B-I-1 Byrokracia

Na tejto úlohe bolo pekné, že sa dala riešiť mnohými rôznymi spôsobmi. Ukážeme si jedno pomalé a tri rozličné rýchle riešenia.
Všetky tieto riešenia sa nám budú lepšie vysvetľovať, ak si úlohu reprezentujeme pomocou grafov, takže začneme trochou teórie. Ak už grafy poznáte, môžete preskočiť až po sekciu „Riešenie simuláciou“.
Graf sa skladá z vrcholov a hrán, pričom v našom prípade *vrcholy* sú kancelárie a *hrany* sú spojenia medzi kanceláriami. Vrcholy zvykneme označovať malými písmenami, napríklad $u, v, w, u_1, u_i, \ldots$ Ak nás z kancelárie u pošlú do kancelárie v, tak hovoríme, že z vrcholu u vedie hrana do vrcholu v.
V tejto úlohe pracujeme s *orientovaným grafom* – každá hrana nášho grafu má *orientáciu*, čiže smer, ktorým vedie. Hrana vedúca z u do v teda nie je to isté ako hrana vedúca z v do u.
Viac o grafoch si môžete prečítať v tomto článku: https://www.ksp.sk/kucharka/grafy_uvod/.
Keďže v našej úlohe z každého vrcholu vedie len jedna hrana, môžeme si definovať nový pojem *krok* – presun po hrane. Napr. ak začneme vo vrchole u_0 a spravíme 5 krokov, tak skončíme vo vrchole u_5, ktorý je jednoznačne určený tým, že v našom grafe existujú vrcholy u_1, u_2, u_3, u_4 také, že z u_0 do u_1, z u_1 do u_2, z u_2 do u_3, z u_3 do u_4 a z u_4 do u_5 vedú hrany. Tieto vrcholy nemusia byť všetky rôzne. Napr. ak z vrcholu 1 vedie hrana do 2 a z vrcholu 2 vedie hrana do 1, tak z 1 sa na 5 krokov dostaneme do 2, keďže z vrcholu 1 postupne urobíme kroky do vrcholov 2, 1, 2, 1 a 2.

Riešenie simuláciou

Ak sa vrátime k pôvodnej úlohe, tak v reči grafov sa pýtame, koľko existuje takých vrcholov v, pre ktoré existuje neprázdna postupnosť krokov z v naspäť do v (čo budeme ďalej značiť aj $v \to v$).
Toto je ekvivalentné zadanej úlohe, lebo

- Ak neexistuje postupnosť krokov $v \to v$, tak nevieme počas jednej návštevy ministerstva navštíviť vrchol v opakovane a teda ani stokrát.
- Ak existuje postupnosť krokov $v \to v$, tak vieme počas jednej návštevy ministerstva navštíviť vrchol v aspoň stokrát a to napríklad tak, že začneme vo vrchole v a 99-krát zopakujeme postupnosť krokov, ktorá nás dostane naspäť do v.

Postupností krokov z v do v môže existovať viacero (napríklad zopakovaním $v \to v$), ale vždy zo všetkých možností vieme vybrať tú najkratšiu.
Dá sa ľahko dokázať, že v najkratšej postupnosti $v \to v$ sa žiadne vrcholy nezopakujú (okrem v na začiatku a na konci). Ak by sa opakoval nejaký vrchol u, máme postupnosť $v \to u \to u \to v$, ale potom by táto postupnosť nebola najkratšia, lebo $v \to u \to v$ je kratšia. Keďže sa v najkratšej postupnosti vrcholy neopakujú a vrcholov je najviac n, tak najkratšia postupnosť má najviac $n - 1$ krokov.
Ak existuje postupnosť krokov $v \to v$, budeme tiež hovoriť, že vrchol v sa nachádza na *cykle*.
Úloha zo zadania by sa teraz dala preformulovať aj nasledovne: „Pre zadaný orientovaný graf, v ktorom z každého vrcholu vychádza práve jedna hrana, spočítajte, koľko vrcholov leží na nejakom cykle.“

Jednoduché, ale pomalé riešenie by teda mohlo vyzerať nasledovne: Vyskúšame začať v každom vrchole. Pre každý vrchol odsimulujeme najviac n krokov. Ak počas týchto krokov navštívime začiatočný vrchol, tento vrchol sa nachádza na cykle, inak sa na cykle nenachádza.
Časová zložitosť tohto riešenia je $O(n^2)$, pre každý z n vrcholov odsimulujeme najviac n krokov.

Riešenie efektívnou simuláciou

Pomocou nasledujúceho pozorovania dokážeme simuláciu zrýchliť. Ak je vrchol v na cykle, a z v do u vedie hrana, tak aj u je na cykle. Jednoducho, ak $v, u \to v$ je cyklus, tak $u \to v, u$ je tiež cyklus. Čiže ak máme postupnosť krokov, ktoré postupne navštívia vrcholy $u_0, u_1, u_2, u_3, \ldots, u_k, u_0$, tak nielen u_0 je na cykle, ale aj každý z vrcholov $u_1, \ldots, u_k$ je na cykle.
Podobne, ak máme postupnosť vrcholov $u_0, u_1, \ldots, u_k$ a vieme, že u_k nie je na cykle, tak ani žiadne z $u_0, \ldots, u_{k-1}$ nie sú na cykle. Ak by napríklad u_i bol na cykle, tak by na cykle museli byť aj vrcholy u_{i+1}, u_{i+2} až u_k, čo je spor.

Ako príklad si môžeme predstaviť, že simulujeme kroky z vrcholu 7 a postupne navštívime vrcholy $7, 9, 6, 8, 5, 2, 8$. Prvý vrchol, ktorý je na cykle je vrchol 8. Z toho vieme, že vrcholy $8, 5, 2$ sú na cykle a vrcholy $7, 9, 6$ na žiadnom cykle nie sú. S touto znalosťou môžeme ušetriť veľa výpočtového času, pretože nemusíme púšťať simuláciu z vrcholov $9, 6, 8, 5, 2$.

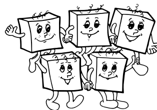
Túto myšlienku vieme využiť pri implementácii. Chceme upraviť riešenie simuláciou tak, aby sme nepúšťali hľadanie cyklu z vrcholov, o ktorých už poznáme odpoveď z predchádzajúcej simulácie. Vieme, že ak spustíme simuláciu z vrcholu v po najviac n krokoch sa nejaký vrchol zopakuje. Nech sa ako prvý zopakuje vrchol u. Naša simulácia postupne prešla cestou $v \to u \to u$. Z tejto simulácie vieme rovno povedať, že všetky vrcholy navštívené v časti $u \to u$ ležia na cykle a všetky vrcholy $v \to u$ (s výnimkou u) na cykle neležia. Z týchto všetkých vrcholov teda simuláciu púšťať nemusíme.
Vieme si však uvedomiť ešte jednu vlastnosť. Predstavme si, že začneme simulovať kroky z vrchola v a zrazu narazíme na vrchol w, ktorý sme spracovali v niektorej z predchádzajúcich simulácií. Uvedomme si, že v takom prípade môžeme o všetkých vrcholoch $v \to w$ rovno povedať, že sa na cykle nenachádzajú a v simulácii nepokračovať.
Totiž, ak sa vrchol w na cykle nenachádza, znamená to, že z neho vedie cesta, ktorá sa cyklí neskôr tvaru $w \to u \to u$. Pridaním našej novoobjavenej cesty $v \to w$ to vieme predĺžiť na $v \to w \to u \to u$, z toho však vieme, že vrcholy $v \to w$ na cykle nie sú.
A ak sa vrchol w nachádzal na cykle $w \to w$, tento cyklus nemôže obsahovať žiadne z vrcholov $v \to w$. V opačnom prípade by totiž tieto vrcholy už boli označené a vrchol w by nebol prvý skôr spracovaný vrchol na ktorý sme narazili. Dostávame tak cestu $v \to w \to w$, z čoho opäť usúdime, že vrcholy $v \to w$ neležia na cykle.

