riešenie v Pythone ide napríklad takto:

Listing programu (Python)

```
n = int(input())

piskoty = input()
useky = piskoty.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)
```

A-I-4 Nechaj to na solver

Podúlohy A-E: investície

O každom projekte sa potrebujeme rozhodnúť, či ho podporíme alebo nie. Toto je prirodzené modelovať tak, že pre každý projekt i budeme mať binárnu premennú x_i hovoriacu, či ho podporíme alebo nie: 0 (false) je nie, 1 (true) je áno.
Ak podporíme projekt i, dáme naň s_i peňazí a neskôr dostaneme výnos v_i, na konci teda budeme mať o $(v_i - s_i)$ peňazí viac ako na začiatku. Náš záverečný stav na konte teda vieme vyjadriť ako jeho úvodný stav (e) plus súčet všetkých výrazov tvaru $(v_i - s_i)x_i$: nepodporené projekty stav konta nezmenia (násobíme nulou), podporené ho zmenia o svoj čistý zisk. Túto hodnotu chceme maximalizovať.
V podúlohách A a D nemáme ani konflikty, ani závislosti. V týchto podúlohách nám stačí dodržať jediné obmedzenie: musíme mať dosť peňazí na podporenie všetkých vybratých projektov. Inými slovami, súčet $s_i x_i$ cez všetky i nesmie prekročiť e.
Ako modelovať konflikty? Podmienku „spomedzi projektov s a t smieš podporiť najviac jeden“ vieme zjavne matematicky zapísať nasledovne: $x_s + x_t \leq 1$. Toto je priamo obmedzenie v povolenom tvare, takže ho len pridáme do nášho ILP a máme hotovo. (Môžeme si tiež všimnúť, že presne rovnako by sa modelovali aj všeobecnejšie konflikty zahŕňajúce viac ako dva projekty, ak by také niečo mohlo nastať.)
A ako modelovať závislosti? Ak projekt u závisí na projekte v, nemôže naraz platiť $x_u = 1$ a $x_v = 0$. Inými slovami, x_u musí vždy byť menšie alebo rovné ako x_v. No a $x_u \leq x_v$ je opäť podmienka v povolenom tvare, takže máme hotovo.
Vygenerovaný ILP vo formáte pre `lp_solve` pre príklad zo zadania:

```
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;
```

Optimálne riešenie pre vstup D má zisk 242 559 804 a pre vstup E je optimálny zisk 578 053 900.
Program v Pythone riešiaci vstupy pomocou knižice PuLP:

Listing programu (Python)

```
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 lebo interne cislujeme od 0
z = int(input())
```

```
D = [ [ int(_)-1 for _ in input().split() ] for __ in range(z) ]

problem = pulp.LpProblem('Investicie', 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: chovatelia prasiat

Tentokrát na modelovanie použijeme celočíselné premenné. Stačia nám tie popisujúce transport: pre každý typ plodov t, sýpku i a chovateľa j budeme mať premennú $x_{t,i,j}$ predstavujúcu počet kg tohto typu, ktoré chovateľ j dostane a zaplatí v balíku poslanom zo sýpky i.
Jednotlivé obmedzenia zodpovedajúce zadaniu teraz vyzerajú nasledovne:

- Jedno obmedzenie pre každú sýpku: súčet všetkého z nej odchádzajúceho je menší alebo rovný jej kapacite. (Zjavne nemá zmysel zbierať viac ako potom rozošleme.)
- Jedno obmedzenie pre každú dvojicu (sýpka, chovateľ): súčet poslaných bukvíc a žaluďov v balíku je menší alebo rovný w.
- Jedno obmedzenie pre každú dvojicu (typ plodov, chovateľ): celkové množstvo doručených plodov tohto typu je nanajvýš rovné príslušnej hodnote g, teda limitu udávajúcemu, koľko je toho tento chovateľ ochotný kúpiť.

No a cieľ, teda náš zisk? Za zber máme konštantnú stratu 5000, teda zisk -5000. Balík poslaný zo sýpky i chovateľovi j má váhu $(x_{1,i,j} + x_{2,i,j})$ a teda zaň zaplatíme $\ell_{i,j} \cdot (x_{1,i,j} + x_{2,i,j})$. No a za predaj plodov typu t chovateľovi j dostaneme $h_{j,t} \cdot (x_{t,1,j} + \cdots + x_{t,i,j})$. Toto celé sčítané dokopy nám dá jeden veľký lineárny výraz, ktorého hodnotu chceme maximalizovať.
Vygenerovaný ILP vo formáte pre `lp_solve` pre príklad zo zadania:

```
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;
```

Optimálne riešenia pre hodnotené vstupy: H = 2 842 033, I = 1 219 206 a J = 2 287 290.
Program v Pythone riešiaci vstupy pomocou knižice PuLP:

Listing programu (Python)

```
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('Chovatelia', 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)
]

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

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 záver: Súťažná úloha o chovateľoch sa tiež dala modelovať a efektívne riešiť aj pomocou iného formalizmu – vieme ju vyriešiť aj nájdením najdrahšieho toku vo vhodne postavenej sieti, resp. najdrahšieho párovania vo vhodne postavenom ováhovanom bipartitnom grafe. Zložitejšie úlohy podobného typu však už túto vlastnosť vo všeobecnosti nemajú – optimalizačné úlohy, ktoré zahŕňajú viac ako jeden typ tovaru, pričom tieto typy nie sú nezávislé, sú vo všeobecnosti veľmi ťažké.

ŠTYRIDSIATY PRVÝ ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Truc Lam Bui, Michal Forišek, Ján Hozza, Paulína Smolárová
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: NIVAM – Národný inštitút vzdelávania a mládeže, Bratislava 2025