B-II-1 Plátanie ciest

Zo zadania vyplýva, že chceme vždy plátať najhlbšiu dieru. Začnime s najpriamočiarejším riešením: vždy prejdeme celé pole hĺbok dier, vyberieme tú najhlbšiu a odčítame z nej jeden centimeter. Po k zopakovaniach tohto procesu len vypíšeme výšku poslednej zaplátanej diery.
Pri hľadaní najhlbšej diery prechádzame poľom dĺžky n a toto musíme spraviť k krát. Výsledná časová zložitosť tohto riešenia je preto $O(nk)$, za čo sme vedeli získať nanajvýš 3 body.

Teraz sa musíme opýtať ťažkú otázku: „Dá sa to lepšie?“ Keď sa pozrieme na vyššie uvedené riešenie, sú v ňom dve podstatné časti: hľadanie najväčšieho prvku v poli (najväčšej diery) a odčítavanie od tohto prvku (plátanie diery). Pozrime sa na tieto dve časti jednotlivo.

Pri každom prejdení poľa hľadáme najväčší prvok (dokonca ktorýkoľvek z nich, pokiaľ ich je viac). Ak by sme presne vedeli, kde v poli sa nachádza najväčší prvok, vedeli by sme si ušetriť prechádzanie celého poľa. A v akom poli sa nachádza najväčší prvok na známom mieste? No predsa v *usporiadanom*! Ak si teda naše pole usporiadame (ideálne vzostupne), budeme mať určite na jeho konci najväčší prvok a teda ho budeme vedieť nájsť v čase $O(1)$ namiesto $O(n)$. Toto usporiadanie poľa nás bude síce stáť $O(n \log n)$ času, to je však menej ako $O(nk)$. Pozrime sa teraz na druhú časť našej skladačky.

V našom jednoduchom riešení odčítavame čísla po jednom. Predstavme si však, že máme nejaké množstvo dier hĺbky 1 a v strede jeden masívny kráter o hĺbke 10^9. V tomto prípade budeme naozaj veľmi dlho vyberať na plátanie tú istú dieru, náš kráter. Môžeme teda miesto plátania po jednom centimetri urobiť naraz viacero výjazdov.
Nech x je hĺbka najhlbšej diery. Koľkokrát po sebe budeme plátať túto dieru? No predsa kým sa jej hĺbka nevyrovná druhej najhlbšej diere. Ak je hĺbka druhej najhlbšej diery y tak vieme, že najbližších $x - y$ výjazdov bude smerovať k tej istej diere. Navyše, zistiť hĺbku druhej najhlbšej diery je stále veľmi jednoduché, je to predsa predposledná hodnota v našom usporiadanom poli.
Toto funguje dobre, ak máme jeden veľký kráter. Pri všeobecnom prípade nám však môže nastať situácia, že budeme striedať medzi skupinou dier, ktoré majú neustále rovnaké výšky. Predstavme si napríklad tri diery o veľkosti 10, teda pole $[10, 10, 10]$. Postupne budú hĺbky dier vyzerať následovne:

$$[10, 10, 10] \to [9, 10, 10] \to [9, 9, 10] \to [9, 9, 9] \to [8, 9, 9] \to \dots$$

Spomeňme si však, že v našej úlohe nás zaujíma iba hĺbka plátanej diery, nie to, ktorá diera bude plátaná. Ak teda máme viacero najhlbších dier hĺbky x, nech je takýchto dier p, tak najbližších p výjazdov iba zmenší hĺbku nejakej diery z x na $x - 1$. Navyše, po týchto p výjazdoch budeme mať p dier s hĺbkou $x - 1$.

Rozšírme si náš príklad ešte o jednu dieru navyše.

$$[7, 10, 10, 10] \to [7, 10, 10, 10] \to [7, 9, 9, 10] \to [7, 9, 9, 9] \to \dots \to [7, 7, 7, 8] \to [7, 7, 7, 7] \to \dots$$

Tento príklad nám ukazuje, že ešte viac výjazdov vieme spraviť naraz. Ak máme p najhlbších dier s hĺbkou x a ďalšia najhlbšia diera má hĺbku y, tak najbližších $p(x - y)$ výjazdov bude iba postupne meniť týchto p dier. No a po týchto výjazdoch sa diera hĺbky y „pripojí“ k našej skupine najväčších dier, keď zrazu budeme mať $p + 1$ dier hĺbky y.
Teraz si však opäť vieme nájsť ďalšiu najhlbšiu dieru s hĺbkou z a preskočiť toľko výjazdov, kým nebudeme mať $p + 2$ najhlbších dier s hĺbkou z.
Nášmu riešeniu teda stačí postupne zväčšovať počet dier v skupine najhlbších dier a naraz spracovávať všetky výjazdy po ďalšie zväčšenie tejto skupiny.

