A-II-1 Hladná myška

Základné riešenie za pár bodov: Po každom pohybe je myška v niektorom vrchole (n možností) a má nejakú celočíselnú hmotnosť ($c = \max c_i$ zmysluplných možností). Pre každú začiatočnú hmotnosť Algernona môžeme postupne prehľadávať graf, ktorého vrcholy tvorí všetkých možných nc stavov Algernona a ktorého hrany zodpovedajú jednotlivým pohybom Algernona po bludisku.
Lepšie riešenie môžeme založiť na dvoch pozorovaniach, z ktorých každé môžeme použiť samostatne.
Prvé pozorovanie je, že ak existujú dve možnosti ako Algernona dostať do tej istej miestnosti, tá z nich, kde je chudší, je zjavne lepšia. Pre každú miestnosť nás preto bude zaujímať len najlepšia možnosť, ako sa do nej dostať.
Druhé pozorovanie: Nech x je maximálna štartovacia hmotnosť, s ktorou sa ešte Algernon vie dostať cez bludisko. Potom vieme, že táto vlastnosť je monotónna – ak bude mať Algernon na štarte ľubovoľnú hmotnosť menšiu ako x, do cieľa sa dostane, a ak bude mať ľubovoľnú väčšiu hmotnosť, do cieľa sa už nedostane. To ale znamená, že pri hľadaní optimálneho x nemusíme skúšať všetky možnosti – namiesto toho môžeme použiť binárne vyhľadávanie.
Spojením oboch pozorovaní vieme dostať riešenie s časovou zložitosťou $\Theta(m \log n \log c)$: budeme mať $\log_2 c$ iterácií binárneho vyhľadávania a v každej z nich pre konkrétnu štartovú hmotnosť x použijeme Dijkstrov algoritmus, aby sme pre každú miestnosť zistili, či a s akou najmenšou hmotnosťou sa do nej vie Algernon dostať. Detaily tohto riešenia neuvádzame, keďže budú dostatočne podobné nižšie uvedenému vzorovému riešeniu.

Vzorové riešenie

Nebudeme potrebovať ani binárne vyhľadávanie, ak sa na celú úlohu pozrieme od konca. Zjavne nie len pre štartovú miestnosť ale pre úplne každú miestnosť v bludisku existuje nejaká maximálna hmotnosť, ktorú tam môže Algernon mať, ak sa ešte má vedieť dostať do cieľa. Tieto hodnoty budeme chcieť určiť.
Tieto hodnoty sa správajú podobne ako vzdialenosti. Označme $D[i]$ najväčšiu známu hmotnosť, s ktorou vo vrchole i sa Algernon vie dostať do cieľa. (Táto hmotnosť už v sebe obsahuje syr zjedený vo vrchole i.) Na začiatku máme $D[n-1] = \infty$ a $D[i] = -\infty$ pre všetky vrcholy.
Hodnoty D vieme postupne zväčšovať. Majme nejakú chodbičku, ktorá spája miestnosti a a b a platí pre ňu maximálna hmotnosť c. Pripomíname, že s_i je množstvo syra v miestnosti i. Ak už poznáme hodnotu $D[b]$, čo vieme povedať o hodnote $D[a]$? Ak sa chceme dostať z miestnosti a do cieľa, jednou z možností je prejsť práve uvažovanou chodbičkou do miestnosti b a odtiaľ sa dostať do cieľa. Aby sa nám to celé podarilo, musíme vedieť prejsť chodbičkou (čiže počas prechodu ňou mať hmotnosť nanajvýš c) a keď následne zjeme syr v miestnosti b, musíme vážiť nanajvýš $D[b]$. Toto vieme spraviť, ak sme v miestnosti a s hmotnosťou nanajvýš $\min(c, D[b] - s_b)$. Nová hodnota $D[a]$ bude teda maximom z doterajšej hodnoty $D[a]$ a práve vyskúšanej možnosti.
No a teraz môžeme spraviť rovnakú úvahu ako pri klasickom Dijkstrovom algoritme. Algoritmus bude prebiehať v kolách. V každom kole sa pozrieme na vrcholy, ktoré ešte nie sú označené ako hotové, a ten z nich, ktorý má aktuálne najväčšiu hodnotu D, prehlásime za hotový. Toto môžeme spraviť, lebo táto hodnota sa už nemá ako zmeniť. (Aktuálna hodnota je najväčšia hmotnosť, s ktorou sa odtiaľ Algernon vie dostať do cieľa idúc len cez už hotové vrcholy. No a vieme, že ak by sme išli cez iný nespracovaný vrchol, lepšie riešenie už nedostaneme, lebo už v tom vrchole by sme museli mať nanajvýš rovnakú hmotnosť a pri ďalších pohyboch späť už táto hmotnosť nemá ako stúpnuť.)
Akonáhle sme vrchol prehlásili za hotový, spracujeme všetky chodbičky, ktoré z neho vedú, a pre každý ešte nie hotový vrchol skúsime vyššie popísaným spôsobom zlepšiť jeho doterajšiu hodnotu D.
Tento algoritmus má teda rovnakú časovú zložitosť (a tiež takmer totožnú implementáciu) ako klasický Dijkstrov algoritmus pre hľadanie najkratších ciest: $O(m \log n)$.