Všimnime si, že naša upravená simulácia musí vedieť spracovávať dva veľmi podobné stavy. V jednom narazí na *skôr navštívený vrchol* u, ktorý signalizuje cyklus $v \to u \to u$. V druhom narazí na *skôr navštívený vrchol* w, z čoho usúdi, že celá cesta $v \to w$ neleží na cykle. Rozdielom týchto dvoch stavov je, že u bol navštívený v *aktuálne bežiacej* simulácii a vrchol w v niektorej *predchádzajúcej* simulácii.
Aby sme tieto stavy rozlíšili, v programe použijeme premennú `loop_id`, ktorá sa zvýši pri každom spustení novej simulácie. Navštívenosť vrcholov si budeme značiť v poli `visited`. Nepoužijeme však iba hodnoty `true/false` (navštívený/nenavštívený). Prvok `visited[v]` bude 0 ak je vrchol v nenavštívený a ak už bol v navštívený, `visited[v]` si bude pamätať, v ktorom cykle sa to stalo (vtedajšia hodnota `loop_id`).
Keď narazíme na už navštívený vrchol (`visited[v] > 0`), ľahko rozlíšime, či sme ho navštívili v tejto alebo v skôr bežiacej simulácii.

Pri takejto simulácii prejdeme po každej hrane najviac dvakrát, vďaka čomu bude mať naše riešenie časovú zložitosť $O(n)$.

Listing programu (C++)

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

int n;
int A[2000047];
int visited[2000047];

int main() {
    scanf("%d", &n);
    for (int i = 0; i < n; ++i) {
        scanf("%d", A+i);
        A[i]--;
    }
    // do tejto premennej budeme pocitat prvky na cykloch
    int result = 0;

    for (int i = 0; i < n; ++i) {
        // simulaciu budeme pustat iba z vrcholov, ktore sme este nenavstivili
        if (visited[i]) continue;
        int loop_id = i+1;
        int k = i;

        // zacneme vo vrchole `k = i`, a postupujeme, az kym sa nam nezopakuje nejaky vrchol
        while (true) {
            // tento vrchol uz sme navstivili skor a pustali sme z neho simulaciu, takze mozeme tuto simulaciu skoncit
            if (visited[k] && visited[k] < loop_id) break;
```




```
            // zopakoval sa nam vrchol, takze aktualne `k` lezi na cykle
            if (visited[k] == loop_id) {
                // prejdeme cely cyklus este raz, a spocitame, kolko je na nom vrcholov
                int j = k;
                do {
                    result += 1;
                    j = A[j];
                } while (j != k);
                break;
            }

            // simulujeme krok na dalsi vrchol
            visited[k] = loop_id;
            k = A[k];
        }
    }
    printf("%d\n", result);
}
```

Riešenie so zamyslením

Označme $D(v, k)$ vrchol do ktorého sa dostaneme ak začneme vo vrchole v a spravíme k krokov. Napríklad, ak existuje postupnosť krokov u, v, w, tak $D(u, 0) = u$, $D(u, 1) = v$, $D(u, 2) = w$.
Prvé pozorovanie je, že $D(v, m)$ pre $m \geq n$ určite leží na nejakom cykle. Inak povedané, ak začneme vo vrchole v a spravíme aspoň n krokov, tak sa dostaneme do vrcholu u, ktorý už určite leží na nejakom cykle.

Dôkaz je nasledovný: Postupnosť m krokov navštívi aspoň $n+1$ vrcholov (rátajúc aj ten, v ktorom sme začali). Medzi týmito vrcholmi sa musí nejaký vrchol zopakovať (Dirichletov princíp). Nazvime si ľubovoľný jeden opakujúci sa vrchol w. Postupnosť navštívených vrcholov teda vyzerá nasledovne: $v \to w \to w \to u$. Z toho, že w sa zopakuje, vieme, že w leží na cykle. A z konca našej postupnosti vieme, že z w sa nejakým počtom krokov dostaneme do u. To ale znamená, že u musí byť tiež jedným z vrcholov na tom istom cykle ako w.

Druhé pozorovanie je, že ak máme dané ľubovoľné $m \geq n$ a u leží na nejakom cykle, tak existuje v, pre ktorý platí $D(v, m) = u$.

Dôkaz je jednoduchý: Pozrime sa na cyklus, na ktorom leží u. Jeden možný vrchol v s hľadanou vlastnosťou vieme nájsť tak, že z u spravíme m krokov po cykle, ale „proti smeru šípok“.

Spojením pozorovaní je veľké pozorovanie, že pre každé dostatočne veľké m ($m \geq n$) platí, že:

- Keď pre každý vrchol v nájdeme vrchol $D(v, m)$, všetky takto nájdené vrcholy ležia na cykloch.
- Každý vrchol, ktorý leží na nejakom cykle, pre nejaké v naozaj nájdeme.

Úlohu teda vyriešime tak, že si zvolíme konkrétne dostatočne veľké m a potom pre každý vrchol v spočítame $D(v, m)$. Dostaneme tak zoznam vrcholov, čo sú, ako sme dokázali vyššie, práve všetky vrcholy, ktoré ležia na nejakom cykle. Stačí len vyhádzať duplikáty a spočítať ich.

Pre efektívne počítanie $D(v, m)$, si ako m zvolíme mocninu dvojky. Pre každý vrchol poznáme $D(v, 1)$ – hrana zadaná na vstupe. Ostatné hodnoty spočítame opakovaním vzorca $D(v, 2i) = D(D(v, i), i)$. Vrchol vzdialený $2i$ od vrcholu v nájdeme tak, že sa pohneme o i krokov od vrcholu, ktorý je o i vzdialený od v.
V implementácii nižšie sme si zvolili $m = 2^{22} > 2\,000\,000$.

Toto riešenie má zložitosť $O(n \log n)$, čo nie je optimálne riešenie, ale s prehľadom stihne zbehnúť do pár sekúnd aj na veľkých vstupoch.[1] Pamäťová zložitosť je $O(n)$, pretože keď spočítame $D(v, 2i)$ pre všetky v, môžeme zabudnúť $D(v, i)$.

[1] Odkiaľ sa vzal ten logaritmus? Namiesto pevne zvolenej dostatočne veľkej konštanty by sme m mohli zvoliť tak, že nájdeme najmenšiu mocninu dvoch, ktorá je väčšia alebo rovná n. Toto je práve hodnota 2^k, kde k je nahor zaokrúhlená hodnota dvojkového logaritmu čísla n. No a náš program na nájdenie vrcholov tvaru $D(v, m)$ pre $m = 2^k$ potrebuje spraviť presne k (čiže približne $\log_2 n$) iterácií a v každej vypočítať nové pole dĺžky n zo starého.

Listing programu (Python)