Ostáva nám to implementovať. Nech n je počet dier, hĺbky dier sú zapísané vo vzostupne usporiadanom poli `diery[]` a naša skupina najhlbších dier má veľkosť p (na začiatku 1).
Vieme, že v tejto skupine nutne musí byť posledných p dier z nášho poľa. Keďže je usporiadané, tak tieto diery boli na začiatku najhlbšie a iba oni sa mohli dorovnať. Navyše, v okamihu ako vznikne skupina p rovnako hlbokých dier je ich hĺbka rovná presne hodnote `diery[n - p]` – všetky väčšie diery dorovnali p-tu najhlbšiu dieru.

Do skupiny najhlbších dier sa ako ďalšia pridá diera na pozícii `diery[n - p - 1]` a to práve vtedy, keď sa všetkých p aktuálne najhlbších dier zmenší z hĺbky `diery[n - p]` na hĺbku `diery[n - p - 1]`. A to sa stane po presne $p \cdot (\texttt{diery}[\texttt{n} - \texttt{p}] - \texttt{diery}[\texttt{n} - \texttt{p} - 1])$ výjazdoch. Tento počet výjazdov teda odčítame od k a p zväčšíme o 1.
Tento postup opakujeme až do momentu, kedy už nemáme k dispozícii dostatočný počet výjazdov k. Zvyšné výjazdy ešte dokážu znížiť hĺbku každej aktuálne najhlbšej diery o k/p centrimetrov a ak ešte stále nejaké výjazdy ostali (je ich už menej ako p), niektoré z týchto dier sa zaplátajú ešte o jeden centimer. Z tohto vieme pomerne jednoducho spočítať výslednú hodnotu.
Aká je časová zložitosť tohto algoritmu? Pole si zoradíme v čase $O(n \log n)$. Následne ideme odzadu a postupne zväčšujeme veľkosť skupiny najhlbších dier. Vďaka šikovnému riešeniu viacerých výjazdov naraz spravíme každé takéto rozšírenie v konštatnom čase, na túto časť nám teda stačí $O(n)$ času. Finálna časová zložitosť je teda $O(n \log n)$ a toto stačí na vyriešenie ľubovoľného vstupu zo zadania.
Nasleduje Python-ové riešenie úlohy. Môžete z neho vyčítať rôzne implementačné detaily popísaného riešenia.

Listing programu (Python)

```
n, k = map(int, input().split())
diery = list(map(int, input().split()))

diery = list(sorted(diery))
# Pridame si dieru hlbky nula, aby sme mali na com zastavit
# a nevyskocili sme von z pola.
diery = [0] + diery
n += 1
# Ideme pole prechadzat sprava. Rovnako dobre sa to da naprogramovat
# aj zlava, ja to iba takto preferujem.
bezec = len(diery) - 1
while bezec != 0:
    # Prejdeme dolava az pokial nenarazime na dieru inej hlbky.
    while diery[bezec] == diery[bezec - 1]:
        bezec -= 1

    # Bezec teraz ukazuje na poslednu najvacsiu dieru.
    rozdiel_dier = diery[bezec] - diery[bezec - 1]
    pocet_dier = n - bezec
    potrebne_zaplaty = rozdiel_dier * pocet_dier

    if k >= potrebne_zaplaty:
        # Mame dost na zaplatanie az do hlbky mensej diery, pokracujeme ...
        k -= potrebne_zaplaty
    else:
        # Nemame dost na zaplatanie az po dalsiu dieru. Musime zistit,
        # ktora je posledna hlbka, ktoru sa nam podari zaplatat.

        # Vypocitame kolkokrat vieme zaplatat' 1cm na vsetkych dierach.
        cele = k // pocet_dier
        # Vypocitame, ci nam ostalo na zaplatanie iba zopar dier.
        zvysok = k % pocet_dier
        vysledna_hlbka = diery[bezec] - cele

        if zvysok > 0:
            # Ostalo nam trochu materialu na zapltanie casti dier, teda
            # niektore zo skupiny maju hlbku o 1cm mensiu.
            print(vysledna_hlbka - 1)
        else:
            # Zaplatali sme vsetky diery v skupine.
            print(vysledna_hlbka)
        break

    bezec -= 1
```