Listing programu (C++)

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

const int NEKONECNO = 987654321;

struct hrana { int kam, limit; };

int main() {
```

```
    // nacitame vstup
    int N, M;
    cin >> N >> M;
    vector<int> S(N);
    for (int n=0; n<N; ++n) cin >> S[n];
    vector< vector<hrana> > G(N);
    for (int m=0; m<M; ++m) {
        int a, b, c;
        cin >> a >> b >> c;
        G[a].push_back( {b,c} );
        G[b].push_back( {a,c} );
    }

    // spustime Dijkstrov algoritmus od konca
    vector<int> D(N, -NEKONECNO);
    D[N-1] = +NEKONECNO;
    set< pair<int,int> > Q;
    for (int n=0; n<N; ++n) Q.insert( {D[n],n} );
    while (!Q.empty()) {
        int kde = (--Q.end())->second;
        Q.erase( --Q.end() );
        for (auto h : G[kde]) {
            int pred = h.kam, maxw = h.limit;
            int nove = min( maxw, D[kde]-S[kde] );
            if (nove > D[pred]) {
                Q.erase( {D[pred],pred} );
                D[pred] = nove;
                Q.insert( {D[pred],pred} );
            }
        }
    }
    if (D[0] > 0) {
        cout << D[0] << endl;
    } else {
        cout << (-1) << endl;
    }
}
```

A-II-2 Červené jabĺčko

Nech by Janko položil hocijako komplikovane formulovanú áno/nie otázku, vždy o nej môžeme uvažovať nasledovne: Z Jankovho pohľadu existuje 1000 *možných svetov*, ktoré sa od seba líšia len tým, ktoré z jeho 1000 jabĺk je červené. V niektorých svetoch by Peťka na otázku, ktorú práve položil, odpovedalá „áno“, v ostatných „nie“.
Predstavme si, že sme zobrali všetky jablká a rozdelili ich na dve kôpky. Na prvú dáme tie, pre ktoré by Peťka odpovedala „áno“, keby boli červené, na druhú dáme všetky ostatné.
No a vidíme, že namiesto pôvodnej komplikovanej otázky mohol Janko jednoducho zobrať prvú kôpku jabĺk, ukázať ju Peťke a opýtať sa: „Je niektoré z týchto jabĺk červené?“ Získal by tak presne tú informáciu ako zo svojej pôvodnej otázky. V celom vzorovom riešení preto budeme používať len otázky tohto typu.

Lineárne vyhľadávanie

Peťka môže len raz klamať. Čo sa teda stane, ak jej položíme tú istú otázku dvakrát? Ak oba razy dostaneme tú istú odpoveď, vieme, že táto odpoveď je skutočne správnou odpoveďou na našu otázku. No a naopak, ak dostaneme raz odpoveď „áno“ a raz odpoveď „nie“, vieme, že pri jednej z nich Peťka klamala. Nevieme síce ešte, aká je skutočná odpoveď na otázku, ktorú sme sa práve dvakrát spýtali, ale vieme niečo omnoho silnejšie – to, že odteraz musí Peťka na všetky nasledujúce otázky odpovedať pravdivo.
Základná korektná stratégia teda môže vyzerať napríklad nasledovne: Janko bude postupne jedno po druhom brať do ruky jednotlivé jablká. Každé ukáže Peťke a dvakrát sa jej opýta, či je červené. Ak niekedy dostane dvakrát „áno“, má červené jablko. Ak niekedy dostane „áno“ a „nie“, prichytil Peťku pri klamaní a odteraz sa na každé jablko (vrátane toho, ktoré práve drží) bude pýtať len raz.