```
_, D = input(), [int(x)-1 for x in input().split()]
for _ in range(22):
    # z D(v, i) spočítame D(v, 2i), v podstate D[v] := D[D[v]]
    D = [D[a] for a in D]
# set(D) vyhodí duplikáty v čase O(n), len() spočíta počet.
print(len(set(D)))
```

Riešenie s celkom iným zamyslením

Úplne iný pohľad na úlohu je nasledovný. Vrchol, do ktorého nevedie žiadna hrana, nie je na cykle.
Takéto vrcholy môžeme z grafu odstrániť. Keď ich odstránime, mohlo sa stať, že do nejakých nových vrcholov zrazu žiadna hrana nevedie. Tak odstránime aj tie a opakujeme, až kým sa nestane, že do každého vrchola vedie hrana.
Ak máme orientovaný graf, kde z každého vrchola vychádza *práve* jedna hrana a do každého vrchola vchádza *aspoň* jedna hrana, tak zároveň platí, že do každého vrchola vchádza *práve* jedna hrana. Musí to platiť, lebo celkový počet vchádzajúcich hrán musí byť rovnaký ako celkový počet vychádzajúcich hrán.
Takýto graf je potom nutne tvorený niekoľkými disjunktnými cyklami a ničím iným (skúste dokázať, prečo).
Riešenie je teda nasledovné. Kým sa dá, odstraňujeme z grafu vrcholy do ktorých nevedie hrana. Keď už nič ďalšie odstrániť nevieme, spočítame vrcholy, ktoré nám ostali.
Pre efektívnu implementáciu si pre každý vrchol pamätáme, koľko do neho vedie hrán a udržujeme si zoznam nespracovaných vrcholov, do ktorých žiadna hrana nevedie. Pri spracovaní vrcholu v sa pozrieme kam smeruje hrana z neho a tomu vrcholu o 1 zmenšíme počet do neho vedúcich hrán. Ak toto číslo klesne na nula, pridáme ho medzi nespracované vrcholy, aby bol neskôr z grafu odstránený.

Časová zložitosť tohto riešenia je $O(n)$ – každý vrchol najviac raz odstránime z grafu a pri každom odstránení aktualizujeme počet hrán pre jeden iný vrchol. Pamäťová zložitosť je $O(n)$.

Listing programu (Python)

```
n = int(input())
A = [int(x)-1 for x in input().split()]
B = [0 for _ in range(n)]

# v premennej `B[a]` si pamatame, kolko hran vedie do `a`
for a in A:
    B[a] += 1

# najdeme vsetky vrcholy do ktorych nic nevedie
to_remove = [a for a in range(n) if B[a] == 0]
while len(to_remove) > 0:
    # pocas toho, ako mazeme tieto vrcholy z grafu, aktualizujeme si B[] pre ostatne vrcholy
    b = to_remove.pop()
    B[A[b]] -= 1

    # ked zrazu do noveho vrcholu nevedie ziadna hrana, pridame ho do zoznamu
    if B[A[b]] == 0:
        to_remove.append(A[b])

# spocitame, kolko vrcholov nam ostalo v grafe. Tie, ktore sme odstranili maju B[a] == 0.
print(sum(B[a] > 0 for a in range(n)))
```

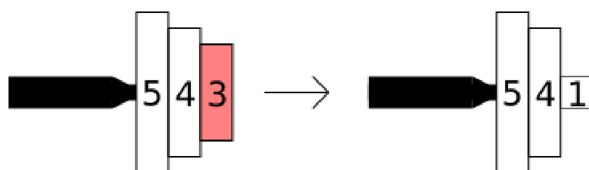
B-I-2 Činka

Na začiatok niekoľko zrejmých pozorovaní. Ak je požadovaná hmotnosť v menšia ako 20 kg, tak takú hmotnosť nevieme dosiahnuť – samotná tyč váži 20 kg. Ak je v nepárne, tak je tiež nedosiahnuteľné, lebo činka musí byť symetrická.
Ďalej sa budeme zaoberať len zostávajúcimi prípadmi. Stačí nám na jednej strane tyče dosiahnuť hmotnosť $\frac{v-20}{2}$ kg. Totiž, ak toto dosiahneme, tak na druhej strane tyče vieme dosiahnuť rovnakú hmotnosť rovnakými operáciami (len na opačnej strane), a dostať tak požadovanú hmotnosť $20 + 2 \cdot \frac{v-20}{2} = v$. Označme požadovanú hmotnosť polovice činky ako $v' = \frac{v-20}{2}$.

Zo symetrie tiež vyplýva, že na jednu stranu tyče smieme umiestniť každú hmotnosť najviac raz. Jednotlivé kotúče máme síce dvakrát, ale ak by sme oba kópie umiestnili na tú istú stranu, tak by nám nezostalo nič na druhú stranu.

Začíname s prázdnou činkou

Riešme teraz jednoduchší prípad, keď je začiatočná činka prázdna.
Označme si hmotnosti kotúčov na jednej strane naloženej činky $v_1 > v_2 > \ldots > v_i$. Každú hmotnosť od 1 po n vieme mať zastúpenú najviac raz, čo znamená, že maximálna dosiahnuteľná hmotnosť je $1 + 2 + \ldots + n$. Musí teda platiť, že $1 + 2 + \ldots + n \geq v'$. Otázka znie: vieme dosiahnuť každú takúto hmotnosť?
Vieme. Predstavme si totiž, že postupne pridávame na pol-činku kotúče n, potom $n-1$, ..., až po 1. Ak nikdy nepresiahneme požadovanú hmotnosť, tak musí platiť $v' = 1 + 2 + \ldots + n$ a práve sme túto hmotnosť dosiahli (tým, že sme uložili všetky kotúče, od najťažšieho po najľahší). Ak požadovanú hmotnosť niekedy presiahneme, tak sa pozrime detailnejšie na tento moment. Na tyči museli byť kotúče $n, n-1, \ldots, k$ a vtedy bola tyč ťažká nanajvýš v' kg. Po umiestnení kotúča $k-1$ ale túto hmotnosť prekročíme. To ale znamená, že chýbajúca váha na pol-činke bola menej ako $k-1$. Stačí nám teda pridať kotúč chýbajúcej hmotnosti, čím dosiahneme hmotnosť v'. (Prípadne nepridať žiaden kotúč, ak by si to vyžadovalo umiestnenie kotúča hmotnosti 0.)



Príklad vyššie uvedeného procesu, kde $n = 5$ a $v' = 10$. Po umiestnení kotúčov $5, 4$ a 3 máme hmotnosť 12 kg, čo je o 2 kg viac, ako chceme. Stačí nám potom namiesto posledného kotúča umiestniť kotúč o 2 kg ľahší, t.j. kotúč 1. Dostaneme tak pol-činku $5, 4, 1$.

Je toto ale najlacnejšie riešenie? Nevieme rovnakú hmotnosť tyče dosiahnuť presunutím menšej hmotnosti? Nevieme. Aby sme sa z hmotnosti 0 dostali na hmotnosť v', určite potrebujeme dokopy presunúť aspoň hmotnosť v'. No a v našom riešení je celková presunutá hmotnosť presne v', keďže nikdy nijaké kotúče neodoberáme, len postupne pridávame kotúče s celkovou hmotnosťou presne v'.

Listing programu (C++)

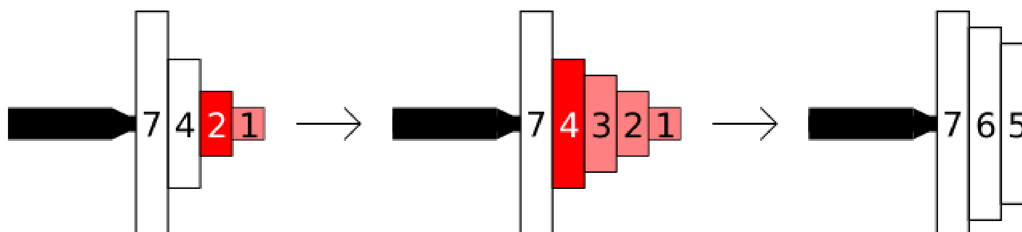
```
#include <iostream>
#include <vector>
using namespace std;

int main() {
    int n, v;
    cin >> n >> v;
    if (v % 2 == 1 || v < 20) {
        cout << "-1\n";
        return 0;
    }
    v = (v - 20) / 2;