B-II-2 Chemická továreň

Simulácia

Skúsme sa pozrieť na jednoduchšiu verziu úlohy – ako by sme vedeli overiť, či **jeden** konkrétny pracovník dokáže uniknúť z továrne, **ak by sme poznali čas** t, v ktorom bol vyhlásený poplach?
Aby sme mohli simulovať pohyb tohto pracovníka, potrebujeme vedieť, kedy budú jednotlivé políčka továrne zaplavené kyselinou. Toto je v však rovnaký problém, ako sme riešili v domácom kole, akurát namiesto informácie

o zadaniach sa v našej mriežke šíri kyselina. Použiť teda môžeme prehľadávania do šírky (BFS), detaily necháme na čitateľa.
Ak poznáme správanie kyseliny, vieme opäť pomocou BFS zistiť, ako najrýchlejšie sa dokáže pracovník dostať zo svojho počiatočného políčka na ľubovoľné iné políčko. Iba do algoritmu musíme pridať podmienku, že nedovolíme pracovníkovi vojsť na políčko, na ktoré sa kyselina vie dostať skôr.
Pre riešenie úlohy s jedným pracovníkom a známym časom t nám stačí pustiť dve prehľadávania do šírky: raz pre kyselinu, raz pre pracovníka, a potom len overíme, či sa pracovník vedel dostať na políčko `U`.

Ak je pracovníkov viac, označme si ich počet p, jednoducho zistíme odpoveď pre každého zvlášť. Toto nám to bude trvať nanajvýš p-krát dlhšie.

Ak nevieme, kedy sa spustí poplach, môžeme celý algoritmus spustiť postupne pre čas $t = 0$, $t = 1$, $t = 2$, ... až kým nenájdeme prvú hodnotu t, pre ktorú sa uniknúť nedá.

Hľadanie najmenšieho času t, pre ktoré sa nedá uniknúť, však vieme robiť aj efektívnejšie. Otázka „Dá sa uniknúť, ak pustíme poplach v čase t?“, je ekvivalentná otázke „Dá sa uniknúť, ak pustíme poplach v čase t *alebo skôr*?“. Namiesto postupného overovania $t = 0, 1, 2, \ldots$ môžeme neznámu t binárne vyhľadať.
Keďže vieme, že t je niekde medzi 0 a rs, začneme hodnotou v strede. Ak sa z továrne dá uniknúť, keď sa alarm spustí v čase $\frac{rs}{2}$, tak ďalej budeme hľadať iba vo väčších hodnotách, ako ďalší budeme preto skúšať čas $\frac{3rs}{4}$. V opačnom prípade overujeme iba menšie hodnoty, pokračujeme teda testovaním času $\frac{rs}{4}$. Na $O(\log rs)$ otázok nájdeme týmto postupom hľadané t.
Takto vieme dostať algoritmus, ktorý $O(\log rs)$-krát odsimuluje pohyb p pracovníkov (platí $p \leq rs$) pomocou BFS, jeho časová zložitosť je preto $O(r^2 s^2 \log rs)$.

Nižšie si môžete pozrieť implementáciu, ktorá simuluje všetkých pracovníkov naraz. Pre každú sekundu si udržiavame dvojrozmernú mriežku a pre každé políčko mriežky si pamätáme, či už bolo zaplavené, prípadne, ktorí pracovníci by sa naň už stihli dostať.
Zo stavu sveta v sekunde n vieme spočítať, ako bude vyzerať svet v sekunde $n + 1$ tak, že prehľadávame iba z políčok, ktoré boli prvýkrát navštívené v poslednej sekunde (rovnako ako BFS).

Listing programu (Python)

```
r, c = [int(x) for x in input().split()]
T = [input().strip() for x in range(r)]
di, dj = [0, 0, 1, -1], [1, -1, 0, 0]

def workers_can_be_saved(release_time=1):
    if release_time == -1: return True # bin-search sentinel
    visited = [[set() for _ in t] for t in T] # set of workers (& acid as 0) that can be on this square
    next = []

    def try_spread_acid(i, j):
        if 0 not in visited[i][j]:
            visited[i][j].add(0)
            next.append((i, j, 0))
    def try_spread_worker(i, j, w):
        if 0 not in visited[i][j] and w not in visited[i][j]:
            visited[i][j].add(w)
            next.append((i, j, w))

    # simulate each second
    for t in range(r * c):
        if t == 0: # release acid
            for i in range(r):
                for j in range(c):
                    if T[i][j] == 'K':
                        try_spread_acid(i,j)
        if t == release_time: # release workers
            pid = 0
            for i in range(r):
                for j in range(c):
                    if T[i][j] == 'P':
                        pid += 1
                        try_spread_worker(i,j,pid)

        current, next = next, []
        if not current and t > release_time:
            break
        for i,j,k in current:
            for d in range(4):
                ii, jj = i + di[d], j + dj[d]
```

```
                if 0 <= ii < r and 0 <= jj < c and T[ii][jj] != '#':
                    if k == 0: try_spread_acid(ii,jj)
                    else:      try_spread_worker(ii,jj,k)

    escaped = set() # which workers managed to escape
    for i in range(r):
        for j in range(c):
            if T[i][j] == 'U':
                escaped |= visited[i][j]
    return len(escaped - set([0])) == pid


b, e = -1, r * c
while e - b > 1:
    m = (b + e) // 2
    if workers_can_be_saved(m): b = m
    else: e = m
print(b)
```

Vzorové riešenie

Najprv si ukážeme, ako sa zbaviť binárneho vyhľadávania, a potom ako efektívne vyriešiť p pracovníkov iba s jedným spustením BFS.