Ak by napríklad Peťka klamala hneď vo svojej prvej odpovedi, po prvých dvoch otázkach bude Janko vedieť, že klamala, ale ešte nebude nič vedieť o jablkách. Následne by mu už stačilo 999 ďalších otázok – ak sa postupne o 999 jablkách dozvie, že sú zelené, to posledné musí byť červené.
Najhorší možný prípad pre túto stratégiu je, že až posledné jablko, ktoré Janko zoberie, bude červené, a až pri predposlednom jablku Peťka raz zaklame. Takto Janko 999× položí dve otázky a na záver potrebuje ešte jednu,

aby zistil, či je teda predposledné jablko zelené alebo červené. Dokopy teda máme 1999 otázok.

Binárne vyhľadávanie

Samozrejme, Janko môže používať omnoho efektívnejšie stratégie. Keby Peťka stále hovorila pravdu, Jankovou najlepšou stratégiou by bolo použiť *binárne vyhľadávanie*. Pri každej otázke by stačilo rozdeliť jablká na dve približne rovnako veľké kôpky a potom sa Peťky opýtať, či je jablko na jednej z nich. Každou otázkou by tak vylúčil polovicu jabĺk. Pre 1000 jabĺk (a tiež ľubovoľný väčší počet až po $1024 = 2^{10}$ jabĺk) by Jankovi stačilo 10 otázok.
Túto istú stratégiu môžeme použiť aj v našej úlohe, len samozrejme pridáme techniku, ktorú sme objavili vyššie: kým nevieme o klamstve, každú otázku položíme Peťke dvakrát.
Najhorším možným prípadom je zjavne situácia, kedy Peťka bude klamať až keď Jankovi ostanú posledné dve jablká. Vtedy Janko položí postupne 21 otázok: prvých 20 na simuláciu binárneho vyhľadávania a poslednú na záverečné rozlíšenie medzi poslednými dvoma jablkami.

Vo zvyšku tohto textu si ukážeme dva možné pohľady na vzorové riešenie. V oboch prípadoch budeme úlohu riešiť pre 1024 jabĺk namiesto 1000, ušetríme si tak starosti so zaokrúhľovaním. (Janko si teda okrem 1000 reálnych jabĺk predstaví 24 virtuálnych. Tie nemôžu byť červené, takže je jedno, či ich Peťke ukazuje alebo nie.)

Vzorové riešenie, pohľad prvý

Jeden možný pohľad na našu úlohu je nasledovný: pre každé jablko nás zaujíma, koľkokrát nám o ňom Peťka tvrdila, že nie je červené. Akonáhle sa to o nejakom jablku dozvieme dvakrát, môžeme ho zahodiť – aspoň jeden z tých dvoch výrokov musel nutne byť pravdivý.
Jablká teda budeme deliť na tri kôpky. Na prvej sú tie, o ktorých zatiaľ Peťka vždy tvrdila, že sú červené, na druhej sú tie, pre ktoré už raz povedala, že sú zelené, no a tretia (odpadová) kôpka obsahuje ostatné jablká – tie, ktoré už sú naisto zelené.
Každú možnú situáciu teda môžeme popísať dvoma číslami: počet jabĺk na prvej a počet jabĺk na druhej kôpke. Na začiatku sme v situácii $(1024, 0)$: všetky jablká zatiaľ môžu byť červené.
Nech si teraz Janko ako prvú otázku vyberie x jabĺk a opýta sa, či je červené medzi nimi. Ak Peťka odpovie „áno“, dostaneme sa do situácie $(x, 1024 - x)$. (Všetky ostatné jablká ešte môžu byť červené, ale len ak Peťka práve klamala.) Ak Peťka odpovie „nie“, nastane presne opačný prípad a dostaneme sa do situácie $(1024-x, x)$. Ak by sme začali rozdelením jabĺk presne napoly, bez ohľadu na to, ako Peťka odpovie, sa dostaneme do situácie $(512, 512)$.