    vector<int> polCinky;
    int aktualnaHmotnost = 0;
    for (int vi = n; vi >= 1; vi--) {
        if (aktualnaHmotnost + vi > v) {
            int naPridanie = v - aktualnaHmotnost;
            if (naPridanie > 0) {
                polCinky.push_back(naPridanie);
                aktualnaHmotnost += naPridanie;
            }
            break;
        }
        else {
            polCinky.push_back(vi);
            aktualnaHmotnost += vi;
        }
    }

    if (aktualnaHmotnost < v) {
        // vacsiu hmotnost nevieme dosiahnut
```

```
        cout << "-1\n";
        return 0;
    }

    cout << 2 * polCinky.size() << "␣" << 2 * aktualnaHmotnost << "\n";
    for (int vi : polCinky) {
        cout << "L␣+␣" << vi << "\n";
        cout << "R␣+␣" << vi << "\n";
    }

    return 0;
}
```

Časová zložitosť je $O(n)$ – pre každý z kotúčov v konštantnom čase overíme, či jeho pridanie prekročí požadovanú hmotnosť v'. Ak nie, tak ho pridáme do pol-činky (čo je opäť v konštantnom čase). Pamäťová zložitosť je $O(n)$, lebo si pamätáme kotúče na pol-činke kým ju zostrojujeme.

Všeobecné riešenie

Predstavme si teraz, že už na pol-činke máme kotúče $v_1 > v_2 > \ldots > v_i$. Ako najlacnejšie dosiahnuť hmotnosť v'? Určite musíme presunúť aspoň hmotnosť $\Delta v' = v' - v_1 - v_2 - \ldots - v_i$, a aby sme presunuli presne túto hmotnosť, musíme kotúče len pridávať (nesmieme ich odoberať).
Ak $\Delta v' < 0$, tak to zrejme nepôjde (cieľová hmotnosť je menšia ako začiatočná, takže určite musíme aspoň jeden kotúč odobrať). Ak $\Delta v' \geq 0$, tak to možno pôjde. Otázka je, či vieme dosiahnuť túto hmotnosť ak máme k dispozícii iba kotúče hmotnosti menšej ako v_i. Toto sme už ale riešili v predchádzajúcej časti: môžeme si predstaviť, že tieto kotúče postupne pridávame od najťažšieho po najľahší. Ak umiestnime všetky kotúče a stále sme ľahší ako $\Delta v'$, tak to nepôjde. V opačnom prípade sme možno prestrelili (máme ťažšiu pol-činku ako treba), tak posledný kotúč odoberieme a prípadne pridáme kotúč správnej hmotnosti.
Ak takto vieme dosiahnuť $\Delta v'$, tak sme vyhrali. V opačnom prípade určite musíme najkrajnejší kotúč (v_i) odobrať. To ale znamená, že sme práve našli nutný prvý krok optimálneho riešenia a teda ho môžeme rovno vykonať. Najkrajnejší kotúč teda odoberieme. Následne môžeme celú vyššie rozpísanú úvahu zopakovať s novou pol-činkou, na ktorej je o kotúč menej. Ak ani po odobratí všetkých kotúčov nevieme dosiahnuť požadovanú hmotnosť následným pridávaním, tak danú hmotnosť nevieme dosiahnuť.

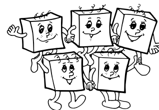


Príklad vyššie uvedeného procesu, kde počiatočný stav pol-činky je $v_1 = 7, v_2 = 4, v_3 = 2$ a chceme $v' = 18$.

Analyzujme časovú zložitosť:

- Ako prvé testujeme, či vieme dosiahnuť v' s tým, že iba pridávame na pol-činku $v_1, \ldots, v_i$. Toto trvá nanajvýš v_i krokov (lebo postupne pridávame nanajvýš $v_i - 1$ kotúčov).
- Ak ten test neuspel, odoberieme v_i a potom znova testujeme, či vieme dosiahnuť v' pridávaním kotúčov, pričom tentokrát ich už pridávame na pol-činku $v_1, \ldots, v_{i-1}$. Tento test trvá nanajvýš v_{i-1} krokov.
- Ak ani ten test neuspel, ...
- V najhoršom prípade sa dostaneme až k tomu, že odoberieme aj kotúč v_1 a potom ešte otestujeme, či vieme dosiahnuť v' pridávaním kotúčov na prázdnu činku. Tento posledný test má nanajvýš n krokov.

V každej úrovni spravíme nanajvýš n krokov a úrovní je n. Časová zložitosť je teda $O(n^2)$. Pamäťová zložitosť je $O(n)$.



Listing programu (C++)

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

// Vrati true ak sa naslo riesenie, inak false.
bool vyriesPolcinkuLenPridavanim(int maxKotuc, int deltaV, vector<int>& pridaneKotuce) {
    if (deltaV < 0) {
        return false;
    }

    int aktualnaHmotnost = 0;
    for (int vi = maxKotuc; vi >= 1; vi--) {
        if (aktualnaHmotnost + vi > deltaV) {
            int naPridanie = deltaV - aktualnaHmotnost;
            if (naPridanie > 0) {
                pridaneKotuce.push_back(naPridanie);
                aktualnaHmotnost += naPridanie;
            }
            break;
        }
        else {
            pridaneKotuce.push_back(vi);
            aktualnaHmotnost += vi;
        }
    }

    if (aktualnaHmotnost < deltaV) {
        // vacsiu hmotnost nevieme dosiahnut
        pridaneKotuce.clear();
        return false;
    }

    return true;
}

int main() {
    int n, v;
    cin >> n >> v;
    if (v % 2 == 1 || v < 20) {
        cout << "-1\n";
        return 0;
    }
    v = (v - 20) / 2;

    int k;
    cin >> k;

    vector<int> stav;
    int aktualnaHmotnost = 0;
    for (int ki = 0; ki < k; ki++) {
        int vi;
        cin >> vi;
        stav.push_back(vi);
        aktualnaHmotnost += vi;
    }

    int pocetOdobranychKotucov = 0;
    int presunutaHmotnost = 0;
    vector<int> pridaneKotuce;
    while (true) {
        int maxKotuc = ( stav.empty() ? n : (stav.back() - 1) );
        if (vyriesPolcinkuLenPridavanim(maxKotuc, v - aktualnaHmotnost, pridaneKotuce)) {
            for (int kotuc : pridaneKotuce) {
                presunutaHmotnost += kotuc;
            }
            break;
        }

        if (stav.empty()) {
            // ani z prazdnej cinky to nevieme dosiahnut
            cout << "-1\n";
            return 0;
        }

        pocetOdobranychKotucov++;
        presunutaHmotnost += stav.back();
        aktualnaHmotnost -= stav.back();
        stav.pop_back();
    }

    cout << 2 * (pocetOdobranychKotucov + pridaneKotuce.size()) << "␣" << 2 * presunutaHmotnost << "\n";
    for (int i = 0; i < pocetOdobranychKotucov; i++) {
        cout << "L␣-\n";
        cout << "R␣-\n";
    }
    for (int kotuc : pridaneKotuce) {
```

```
        cout << "L␣+␣" << kotuc << "\n";
        cout << "R␣+␣" << kotuc << "\n";
    }

    return 0;
}
```

Ešte vieme spraviť jednu optimalizáciu časovej zložitosti. Keď testujeme, či vieme dosiahnuť nejaké $\Delta v'$ iba pomocou kotúčov menších ako v_j, pýtame sa vlastne na to, či $0 \leq \Delta v' \leq 1 + 2 + \ldots + (v_j - 1)$. Na hodnotu na pravej strane nerovnosti ale existuje vzorec: $1 + 2 + \ldots + k = \frac{k \cdot (k+1)}{2}$. Takže to vieme zistiť v čase $O(1)$ a riešenie skladať jedine vtedy, keď tento test prejde. Časovú zložitosť tak zredukujeme na $O(n)$ – lebo kotúče budeme pridávať na pol-činky jedine vtedy, keď už sme si istí, že to vedie k riešeniu.

```
    // ...

    // Vrati true ak sa naslo riesenie, inak false.
    bool vyriesPolcinkuLenPridavanim(int maxKotuc, int deltaV, vector<int>& pridaneKotuce) {
        if (deltaV < 0 || deltaV > maxKotuc * (maxKotuc + 1) / 2) {
            return false;
        }
        // ...
    }

    // ...
```

B-I-3 Ideme na Túru

Ako sme v zadaní naznačili, v tejto úlohe sa nedá robiť nič iné, ako skúšať všetky možnosti (s rôznymi úrovňami šikovnosti)[2].