Všimnime si, že ak sa pracovník vie dostať na cieľové políčko `U` skôr, ako ho zaleje kyselina, tak musí existovať cesta z jeho počiatočného políčka do cieľového políčka taká, že na každé políčko na ceste sa dostane pracovník skôr ako kyselina.
Tvrdenie dokážeme sporom. Predpokladajme, že žiadna taká cesta neexistuje, čiže všetky cesty pracovníka, ktoré skončia na nezaplavenom `U`, majú nejaké políčko, na ktoré sa kyselina dostane skôr ako pracovník. Potom sa ale kyselina môže z tohto políčka rozlievať tým istým spôsobom, akým plánuje ísť pracovník, a teda na políčko `U` sa dostane skôr ako on. To je spor.
Ak sa teda pracovník nevie dostať na `U` skôr ako kyselina, tak nevie uniknúť.

(Mimochodom, pri hodnotení úlohy sme strhávali body riešeniam, ktoré toto tvrdenie využívali, ale nemali ho dostatočne zdôvodnené. Nestačí totiž povedať, že to zjavne platí. Pri malej zmene úlohy to totiž platiť prestane, viď sekciu Bonus nižšie.)

Nech k je najkratšia vzdialenosť spomedzi všetkých políčok `K` k políčku `U`. Políčko `U` bude teda zaplavené v čase k.
Ak teda chceme overiť, či vie jeden konkrétny pracovník uniknúť pokiaľ vyštartuje v čase t, stačí overiť, či sa dostane na políčko `U` skôr ako v čase k, teda či je jeho vzdialenosť od `U` menšia ako $k - t$.
Ak uvažujeme viacerých pracovníkov, nech p je najväčšia zo vzdialeností pracovníkov k políčku `U`. Musí platiť, že $p < k - t$, resp. $t \leq k - p - 1$.
Tým pádom sme si ale ukázali, že najneskorší možný čas, v ktorom musíme spustiť poplach je $t = k - p - 1$.

Pre políčka `K` hľadáme *najkratšiu* cestu k políčku `U`. Využiť preto môžeme ten istý princíp ako v domácom kole, keď sme spustili jedno BFS naraz zo všetkých políčok `K`.
Pre pracovníkov `P` však hľadáme *najdlhšiu* zo všetkých najkrajtších ciest do `U`, čo tento spôsob nevyrieši správne. Ako sa teda vieme vyhnúť opakovanému púšťaniu BFS?
Uvedomme si, že vzdialenosť z a do b je v našej mriežke rovnaká ako vzdialenosť z b do a. Namiesto toho aby sme pre každé `P` hľadali najkratšiu cestu do `U` môžeme pustiť opačné BFS, ktoré pustíme z políčka `U`, a ktoré nám nájde najkratšiu cestu do všetkých zvyšných políčok, vrátane všetkých `P`.

Pre zhrnutie, úlohu vyriešime nasledovne: Pomocou jedného prehľadávania do šírky so začiatkom v `U` zistíme vzdialenosti všetkých políčok na mape od políčka `U`. Ak sa na nejaké políčko nedá dostať, jeho vzdialenosť bude nekonečno.
Spomedzi všetkých vzdialeností `K` vyberieme **najmenšiu** a označíme ju k (v zadaní máme zaručené, že k je konečne veľké).
Spomedzi všetkých vzdialeností `P` vyberieme **najväčšiu** a označíme ju p (môže byť aj nekonečno).
Ak je $k - 1 < p$, vypíšeme -1, inak je odpoveď $k - p - 1$.

Časová zložitosť riešenia je $O(r \times s)$, lebo pustíme len jedno prehľadávanie do šírky na mriežke veľkosti $r \times s$ a zo získaných hodnôt už potom ľahko spočítame odpoveď.