Pokračujme teraz v delení jabĺk na polovice, a to nasledovne. Pri druhej otázke si vyberieme 256 jabĺk z prvej a 256 jabĺk z druhej kopy a opýtame sa Peťky, či je červené jablko medzi nimi. Bez ohľadu na to, ako odpovie, dostaneme sa do situácie $(256, 512)$. Rozmyslite si, prečo: Na druhej kope budeme mať 256 jabĺk, ktoré Peťka už po druhýkrát označí za zelené. Tie rovno zahodíme. Ostatné jablká z druhej kôpky na nej ostanú. No a podobne na prvej kôpke ostane 256 jabĺk (tie, ktoré podľa Peťkinej aktuálnej odpovede môžu byť červené) a 256 jabĺk presunieme z prvej kôpky na druhú.
Aj v ďalšom kroku rozdelíme jablká na polovice. Na prvej kôpke nám ich ostane 128, na druhej kôpke ich ostane 256 a plus k nim presunieme 128 z prvej kôpky. Po tretej otázke teda budeme v situácii $(128, 384)$.
V rovnakom duchu prebehne štvrtá až desiata otázka. Postupne sa takto dostaneme do nasledujúcich situácií: $(64, 256)$, $(32, 160)$, $(16, 96)$, $(8, 56)$, $(4, 32)$, $(2, 18)$ a po desiatej otázke budeme v situácii $(1, 10)$.
Máme teda zaručene práve jedno jablko, ktoré je červené, ak Peťka nikdy neklamala, a presne 10 jabĺk, z ktorých každé je červené, ak Peťka už práve raz klamala.
Potrebujeme medzi týmito jablkami nájsť to jedno naozaj červené a ostávajú nám na to posledné štyri otázky. Teraz už nevieme položiť otázku, ktorá bude úplne symetrická, Peťkine odpovede nám preto vyrobia dve rôzne situácie.
V jedenástej otázke ukážeme Peťke jablko z prvej kôpky a tri jablká z druhej. Ak Peťka odpovie, že červené je medzi nimi, zahodíme zvyšné jablká a ostaneme v situácii $(1, 3)$. Ak Peťka odpovie, že sú všetky zelené, dostaneme sa do situácie $(0, 8)$: na druhej kôpke nám ostalo 7 jabĺk, na ktoré sme sa nepýtali, a pribudlo tam

jablko, ktoré bolo doteraz na prvej kôpke.
Ak sme v situácii $(0, 8)$, vieme, že Peťka už klamala – žiadne jablko nie je konzistentné so všetkými jej odpoveďami. Môžeme teda použiť štandardné binárne vyhľadávanie a na tri otázky zistiť, ktoré z našich 2^3 jabĺk je to červené.
Ostáva nám teda situácia $(1, 3)$ a tri otázky na jej vyriešenie. Jedna možnosť, ako naše riešenie dokončiť, je opýtať sa Peťky na jediné jablko: to z prvej kôpky. Kladná odpoveď znamená, že máme naše červené jablko, záporná odpoveď nás dostane do situácie $(0, 4)$ a tú už na dve otázky doriešime.

Vzorové riešenie, pohľad druhý

Očíslujme si jablká číslami od 0 po 1023. Peťke postupne položíme 10 otázok. V otázke i jej ukážeme tých 512 jabĺk, ktorých čísla majú nastavený i-ty bit. (Prvá otázka sú teda jablká 1, 3, 5, 7, 9, ..., druhá otázka sú jablká 2, 3, 6, 7, 10, 11, ... a tak ďalej.)
Vždy, keď nám Peťka odpovie „áno“, zapíšeme si bit 1, a keď nám odpovie „nie“, zapíšeme si 0.