Skúšame všetky možnosti

Prvá možnosť je vyskúšať naozaj všetky možnosti. Ako ste si prečítali v zadaní, existuje nanajvýš $10! = 10 \cdot 9 \cdot 8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$ možných okruhov – máme 10 možností kde začať, 9 možností, kam ísť potom, atď, až kým nám neostane posledné nenavštívené miesto, z ktorého sa vrátime na začiatok.
Každý okruh si vieme zapísať ako *permutáciu* čísel od 1 po 10, t.j. ako postupnosť čísel, v ktorej sa každé číslo od 1 po 10 objaví práve raz.
Postupnosti, to znie povedome! Môžete si spomenúť, že v zadaní ste dostali užitočný návod na prechádzanie cez všetky permutácie.
Takto dostaneme nasledovný algoritmus: pomocou vhodnej knižničnej funkcie si pre každý možný okruh (začínajúc okruhom $(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)$) vyskúšame, či sa ním dá prejsť (teda existujú prechody medzi každými dvoma susednými miestami), a koľko takéto prejdenie trvá. Vždy, keď bude aktuálny okruh kratší ako doteraz najkratší vyskúšaný okruh, aktualizujeme si doterajší najkratší okruh.

Keďže polia vo väčšine rozumných programovacích jazykoch začínajú nultým indexom, v implementácii sme si miesta prečíslovali od 0 po 9.

Listing programu (Python)

```
from itertools import permutations

n = int(input()) # keby sme chceli všeobecné n, inak n = 10
t = [list(map(int, input().split())) for _ in range(n)]

postupnost = [i for i in range(n)]

najlepsi = None

for permutacia in permutations(postupnost):
    cas = 0
    # spočítame, či sa okruh dá prejsť, a ak áno, za ako dlho
    for i in range(n):
        if t[permutacia[i - 1]][permutacia[i]] < 0:
```

[2]Ak by ste náhodou prišli na riešenie ktoré funguje v čase polynomiálnom od počtu vrcholov, čaká vás sláva a milión dolárov https://en.wikipedia.org/wiki/P_versus_NP_problem

```
            cas = -1
            break
        cas += t[permutacia[i - 1]][permutacia[i]]

    if cas >= 0 and (najlepsi == None or najlepsi[0] > cas):
        najlepsi = (cas, permutacia)

if najlepsi == None:
    print("ziadny okruh neexistuje")
else:
    print("{}\n{}".format(najlepsi[0], " ".join([str(a + 1) for a in najlepsi[1]])))
```

O niečo lepšie

Ako toto riešenie zrýchliť? Kľúčové pozorovanie je, že v predchádzajúcom riešení každý z okruhov skúšame až 10-krát.
Prečo? Pozrime sa na menší príklad, povedzme s troma miestami. Ak máme okruh $(3, 1, 2)$, tento okruh má vlastne rovnakú dĺžku ako okruhy $(1, 2, 3)$ a $(2, 3, 1)$. Ak totiž prechádzame okružnú trasu, náš celkový čas nebude závisieť od toho, kde začneme.
Okružnú cestu cez *všetkých* 10 miest teda vieme vždy začať na mieste číslo 1. (Každý iný okruh si vieme posunúť tak, aby začínal na mieste 1, a ako sme si vyššie rozmysleli, nezmeníme tým jeho celkový čas.)
Zrazu nás zaujíma len poradie ostatných deviatich miest. A týchto poradí je dokopy len 9!, takže budeme skúšať desaťkrát menej možností ako v pôvodnom riešení. Toto riešenie nám vie získať až osem bodov.

Listing programu (C++)

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

using namespace std;

int main() {
    int n;
    cin >> n;
    vector<vector<int> > t(n, vector<int>(n));
    for (int i = 0; i < n; i ++) {
        for(int j = 0; j < n; j ++)
            cin >> t[i][j];
    }

    // zacneme s P = {1,2,..., n-1}
    vector<int> P;
    for (int i = 1; i < n; i ++) P.push_back(i);

    vector<int> najlepsi_okruh = {};
    int najlepsi_cas = -1;

    do {
        // spocitame ci je P validny okruh, a ak ano, kolko trva
        if (t[0][P[0]] < 0 || t[P[n - 2]][0] < 0) continue;
        int cas = t[0][P[0]] + t[P[n - 2]][0];
        for (int i = 0; i < n - 2; i ++) {
            if (t[P[i]][P[i + 1]] == -1) {
                cas = -1;
                break;
            }
            cas += t[P[i]][P[i + 1]];
        }
        if (cas < 0) continue;

        // skontrolujeme, ci je P doteraz najlepsi okruh
        if (cas < najlepsi_cas || najlepsi_cas < 0) {
            najlepsi_okruh = P;
            najlepsi_cas = cas;
        }

    } while(next_permutation(P.begin(), P.end()));

    if (najlepsi_cas < 0) {
        cout << "ziadny okruh neexistuje\n";
    }
    else {
        cout << najlepsi_cas << endl;
        cout << 1;
        for (int i = 0; i < n - 1; i ++) cout << " " << najlepsi_okruh[i] + 1;
        cout << endl;
    }
}
```

Backtracking, alebo skúšame len všetky prípustné možnosti

Ako by sme toto riešenie mohli (v niektorých prípadoch) zlepšiť? Ako si ukážeme, stále sa bude jednať o skúšanie všetkých postupností, ale budeme to robiť trochu ináč. Riešenie budeme zostrojovať postupne, cestu po ceste, a budeme skúšať len také možnosti, ktoré sa *naozaj dajú prejsť*.
Čo tým myslíme? Pozrime sa napríklad na posledný vzorový vstup v zadaní, v ktorom je pomerne veľa neprejazdných chodníkov. Napríklad v ňom nevieme ísť priamo z miesta 1 do miesta 4. Všetky okruhy, ktoré začínajú $1, 4, \ldots$, sú teda neplatné! Takých okruhov je až 8!, čiže až devätina zo všetkých možností skúšaných v druhom riešení. A to nie je jediná neprejazdná priama cesta!
Ako by sme mohli neskúšať všetky postupnosti, o ktorých už z prvých krokov vieme naisto povedať, že nevedú k platnému riešeniu? Poďme opustiť knižničné funkcie a začať skúšať postupnosti trochu inak.
Začať stačí (ako sme si už vyššie ukázali) na mieste 1. Teraz máme 9 možností, kam ísť ďalej. Skúsime postupne všetky. Ak na našu aktuálnu voľbu druhého navštíveného miesta neexistuje z prvého miesta priama cesta, ďalej už túto možnosť neskúšame. Ak takáto cesta existuje, tak pokračujeme. Teda opäť vyskúšame všetky možnosti, kam ísť v ďalšom kroku, a opäť rovno zahodíme tie, do ktorých sa nevieme dostať.

Ako si to implementujeme? Pomocou *rekurzie*.
Budeme mať funkciu `najdi_okruh(postupnost)`, ktorej dáme začiatok okruhu, a ona nám vráti najkratší okruh ktorý začína touto postupnosťou. Vo funkcii si prejdeme všetky nenavštívené miesta (môžeme mať pomocnú premennú, v ktorej si pamätáme, ktoré vrcholy nie sú v aktuálnej postupnosti), a ak je i-te miesto nenavštívené a z posledného miesta v aktuálnej postupnosti existuje cesta do i, pridáme si i na koniec postupnosti a rekurzívne zavoláme funkciu `najdi_okruh(postupnost + [i])`.
Zo všetkých vyskúšaných okruhov vyberieme a vrátime ten najkratší z nich (ak nejaký existuje).
Ak funkciu zavoláme s postupnosťou, v ktorej sme navštívili všetky vrcholy, potom namiesto ďalšieho skúšania overíme, či je tento okruh naozaj okruh (teda či existuje aj cesta z posledného prvku postupnosti naspäť do prvého), a ak áno, spočítame si, koľko času práve zostrojený okruh zaberie.
Napokon, na začiatku nám stačí zavolať `najdi_okruh([0])` – keďže ako sme si už ukázali, ak nejaký okruh existuje, vždy vie byť posunutý, aby začínal v prvom mieste. (Pripomíname, že v programe miesta číslujeme od nuly.)
Takto naozaj vyskúšame všetky možnosti: ak nejaký okruh existuje, všetky cesty v okruhu musia byť prejazdné, takže sa k nemu pri skúšaní určite dostaneme.

Listing programu (Python)