Listing programu (Python)

```
from collections import deque

r, c = [int(x) for x in input().split()]
T = [input().strip() for x in range(r)]
di, dj = [0, 0, 1, -1], [1, -1, 0, 0]
INF = 1023456789

# BFS
distance = [[INF for _ in t] for t in T]
q = deque()
min_acid_dist = INF
max_worker_dist = -1

for i in range(r):
    for j in range(c):
        if T[i][j] == 'U':
            q.append((i, j))
            distance[i][j] = 0

while q:
    i, j = q.popleft()
    if T[i][j] == 'K':
        min_acid_dist = min(min_acid_dist, distance[i][j])
    if T[i][j] == 'P':
        max_worker_dist = max(max_worker_dist, distance[i][j])
    for d in range(4):
        ii, jj = i + di[d], j + dj[d]
        if 0 <= ii < r and 0 <= jj < c and T[ii][jj] != '#' and distance[ii][jj] > distance[i][j] + 1:
            distance[ii][jj] = distance[i][j] + 1
            q.append((ii, jj))

if min_acid_dist < max_worker_dist + 1:
    print(-1)
else:
    print(min_acid_dist - max_worker_dist -1)
```

Bonus

Na záver sa zamyslite, ako by sa úloha riešila, ak by sa pracovníci mohli hýbať dvakrát rýchlejšie než kyselina (krok každej pol sekundy).
Už nestačí odmerať vzdialenosti, pretože pozorovanie, ktoré sme spravili na začiatku sekcie Vzorové riešenie už nebude platit.

B-II-3 Zber jahôd

Podúloha a)

Pri hľadaní *jednej* najlepšej cesty cez mriežku môžeme využiť riešenie podobné tomu z domáceho kola. Namiesto toho, aby sme počítali len jednu výslednú hodnotu, budeme postupne počítať najvýnosnejšiu cestu s koncom na každom políčku. Presnejšie, pre každé políčko (x, y) chceme vypočítať koľko najviac jahôd vie Hanka vyzbierať na ceste z políčka $(0, 0)$ na políčko (x, y). Túto hodnotu označíme $cesta(x, y)$.
Môžeme si uvedomiť, že na políčko (x, y) sa vieme dostať iba dvoma spôsobmi – buď naň prídeme zhora, z políčka $(x - 1, y)$ alebo zľava z políčka $(x, y - 1)$. No a kým sa dostaneme na tieto políčka, chceme vyzbierať čo najviac jahôd, z ktorých si potom vyberieme tú lepšiu možnosť. Ak $jahody(x, y)$ označuje počet jahôd na políčku (x, y) tak nás to vedie k jednoduchému vzorcu na výpočet:

$$cesta(x, y) = jahody(x, y) + \max(cesta(x - 1, y), cesta(x, y - 1))$$

Tieto hodnoty následne počítame po riadkoch zľava doprava, pričom špeciálne pre začiatočné políčko platí $cesta(0, 0) = jahody(0, 0)$ a ak je x alebo y záporné, tak $cesta(x, y) = -\infty$. To je totiž políčko, z ktorého nevieme prísť, keďže je mimo mriežky.
Týmto spôsobom vypočítame rs hodnôt, každé v konštantnom čase, výsledná časová zložitosť takéhoto riešenia je teda $O(r, s)$.

Podúloha b)

Pri hľadaní najlepších dvoch ciest nás to môže lákať využiť riešenie z predchádzajúcej úlohy. Však čo by sa mohlo pokaziť, ak do riešenia zahrnieme najlepšiu možnú celkovú cestu, že? A hoci je to úvaha, nad ktorou sa oplatí zamyslieť, treba si to skúsiť aj odôvodniť, prípadne nájsť nejaký protipríklad.
Protiargument k Hankinmu navrhovanému riešeniu by mohol znieť tak, že ich cieľom je vyzbierať čo najviac políčok s veľa jahodami. Ak však Hanka pažravo vyzbiera tú najlepšiu cestu, zvyšné plodné políčka nemusia byť všetky navštíviteľné Jurom. Po troche kreslenia vieme následne prísť napríklad s takýmto protipríkladom:

0	**100**	**99**
0	**0**	**0**
99	**100**	**0**

Vidíme, že najlepšia cesta by zobrala obe políčka obsahujúce 100. Dve zvyšné políčka s 99 jahodami však Juro nevie vyzbierať obe naraz. Lepšie riešenie preto je, aby si tieto políčka rozdelili a každý vyzbieral jedno so 100 a jedno s 99 jahodami.

Podúloha c)