Takto dostaneme bitový zápis čísla jediného jablka, ktoré zodpovedá všetkým 10 odpovediam. Buď je toto naše červené jablko (a Peťka ešte neklamala), alebo je práve jeden z týchto 10 bitov zle. Rovnako ako v predošlom riešení sme sa teda dostali do situácie $(1, 10)$.

A-II-3 Turbojazdec

Podobne ako v domácom kole budeme úlohu riešiť pomocou techniky dynamického programovania. Postupne pre každý počet ťahov i a každé súradnice (j, k) vypočítame hodnotu $P[i, j, k]$: počet spôsobov, ktorými sa vieme dostať na políčko (j, k) s tým, že sme zatiaľ navštívili v správnom poradí prvých i písmen slova w.

Niektoré tieto hodnoty sú zjavné. Označme $S[j, k]$ písmeno na políčku (j, k) na šachovnici. Ak $S[j, k] \neq w[i]$, tak zjavne $P[i, j, k] = 0$. Slovne: po preskákaní prvých i písmen slova w vieme byť na políčku len vtedy, ak obsahuje i-te písmeno slova w.
Tiež je zjavné, že ak $S[j, k] = w[1]$, tak $P[1, j, k] = 1$: jediný spôsob, ako navštíviť prvé písmeno slova w, je začať na ňom.

No a ako vypočítame tie zvyšné hodnoty $P[i, j, k]$? Skončiť po i ťahoch na políčku (j, k) vieme tak, že spravíme prvých $i - 1$ ťahov, tými sa dostaneme na nejaké iné políčko a z toho následne i-tym ťahom prídeme na políčko (j, k). Spôsoby, ktoré nás tam dovedú z rôznych políčok, sú zjavne navzájom rôzne. Celkový počet spôsobov teda dostaneme jednoducho tak, že sčítame počty spôsobov pre všetky možnosti posledného ťahu.
Formálne, $P[i, j, k]$ vieme spočítať ako súčet všetkých $P[i - 1, j', k']$ takých, že turbojazdec vie jedným ťahom prejsť z (j', k') na (j, k).
Riešením našej úlohy je potom súčet všetkých hodnôt $P[n, \star, \star]$, kde n je dĺžka slova w.

Na veľkej šachovnici sa turbojazdec vie dostať skoro odvšadiaľ všade: ak sa políčka v každej súradnici líšia aspoň o 3, určite sa medzi nimi vie turbojazdec hýbať, a teda na každé políčko vieme prísť z aspoň $(r-5)(s-5)$ iných.
Priamy výpočet jednej hodnoty $P[i, j, k]$ postupným prezretím všetkých $P[i - 1, j', k']$ teda bude mať časovú zložitosť $\Theta(rs)$, z čoho dostávame celkovú časovú zložitosť $\Theta(nr^2s^2)$.

Lepšie riešenie

Vyššie popísané zlepšenie vieme ešte zefektívniť, a to tak, že budeme šikovnejšie sčitovať možnosti, ako sa na políčko dostať. Časovú zložitosť vieme šikovným sčitovaním zlepšiť až na $O(nrs)$.
Naším hlavným nástrojom budú dvojrozmerné prefixové súčty. Pomocou tejto techniky vieme tabuľku $r \times s$ čísel v čase $O(rs)$ predspracovať a následne budeme vedieť v konštantnom čase zistiť súčet ľubovoľného obdĺžnika. Detaily tejto techniky nájdete popísané tu: https://www.ksp.sk/kucharka/2d_prefixove_sumy/
Vždy, keď vypočítame všetky hodnoty $P[i, \star, \star]$ pre konkrétne i, zoberieme tabuľku týchto hodnôt, predpočítame pre ne dvojrozmerné prefixové súčty a pomocou tých budeme následne efektívnejšie počítať hodnoty $P[i+1, \star, \star]$.

Pozrime sa poriadnejšie na to, odkiaľ všade sa vie turbojazdec dostať na konkrétne políčko `X`. Na schéme nižšie sú všetky tieto políčka označené písmenami.