```python
n = int(input()) # keby sme chceli všeobecné n, inak n = 10
t = [list(map(int, input().split())) for _ in range(n)]

# vrcholy ktoré nie sú v aktuálnom okruhu
pouzite = [False for _ in range(n)]

cnt = 0
cnt_dead_ends = 0

def najdi_okruh(postupnost):
    # ak sú v postupnosti všetky miesta, vyhodnotíme
    if len(postupnost) == n:
        cas = 0;
        for i in range(n):
            # ak nájdeme neprejazdnú cestu, tento okruh nefunguje
            if t[postupnost[i-1]][postupnost[i]] < 0:
                return None
            cas += t[postupnost[i-1]][postupnost[i]]
        return (cas, postupnost.copy())
    # inak hľadáme, kde ďalej
    najlepsi = None
    for i in range(n):
        if not pouzite[i]:
            if t[postupnost[-1]][i] >= 0:
                # našli sme prechodnú cestu na miesto,
                # ktoré ešte nie v postupnosti,
                # vložíme ju do postupnosti,
                # zapamätáme si, že sme ju použili
                postupnost.append(i)
                pouzite[i] = True
                vysledok = najdi_okruh(postupnost)
                # funkcia nám vráti najlepší okruh ktorý našla,
                # (alebo že taký neexistuje)
```

```
                # vrátime postupnost' do pôvodného stavu
                pouzite[i] = False
                postupnost.pop()
                # porovnáme výsledok s doteraz najlepšou postupnost'ou
                if vysledok != None:
                    if najlepsi == None:
                        najlepsi = vysledok
                    elif najlepsi[0] > vysledok[0]:
                        najlepsi = vysledok
    return najlepsi

pouzite[0] = True
vysledok = najdi_okruh([0])
if vysledok == None:
    print ("ziadny okruh neexstuje")
else:
    print("{}\n{}".format(vysledok[0], " ".join([str(a + 1) for a in vysledok[1]])))
```

Vieme tento prístup ešte ďalej zlepšiť? Zatiaľ sme sa pozerali len na to, či po okruhu vieme alebo nevieme prejsť. Vieme vylúčiť aj iné možnosti? Všimnime si, že nás zaujíma len **najrýchlejší okruh**. Predstavme si, že by sme si do funkcie `najdi_okruh()` posielali dva parametre: aktuálnu postupnosť miest a čas, za ktorý sa dá táto konkrétna postupnosť miest prejsť (bez toho, aby sme sa vracali na začiatok). Navyše si ešte niekde mimo rekurzie budeme pamätať, aký najrýchlejší kompletný okruh sme doteraz videli.
Ak máme práve zostrojenú postupnosť miest, ktorú už samú o sebe trvá prejsť dlhšie než celý aktuálny najkratší okruh, neoplatí sa ďalej skúšať: akýkoľvek okruh začínajúci touto postupnosťou miest musí trvať dlhší čas ako aktuálne najkratší okruh, takže nemôže byť riešením, ktoré hľadáme. Tieto postupnosti teda vieme rovno zahodiť!
Ako implementujeme toto riešenie? Podobne ako to predchádzajúce, len si niekde chceme pamätať, aké najlepšie riešenie sme videli. Jedna z možných implementácií (dole) si túto hodnotu pamätá v globálnej premennej.[3]
Inak je riešenie veľmi podobné tomu predchádzajúcemu: keď zavoláme funkciu `najdi_okruh(postupnost, cas)` prejdeme cez všetky doteraz nepoužité miesta, pozrieme, či nimi vieme predĺžiť postupnosť, a či prejdením tejto cesty bude táto postupnosť miest stále rýchlejšia ako doteraz najrýchlejší okruh. Ak áno, rekurzívne zavoláme `najdi_okruh()` pre predĺženú postupnosť.
Ako predtým, ak zavoláme funkciu s postupnosťou, v ktorej sme navštívili všetky vrcholy, tak namiesto ďalšieho skúšania overíme, či je tento okruh naozaj okruh (teda či existuje cesta späť do prvého prvku postupnosti). Ak áno, spočítame si, koľko času nám práve zostrojený okruh zaberie, a skontrolujeme, či sme práve nezlepšili doteraz najlepšie nájdené riešenie.

Listing programu (Python)

```
n = int(input()) # keby sme chceli všeobecné n, inak n = 10
t = [list(map(int, input().split())) for _ in range(n)]

# vrcholy ktoré nie sú v aktuálnom okruhu
pouzite = [False for _ in range(n)]
# (zatial') najlepší okruh a čas za ktorý ho prejdeme
najlepsi = None

def najdi_okruh(postupnost, cas):
    global najlepsi
    # ak sú v postupnosti všetky miesta, vyhodnotíme
    if len(postupnost) == n:
        if t[postupnost[-1]][postupnost[0]] < 0:
            return
        cas += t[postupnost[-1]][postupnost[0]];
        if najlepsi == None or cas < najlepsi[0]:
            najlepsi = (cas, postupnost.copy())
        return
    # inak hl'adáme, kde d'alej
    for i in range(n):
        if not pouzite[i]:
            if t[postupnost[-1]][i] >= 0:
                # našli sme prechodnú cestu na miesto,
                # ktoré ešte nie v postupnosti,
                # vložíme ju do postupnosti,
                # zapamätáme si, že sme ju použili
                postupnost.append(i)
                pouzite[i] = True
```

[3] Na rozdiel od predchádzajúceho algoritmu tu naša funkcia nič nevracia, doterajší najlepší možný okruh si pamätáme ako globálnu premennú. Keby ste ale chceli, mohli by ste program modifikovať tak, aby najlepší nájdený okruh vracal ako výsledok funkcie. A naopak, predchádzajúci program sa dá modifikovať tak, aby si výsledok rovno zapisoval do globálnej premennej.