Ak teda zavrhneme „pažravé“ riešenia, musíme sa pokúsiť vymyslieť spôsob, akým efektívne vyskúšať všetky možnosti. Uvedomme si, že presne toto robilo riešenie v podúlohe a). Postupne sme sa pozerali na všetky možné cesty mriežkou. Tento postup sme však zoptimalizovali tak, že ak na jedno políčko existovalo viacero ciest (zhora aj zľava), zapamätali sme si len to lepšie z nich. Vedeli sme totiž, že do budúcna nás to menej dobré riešenie nemusí zaujímať, lebo môže byť nahradené tým lepším.
V riešení podúlohy a) sme postupovali tak, že pre každé miesto, kde sa nachádzala Hanka sme vyskúšali všetky možnosti, kde sa mohla nachádzať o krok dozadu. Rovnako však môžeme postupovať aj v tomto prípade. Akurát namiesto jedného človeka budeme sledovať pozície dvoch ľudí. Nech teda $cesta(x_h, y_h, x_j, y_j)$ označuje najväčší počet jahôd, ktoré sme mohli vyzbierať ak Hanka prišla na políčko (x_h, y_h) a Juro na políčok (x_j, y_j). Vieme, že hľadaným výsledkom je hodnota $cesta(r, s, r, s)$ a na začiatku platí $cesta(0, 0, 0, 0) = jahody(0, 0)$.
Ostáva zistiť, ako tieto hodnoty počítať. Ako to teda vyzeralo krok dozadu? Každý z nich má dve možnosti, z ktorého políčka prišiel na svoje aktuálne. Dokopy sú teda 4 možnosti toho, kde sa mohli obaja nachádzať krok dozadu a my si vyberieme najlepšiu z nich.
Pri počítaní si ešte musíme dávať pozor, aby sme niektoré políčko nezapočítali dvakrát. Preto chceme pre hodnotu $cesta(x_h, y_h, x_j, y_j)$ pridať jahody z oboch políčok $jahody(x_h, y_h)$ aj $jahody(x_j, y_j)$ iba ak $(x_h, y_h) \neq (x_j, y_j)$. Stanovme si teda pomocnú hodnotu $pocet(x_h, y_h, x_j, y_j)$, pre ktorú platí:

$$pocet(x_h, y_h, x_j, y_j) = \begin{cases} jahody(x_h, y_h) & \text{ak platí } (x_h, y_h) = (x_j, y_j) \\ jahody(x_h, y_h) + jahody(x_j, y_j) & \text{inak} \end{cases}$$

Pomocou tejto hodnoty potom môžeme hodnoty $cesta()$ počítať nasledovne:

$$\begin{aligned} \text{cesta}(x_h, y_h, x_j, y_j) = \max\Big\{&\text{cesta}(x_h - 1, y_h, x_j - 1, y_j), \\ &\text{cesta}(x_h, y_h - 1, x_j - 1, y_j), \\ &\text{cesta}(x_h - 1, y_h, x_j, y_j - 1), \\ &\text{cesta}(x_h, y_h - 1, x_j, y_j - 1)\Big\} \\ &+ \text{pocet}(x_h, y_h, x_j, y_j). \end{aligned}$$

Takéto riešenie má časovú zložitosť $O(r^2 s^2)$, toľko rôznych hodnôt totiž musíme spočítať.

Efektívnejšie riešenie

V predchádzajúcom riešení sme sa snažili dávať pozor na to, aby sme žiadne políčko nezapočítali dvakrát. Spravili sme to však naozaj správne? Pozerali sme sa totiž iba na prípad, keď sa obaja postavili na jedno políčko naraz. Nemohlo sa však stať, že niektoré políčko Hanka navštívila ako prvá a až neskôr prišiel na toto políčko Juro?

V našom riešení simulujeme prechod Hanky a Jura cez mriežku naraz. Vždy keď spraví jeden krok prvý z nich, spravý jeden krok aj druhý. To ale znamená, že obaja sú od začiatku $(0,0)$ vždy vzdialení rovnako veľa. Ak teda obaja navštívia niektoré políčko (x,y), musia tak spraviť obaja naraz v tom istom kroku, preto naše predchádzajúce riešenie funguje.

Toto pozorovanie nám však zároveň dáva návod na efektívnešie riešenie. Ak totiž počítame hodnotu $cesta(x_h, y_h, x_j, y_j)$ pre všetky možné štvorice súradníc, zbytočne počítame aj so situáciami, ktoré nikdy nemôžu nastať. Napríklad situácia $cesta(0,0,r,s)$ alebo $cesta(1,4,2,2)$ je zo začiatku nedosiahnuteľná. Ak sa teda zbavíme týchto zbytočných výpočtov, náš algoritmus zefektívnime.

Vzdialenosť políčka (x,y) od začiatku $(0,0)$ je $x+y$: počet krokov ktoré sme išli doprava plus počet krokov, ktoré sme išli doľava. Vieme, že naša simulácia sa správa tak, že Hanka aj Juro sú od začiatku vzdialení vždy rovnako veľa krokov. Každú možnú pozíciu Hanky a Jura si preto vieme reprezentovať iba pomocou troch hodnôt x_h, x_j a k, kde x_h, x_j sú ich x-ové súradnice a k počet krokov, čo zatiaľ spravili. Platí totiž, že $y_h = k - x_h$ a $y_j = k - y_h$.