```
aaaaaaaab...deeee
aaaaaaaab...deeee
aaaaaaaab...deeee
cccccccc.....ffff
..................
..........X.......
..................
ggggggggg.....jjjj
hhhhhhhhi...kllll
```

Vo všeobecnosti vieme oblasti, odkiaľ sa vie turbojazdec dostať na políčko `X`, rozdeliť na nanajvýš 12 obdĺžnikov. V schéme je každý obdĺžnik iným písmenom. Ak vieme v konštantnom čase zistiť súčet ľubovoľného obdĺžnika, vieme teda v konštantnom čase zistiť celkový počet spôsobov, ako sa vie po ďalšom ťahu dostať turbojazdec na toto konkrétne políčko `X`.
Iný pohľad na tento istý výpočet je taký, že ide o nanajvýš štyri obdĺžniky, len od každého musíme ešte odčítať to jeho rohové políčko, ktoré je najbližšie ku turbojazdcovi.

Listing programu (Python)

```
def vnutri(R, C, r, c):
    return 0 <= r < R and 0 <= c < C

def sucet(PS, r1, c1, r2, c2):
    r2, c2 = r2+1, c2+1
    return PS[r2][c2] - PS[r2][c1] - PS[r1][c2] + PS[r1][c1]

R, C = [ int(_) for _ in input().split() ]
W = input()
board = [ input() for r in range(R) ]

# zistime pocet sposobov (0 alebo 1) pre prve pismeno
P = [ [ int( board[r][c] == W[0] ) for c in range(C) ] for r in range(R) ]

# postupne pre kazde dalsie pismeno spravime prefixove sucty
# a pomocou nich zistime nove pocty
for n in range(1,len(W)):
    PS = [ [ 0 for c in range(C+1) ] for r in range(R+1) ]
    for r in range(R):
        for c in range(C):
            PS[r+1][c+1] = PS[r+1][c] + PS[r][c+1] - PS[r][c] + P[r][c]

    newP = [ [ 0 for c in range(C) ] for r in range(R) ]
    for r in range(R):
        for c in range(C):
            if W[n] != board[r][c]: continue
            if vnutri(R, C, r-2, c-2):
                newP[r][c] += sucet(PS, 0, 0, r-2, c-2) - P[r-2][c-2]
            if vnutri(R, C, r-2, c+2):
                newP[r][c] += sucet(PS, 0, c+2, r-2, C-1) - P[r-2][c+2]
            if vnutri(R, C, r+2, c-2):
                newP[r][c] += sucet(PS, r+2, 0, R-1, c-2) - P[r+2][c-2]
            if vnutri(R, C, r+2, c+2):
                newP[r][c] += sucet(PS, r+2, c+2, R-1, C-1) - P[r+2][c+2]

    P = newP

print( sum( sum(row) for row in P ) )
```

A-II-4 Pokazený rover kóduje čísla

Riešenia podúloh uvádzame v rovnakom poradí ako v zadaní.

Podúloha A: nájdi predpis funkcie

V našom poradí máme všetky dvojice (x, y) usporiadané podľa ich súčtu. Dvojice s rovnakým súčtom sú následne usporiadané podľa x.
Poradové číslo konkrétnej dvojice je rovné počtu iných dvojíc, ktoré sú v poradí pred ňou.
Pred dvojicou (x, y) sú všetky dvojice, ktoré majú súčet menší ako $x + y$: jedna dvojica so súčtom 0, dve so súčtom 1, ..., až $x + y$ dvojíc, ktoré majú súčet $x + y - 1$. Takýchto dvojíc je teda dokopy

$$\Big(1 + 2 + \cdots + (x + y)\Big) = \frac{(x + y)(x + y + 1)}{2}$$

Pred dvojicou (x, y) sú následne ešte dvojice s rovnakým súčtom ale menším prvým prvkom. Tých je x (prvý prvok nadobúda hodnoty od 0 po $x - 1$). Dokopy teda platí:

$$spoj(x, y) = \frac{(x + y)(x + y + 1)}{2} + x$$

Podúloha B: naprogramuj funkciu spoj

Makro už ľahko naskladáme pomocou existujúcich príkazov: do pomocných lokalít si uložíme hodnoty $x + y$ a $x + y + 1$, tie vynásobíme, výsledok vydelíme dvoma a pridáme x.