```
                if najlepsi == None or cas + t[postupnost[-2]][i] < najlepsi[0]:
                    najdi_okruh(postupnost, cas + t[postupnost[-2]][i])
                # vrátime postupnosť do pôvodného stavu
                pouzite[i] = False
                postupnost.pop()

pouzite[0] = True
najdi_okruh([0], 0)
if najlepsi == None:
    print ("ziadny␣okruh␣neexstuje")
else:
    print("{}\n{}".format(vysledok[0], "␣".join([str(a + 1) for a in vysledok[1]])))
```

Ako rýchle sú tieto zlepšienia v skutočnosti? Pozrime sa na posledný vzorový vstup v zadaní.
Riešenie za osem bodov vyskúša v tomto prípade $9! = 362\,880$ možností.
Pri prvom zlepšení vyskúšame 67 872 okruhov a dostaneme sa do 40 341 slepých uličiek (teda postupností, ktoré zahodíme a ďalej neskúšame), teda počet prezeraných možností sme zmenšili približne na tretinu.
No a pri našom druhom zlepšení, kde prestaneme hľadať aj vtedy, ak nám už zostrojená časť okruhu trvá príliš dlho, vyskúšame iba 127 okruhov a dostaneme sa do 19 175 slepých uličiek. To je približne 20-násobné zlepšenie oproti skúšaniu všetkých 9! okruhov!
Samozrejme, takéto zrýchlenie nedostaneme vždy. Stále existujú vstupy, pre ktoré aj toto vylepšené riešenie postupne zostrojí a prezrie úplne všetkých 9! možných okruhov. Ale stále sa oplatí poznať a vedieť používať takéto optimalizačné techniky. Reálne cestné siete sú napríklad veľmi *riedke* – z každého miesta existujú cesty len na malý počet okolitých. V takýchto situáciách bude naše vylepšené riešenie omnoho efektívnejšie ako prezeranie úplne všetkých permutácií.

A už len poznámka na záver, táto technika rekurzívne skúšania všetkých možností (a odrezávania nevyhovujúcich) sa volá *backtracking* (po slovensky tiež niekedy *prehľadávanie s návratom*), a dá sa použiť na pomerne široké spektrum ťažkých problémov.

B-I-4 Svetielkové kódovanie

Našou prvou úlohou je vymyslieť ľubovoľné fungujúce kódovanie, ktoré umožní Alici a Hanke kódovať zadané časy stretnutia. Inšpirovať sa môžeme napríklad digitálnymi hodinkami. Tie predsa vedia zobrazovať ľubovoľný čas dňa pomocou displeja, ktorý je vlastne len mriežka pixelov, ktoré sa buď rozsvietia, alebo ostanú zhasnuté. Ak si povieme, že jedna cifra zaberá 3 stĺpce, medzi každými dvoma ciframi je medzera a medzi hodinami a minútami je pekná dvojbodka, potrebujeme $3+1+3+1+1+1+3+1+3 = 17$ stĺpcov, teda mriežku 17×17. Zakódovanie času `15:38` potom môže vyzerať nasledovne, pričom `#` zobrazuje rozsvietené svetielko.

```
# ###   ### ###
# #   #   # # #
# ###   ### ###
#   # #   # # #
# ###   ### ###
```

Takéto riešenie je však veľmi neefektívne. Pozrime sa na to, čo kódujeme zbytočne. Napríklad všetky dvojbodky a medzery síce robia kódovanie oveľa prehľadnejšie, zaberajú však zbytočne veľa miesta, ktoré *sa nikdy nemení.* A posielať informáciu, ktorá je stále rovnaká je niečo, čoho sa chceme pri kódovaní zbaviť, pretože ak je to nemenné, druhá strana si to vie doplniť. Ak by teda Alica poslala kód

```
##########
##    ## #
##########
#  #  ## #
##########
```

Hanka by si vedela sama doplniť medzery za každou trojicou stĺpcov a opäť prečítať pôvodnú správu. Akurát teraz nám stačí mriežka 12×12.

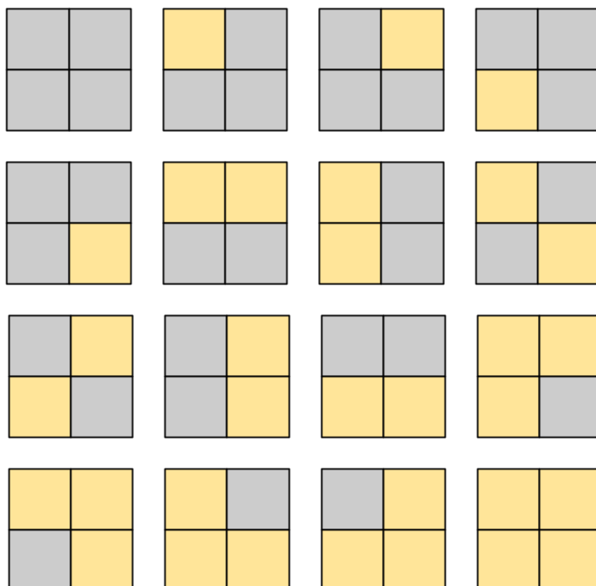
Ďalšia nemenná vec je prvá cifra. Keďže Alica s Hankou potrebujú kódovať iba časy od `14:00` po `18:00`, prvú cifru môžeme vynechať, lebo obe vedia, že to musí byť číslo 1.
Druhá cifra je tiež pomerne obmedzená, môže to byť buď 4, 5, 6, 7 alebo 8, čiže jedna z piatich možností. My na jej kódovanie použijeme až $3 \times 5 = 15$ svetielok. Namiesto toho by Alica s Hankou mohli zobrať prvý stĺpec mriežky a dohodnúť sa, že prvé svetielko bude svietiť ak je čas `14:XX`, druhé ak je čas `15:XX`, atď. až po piate svetielko reprezentujúce `18:XX`.
Na kódovanie nám zrazu stačí mriežka 7×7 a čas `15:38` by vyzeral nasledovne:

```
 ######
#  ## #
 ######
   ## #
 ######
```

Počítanie možností

V predchádzajúcej časti sme jednoduchý vizuálny spôsob kódovania postupne upravovali do efektívnejšej podoby. Stratili sme tým na čitateľnosti, tá však nikdy nebola dôležitá. Jediné, čo je dôležité je dohoda Alice a Hanky. A tie sú schopné si zapamätať ľubovoľné pravidlá.
Podobne, ako prvému stlpcu priradili význam podľa toho, ktoré svetielko sa rozsvieti, mohli by to spraviť pre ľubovoľný z časov. Napríklad sa môžu dohodnúť, že ak svieti prvé svetielko, znamená to čas `14:00`, ak druhé, je to čas `17:28`, ak tretie, tak `15:15`, a tak ďalej. Všimnite si, že v tom nepotrebujú mať dokonca žiaden systém, stačí, že sa dohodnú na jednotnej interpretácii.
A taktiež nemusia používať len jedno svetielko. Napríklad ak svieti celý prvý riadok, môže to znamenať čas `14:01`, ak svieti všetko okrem ľavého horného rohu, ide o čas `14:07`, a podobne.
Medzi `14:00` po `18:00` je dokopy $4 \cdot 60 + 1 = 241$ minút. Alici a Hanke teda bude stačiť, ak si ako pomôcku pre kódovanie a dekódovanie časov spravia veľkú tabuľku s 241 riadkami, pričom v každom riadku bude jeden z možných časov a jemu prislúchajúce rozsvietenie displeja.
A tu sa dostávame k *jednoznačnosti*. Uvedomme si, že každé z uvedených zobrazení displeja musí byť rôzne. Ak by totiž dva z nich vyzerali rovnako, Hanka by sa pri čítaní správy nevedela jednoznačne rozhodnúť, ktorý čas jej Alica zakódovala.

Zoberme si mriežku 2×2. Na obrázku nižšie sú zobrazené všetky možnosti, ako môže takáto mriežka vyzerať. Vidíme, že ich je iba 16. To je ale primálo na to, aby vedeli každému z 241 možných časov priradiť inak rozsvietený displej. Takýto malý displej im teda v žiadnom prípade nemôže stačiť.



Koľko možností zobrazenia má mriežka 3×3? Uvedomme si, že každé svetielko môže byť v dvoch rôznych stavoch – vypnuté alebo zapnuté. A tieto svetielka sú navzájom nezávislé. Pri jednom svetielku máme teda 2 možnosti. Pri dvoch svetielkach sú možnosti $2 \times 2 = 4$, pri troch ich je $2 \times 2 \times 2 = 8$, a tak ďalej. Každé ďalšie svetielko nám počet možností zdvojnásobí. Inými slovami, displej s n svetielkami sa dá rozsvietiť 2^n spôsobmi. V našom prípade máme 9 svetielok, počet rôznych rozsvietení je teda $2^9 = 512$. Keďže toto číslo je väčšie ako 241, takto veľká mriežka bude Alici a Hanke postačovať.
K riešeniu tejto úlohy dokonca nepotrebujeme ani priamo popísať nejaký konkrétny spôsob kódovania. Stačí nám povedať, že Alica s Hankou si vyberú spomedzi 512 možných rozsvietení mriežky 3×3 ľubovoľných 241 navzájom rôznych a každému priradia jeden z možných časov.
Na konci tohto vzorového riešenia však uvedieme aj jeden možný a konkrétny spôsob, akým by mohli k tomuto priradeniu pristúpiť.

Detekcia chyby

V poslednej podúlohe máme vymyslieť taký kód, ktorý navyše pomôže Alici a Hanke detegovať, ak nastane chyba na jednom svetielku a to sa rozsvieti ak malo ostať vypnuté alebo naopak.
Kód musí byť navrhnutý tak, aby Hanka vedela nejakým spôsobom určiť, či je prijatý kód v poriadku. Najjednoduchšou možnosťou je poslať správu v dvoch kópiách vedľa seba. Ak totiž nastane **jedna chyba**, tá sa objaví iba v jednej kópii. Hanka teda bude môcť porovnať obe kópie a ak sa nezhodujú, vie že sa prenos pokazil. (Všimnite si, že nevie zistiť, ktorá kópia je správna, jej však podľa zadania stačí len zistiť, že chyba nastala.)
Na zakódovanie 241 možností stačí 8 svetielok, pretože tie môžu byť rozsvietené $2^8 = 256$ rôznymi spôsobmi. Alica s Hankou si preto môžu zobrať mriežku 4×4 a dohodnúť sa, že prvé dva riadky budú kódovať čas (podľa ľubovoľnej konkrétnej, nimi vopred dohodnutej tabuľky) a zvyšné dva riadky budú kópiou tých prvých.
Keď Hanka rozvieti mriežku a zistí, že vrchné dva riadky sa nezhodujú so spodnými dvoma riadkami, vie, že nastala chyba. A ak sa zhodujú, odčíta z vrchných dvoch riadkov správny čas.

Pokúsme sa teraz vymyslieť šikovnejšie riešenie, ktoré nebude musieť na detekciu jedinej chyby prenášať celú druhú kópiu správy. Nestačil by nám na nejakú formu kontroly menší počet svetielok navyše? Ukáže sa, že áno. Stačilo by nám nájsť nejakú vlastnosť správy, ktorá sa *určite zmení* pokazením ľubovoľného jedného svetielka a Hanka ju bude vedieť skontrolovať.
Takou vlastnosťou je napríkad počet rozsvietených svetielok správy. Samozrejme, ten Hanka dopredu nepozná, keďže rôzne správy môžu rozsvietiť rôzny počet svetielok. S Alicou by sa však mohla dohodnúť, že každá poslaná správa sa bude skladať z dvoch častí – najskôr pomocou niekoľkých svetielok Alica zakóduje čas stretnutia a potom pomocou ďalších pošle Hanke informáciu o tom, koľko svetielok má v prvej časti svietiť.
Bez ohľadu na to, v ktorej časti nastane chyba, nebudú tieto dve časti navzájom korešpondovať a Hanka to bude vedieť zistiť.

No a to sme už len krok od vzorového riešenia. Ešte jednoduchšou vlastnosťou, ktorá sa pokazením jedného svetiela zmení, je **parita počtu rozsvietených svetielok**.
Spomínali sme si, že na poslanie času od `14:00` po `18:00` potrebujú Alica s Hankou 8 svetielok (lebo $2^8 \geq 241$). Pre tieto časy a 8 svetielok sa teda dohodnú na kódovaní rovnako ako v prvej podúlohe. V mriežke 3×3 majú ešte jedno svetielko navyše. Dohodnú sa, že to Alica nastaví tak, aby *celkový počet* rozsvietených svetielok na mriežke bol párny. Ak teda kód konkrétneho času používa nepárny počet svetielok, Alica rozsvieti aj toto deviate svetielko, inak ho nechá vypnuté.
Hanka preto vie, že korektne rozsvietený displej musí obsahovať párny počet rozsvietených svetielok. Ľubovoľná chyba jedného svetielka túto vlastnosť pokazí. Ak teda Hanka vidí nepárny počet rozsvietených svetielok, vie, že správa je pokazená.

Binárny kód

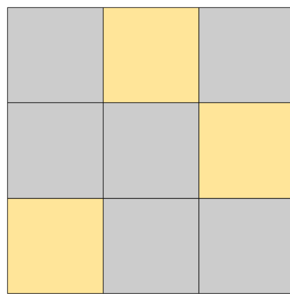
Jedným možným kódovaním, ktoré sa dá využiť na poslanie času, je binárny kód. Notoricky známe je, že počítače pracujú len s 0 a 1, čo je samozrejme obrovské zjednodušenie[4], nie však úplne nesprávne. Na úplne najnižšej

[4]Ktoré vyplýva z toho, že počítač skladáme z množstva súčiastok, ktoré vedia rozlišovať dva stavy – nízke (0) a vysoké (1) napätie.

úrovni počítače naozaj vedia efektívne pracovať s dvoma hodnotami.

Pri bežnom počítaní používame desiatkovú sústavu – jednotky, desiatky, stovky a tisícky sú mocniny čísla 10, pri zápise čísla využívame cifry 0 až 9 a pozícia cifry určuje použitú mocninu. Binárny kód je vlastne dvojková sústava, ktorá používa mocniny čísla 2 a cifry 0 a 1. Číslo 10011 v dvojkovej sústave preto vieme v desiatkovej sústave interpretovať ako $1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 19$.
Dá sa ľahko ukázať, že pomocou dvojkovej sústavy vieme jednoznačne reprezentovať ľubovoľné prirodzené číslo.

No a pre potreby Alice a Hanky je toto veľmi užitočné, lebo svetielka na ich mriežke sú veľmi dobré na reprezentovanie hodnôt 0 (vypnuté) a 1 (zapnuté). Alica s Hankou sa teda dohodnú, ktoré svetielko zodpovedá ktorej mocnine dvojky – napríklad po riadkoch zľava doprava začnúc 2^0 vľavo hore a pokračujúc až po 2^8 vpravo dole (pri mriežke 3×3. Vďaka tomu si vedia poslať v binárnej sústave zakódované ľubovoľné číslo od 0 po $2^9 - 1 = 511$. Následne sa dohodnú, že poslané číslo bude reprezentovať počet minút od `14:00`. Ak sa teda chcú stretnúť v čase `15:38`, musia poslať hodnotu 98 čo je v dvojkovej sústave 1100010. Ich mriežka by preto vyzerala nasledovne (všimnite si, že najpravejšia cifra je v našom kódovaní vľavo hore).



A ak by nielen prenášali čas ale chceli použiť aj vyššie popísanú techniku na detekciu možnej jednej chyby, musela by Alica ešte rozsvietiť aj svetielko v pravom dolnom rohu, aby dokopy svietilo párne veľa svetielok.

ŠTYRIDSIATY ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Truc Lam Bui, Ján Hozza, Paulína Smolárová
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: NIVAM – Národný inštitút vzdelávania a mládeže, Bratislava 2024