Hodnoty $pocet()$ aj $cesta()$ preto vieme počítať aj nasledovne:

$$pocet(x_h, x_j, k) = \begin{cases} jahody(x_h, k-x_h) & \text{ak platí } x_h = x_j \\ jahody(x_h, k-x_h) + jahody(x_j, k-x_j) & \text{inak} \end{cases}$$

$$\begin{aligned} \text{cesta}(x_h, x_j, k) = \max\Big\{ & \text{cesta}(x_h-1, x_j-1, k-1), \\ & \text{cesta}(x_h, x_j-1, k-1), \\ & \text{cesta}(x_h-1, x_j, k-1), \\ & \text{cesta}(x_h, x_j, k-1) \Big\} \\ & + \text{pocet}(x_h, x_j, k). \end{aligned}$$

Aká je časová zložitosť tohto riešenia? Hodnota $k \leq r+s-2$, máme teda $r \cdot s \cdot (r+s-2)$ rôznych hodnôt $cesta()$, ktoré potrebujeme spočítať. Každú z nich vieme spočítať v konštatnom čase, keďže je to iba maximum 4 hodnôt. Časová zložitosť tohto riešenia je preto $O(rs(r+s))$.

Listing programu (Python)

```
r, s = map(int, input().split())
grid = [list(map(int, input().split())) for _ in range(r)]

NEG = -10**18
dp = [[[NEG]*(r+s+1) for _ in range(r)] for _ in range(r)]

# začiatok
dp[0][0][0] = grid[0][0]

for k in range(1, r+s-1):
    for x_h in range(r):
        y_h = k - x_h
        if not (0 <= y_h < s):
            continue

        for x_j in range(r):
            y_j = k - x_j
            if not (0 <= y_j < s):
                continue

            best = NEG
            for dx_h in (0, 1):
                for dx_j in (0, 1):
```

```
                    px_h = x_h - dx_h
                    px_j = x_j - dx_j
                    if px_h >= 0 and px_j >= 0:
                        best = max(best, dp[px_h][px_j][k-1])

            if best < 0:
                continue

            if x_h == x_j and y_h == y_j:
                pocet = grid[x_h][y_h]
            else:
                pocet = grid[x_h][y_h] + grid[x_j][y_j]

            dp[x_h][x_j][k] = best + pocet

print(dp[r-1][r-1][r+s-2])
```

B-II-4 Pomalý krtko

Podúloha a)

Čísla deliteľné tromi

Najprv ukážeme riešenie pre deliteľnosť 3. Spomeňme si na kritérium deliteľnosti trojkou: číslo je deliteľné tromi práve vtedy, keď je deliteľný tromi jeho ciferný súčet. Ako teda budeme budeme generovať číslo zľava doprava, v type nedokončenej miestnosti by sme si mohli „pamätať“ ciferný súčet dosiaľ vygenerovaných cifier. Možných súčtov je však viac ako našich 26 typov miestností, namiesto toho si teda budeme pamätať iba zvyšok tohto súčtu po delení tromi. Tým, že si pamätáme len zvyšok po delení, tak máme zaručené, že nám stačia na ukladanie tejto informácie len 3 typy nedokončených miestností (konkrétne, jeden pre každý zvyšok po delení).

Majme teda tri typy nedokončených miestností: `A` pre zvyšok 0, `B` pre zvyšok 1 a `C` pre zvyšok 2. Pravidlá pre tieto miestnosti sú také, že z každej môžeme vygenerovať ľubovoľnú cifru a zakončiť to nedokončenou miestnosťou zodpovedajúcou upravenému cifernému súčtu.

$$\texttt{A} \to \texttt{0A} \mid \texttt{1B} \mid \texttt{2C} \mid \texttt{3A} \mid \texttt{4B} \mid \texttt{5C} \mid \texttt{6A} \mid \texttt{7B} \mid \texttt{8C} \mid \texttt{9A}$$
$$\texttt{B} \to \texttt{0B} \mid \texttt{1C} \mid \texttt{2A} \mid \texttt{3B} \mid \texttt{4C} \mid \texttt{5A} \mid \texttt{6B} \mid \texttt{7C} \mid \texttt{8A} \mid \texttt{9B}$$
$$\texttt{C} \to \texttt{0C} \mid \texttt{1A} \mid \texttt{2B} \mid \texttt{3C} \mid \texttt{4A} \mid \texttt{5B} \mid \texttt{6C} \mid \texttt{7A} \mid \texttt{8B} \mid \texttt{9C}$$

Konštrukciu smieme zakončiť jedine vtedy, keď dosiaľ vygenerované cifry majú ciferný súčet deliteľný 3, t.j. zvyšok 0. To zabezpečíme tým, že budeme mať pravidlo $\texttt{A} \to \varepsilon$ (smieme ukončiť konštrukciu, ak je ciferný súčet deliteľný 3) a každé iné pravidlo bude mať na pravej strane nejakú nedokončenú miestnosť (žiadne iné pravidlo nevie ukončiť konštrukciu).