```
MAKRO spoj X Y Z
    zapis X [X_plus_Y]
    pridaj Y [X_plus_Y]
    zapis [X_plus_Y] [X_plus_Y_plus_1]
    pridaj J [X_plus_Y_plus_1]
    vynuluj [tmp]
    vynasob [X_plus_Y] [X_plus_Y_plus_1] [tmp]
    vynuluj [dva]
    pridaj J [dva]
    pridaj J [dva]
    vydel [tmp] [dva] Z [zahod_zvysok]
    pridaj X Z
END
```

Podúloha C: naprogramuj funkciu prvy

Keby sme mali nájsť matematický predpis tejto funkcie, asi by sme sa celkom zapotili a zrejme by v ňom figurovala nejaká odmocnina a nejaké celé časti. To by sa roveru pomocou kamienkov nerobilo zrovna jednoducho. Našťastie nič také nepotrebujeme spraviť. A hlavným kľúčom k úspechu je nehľadať *efektívne* riešenie, ale *funkčné a čo najjednoduchšie* riešenie.
Začneme jednoduchým pozorovaním. Pred dvojicou (x, y) sú určite všetky dvojice $(0, 0)$, $(1, 0)$, ..., $(x - 1, 0)$, ktorých je dokopy x. Preto $spoj(x, y) \geq x$. Analogicky, $spoj(x, y) \geq y$.
Ak teraz dostaneme hodnotu $z = spoj(x, y)$ a hľadáme hodnoty x, y, vieme, že $x \leq z$ a zároveň $y \leq z$. To ale znamená, že máme len konečný počet možností a teda ich môžeme všetky prezrieť.
V pseudokóde bude teda naša implementácia tejto funkcie vyzerať nasledovne:

```
funkcia prvy(z):
    pre všetky x od 0 po z:
        pre všetky y od 0 po z:
            ak spoj(x,y) == z:
                vráť hodnotu x
```

Tento program teraz potrebujeme zapísať v jazyku, ktorému rozumie rover. Budeme teda potrebovať dva do seba vnorené cykly.

```
MAKRO prvy Z X
    cyklus1: vynuluj [py]
```

```
    cyklus2: spoj [px] [py] [skusam]
             rovnake [skusam] Z odpoved
             skoc dalsi2
    odpoved: zapis [px] X

     dalsi2: rovnake Z [py] koniec2
             pridaj J [py]
             skoc cyklus2

    koniec2: rovnake Z [px] koniec1
             pridaj J [px]
             skoc cyklus1

    koniec1: cakaj
END
```

Vo vnútornom cykle si vždy pre aktuálne hodnoty x a y do pomocnej premennej spočítame $spoj(x, y)$ a porovnáme to so z. Keď nájdeme tú správnu dvojicu x a y (vždy je práve jedna), zapíšeme x do výstupnej premennej. Cyklus však pre jednoduchosť ani vtedy neprerušíme, necháme ho dobehnúť.
Vždy, keď hodnota y dosiahne z, vnútorný cyklus ukončíme a vrátime sa do vonkajšieho cyklu. Tam inkrementujeme x, vynulujeme y a začneme novú iteráciu vnútorného cyklu – alebo skončíme, ak sme už práve ukončili skúšanie možnosti $x = z$.
(Funkcia `druhy`, spomínaná v podúlohe D, by sa od funkcie `prvy` líšila len v jednom riadku: `odpoved: zapis [py] Y`.)

Podúloha D: dĺžka postupnosti

Majme číslo z, predstavujúce kód postupnosti. Ak je toto číslo nula, ide o kód prázdnej postupnosti, a tá má dĺžku 0.
Ak je z kladné, vieme, že platí $z = 1 + spoj(x_0, k)$, kde x_0 je prvý člen našej postupnosti a k je kód jej zvyšku. To ale znamená, že $z - 1 = spoj(x_0, k)$, inými slovami, že $x_0 = prvy(z - 1)$ a $k = druhy(z - 1)$. Pomocou už implementovaných funkcií teda vieme z kódu neprázdnej postupnosti získať aj jej prvý člen, aj kód jej zvyšku. No a aká je vtedy dĺžka našej postupnosti s kódom z? O jedno väčšia ako dĺžka postupnosti s kódom k.
Túto úvahu nám už len stačí v cykle opakovať: postupne po jednom budeme odoberať členy postupnosti a zároveň si vo výstupnej premennej `x` počítať, koľko sme ich už odobrali. Keď nám zostane prázdna postupnosť, skončíme.

```
MAKRO dlzka Z X
             vynuluj X
             zapis Z [postupnost]
     cyklus: nulove [postupnost] koniec
             pridaj J X
             prenes [postupnost] J K -                # vypocitame si z-1
             druhy [postupnost] [zvysok_postupnosti] # z neho vypocitame druhy(z-1)
             zapis [zvysok_postupnosti] [postupnost] # a to je kod postupnosti co nam ostala
             skoc cyklus
     koniec: cakaj
END
```

TRIDSIATY SIEDMY ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Michal Forišek
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava 2021