Ešte potrebujeme vyriešiť začiatok generovania. Začíname v Z a ciferný súčet je 0, t.j. chceme, aby Z bolo ako A. Avšak na rozdiel od A jednak nesmieme ukončiť konštrukciu (to by sme mali číslo bez cifier), a tiež nesmieme začať cifrou 0 okrem prípadu, keď je to jediná cifra čísla. Budeme mať teda pravidlá

$$\texttt{Z} \to \texttt{0} \mid \texttt{1B} \mid \texttt{2C} \mid \texttt{3A} \mid \texttt{4B} \mid \texttt{5C} \mid \texttt{6A} \mid \texttt{7B} \mid \texttt{8C} \mid \texttt{9A}.$$

Čísla deliteľné 35

Vyššie uvedený postup vieme zovšeobecniť pre deliteľnosť 35: mali by sme 35 typov nedokončených miestností, v ktorých si pamätáme zvyšok po delení 35, a pravidlá na prechody medzi nimi. Pri deliteľnosti 3 sme si podľa pravidla počítali zvyšok po delení *súčtu*, pre 35 však takéto pravidlo platiť nemusí. To ale nevadí, budeme si totiž pamätať rovno zvyšok po delení zatiaľ vytvoreného čísla. Totiž ak dosiaľ vygenerované číslo má zvyšok x, tak po pridaní cifry c bude mať zvyšok $(10 \cdot x + c) \bmod 35$.

Formálne: označme si tie typy nedokončených miestností postupne $\texttt{A}_0, \texttt{A}_1, \ldots, \texttt{A}_{34}$. Potom pravidlá na prechody medzi nimi budú

$$\texttt{A}_i \to \texttt{0A}_{(10\cdot i) \bmod 35} \mid \texttt{1A}_{(10\cdot i+1) \bmod 35} \mid \ldots \mid \texttt{9A}_{(10\cdot i+9) \bmod 35}$$

Problém ale je, že my máme k dispozícii iba 26 typov nedokončených miestností.

Uvedomme si, že číslo je deliteľné 35 práve vtedy, keď je deliteľné 7 a 5 zároveň (kedže $35 = 5 \cdot 7$, a 5 a 7 sú nesúdeliteľné – sú to rôzne prvočísla). No a všimnime si, že deliteľnosť 5 je veľmi jednoduchá – číslo je deliteľné 5 práve vtedy, keď je jeho posledná cifra buď 0 alebo 5. V priebehu generovania nám teda stačí pamätať si zvyšok po delení 7 (na čo potrebujeme iba 7 typov nedokončených miestností), a pridať vhodné pravidlá na dokončenie čísla.

Majme teda typy nedokončených miestností $\mathtt{A}_0, \mathtt{A}_1, \ldots, \mathtt{A}_6$ a pravidlá

$$\mathtt{A}_i \to \mathtt{0A}_{(10 \cdot i) \bmod 7} \mid \mathtt{1A}_{(10 \cdot i+1) \bmod 7} \mid \ldots \mid \mathtt{9A}_{(10 \cdot i+9) \bmod 7}.$$

Ďalej pravidlá pre počiatočný typ nedokončenej nory:

$$\mathtt{Z} \to \mathtt{0} \mid \mathtt{1A}_1 \mid \mathtt{2A}_2 \mid \mathtt{3A}_3 \mid \mathtt{4A}_4 \mid \mathtt{5A}_5 \mid \mathtt{6A}_6 \mid \mathtt{7A}_0 \mid \mathtt{8A}_1 \mid \ldots \mid \mathtt{9A}_2.$$

Nakoniec sa zamyslime nad tým, ako presne vieme číslo zakončiť. To sa dá dvomi spôsobmi: buď cifrou 0, alebo cifrou 5. Po pridaní poslednej cifry má byť číslo deliteľné siedmimi. Ak sme pridali 0, tak nám to deliteľnosť neovplyvní. Kedže 10 a 7 sú nesúdeliteľné, tak 7 delí $10x$ práve vtedy, keď 7 delí x. To zodpovedá pravidlu

$$\mathtt{A}_0 \to \mathtt{0}.$$

Ak zakončujeme cifrou 5, tak chceme, aby 7 delilo $10x + 5$. Vyskúšame ako x všetky možné zvyšky po delení 7, a dostaneme, že jediný vyhovujúci zvyšok je $x \equiv 3 \pmod 7$. To zodpovedá pravidlu

$$\mathtt{A}_3 \to \mathtt{5}.$$

Táto konštrukcia je správna, lebo chceme vygenerovať práve také reťazce, ktoré naraz spĺňajú:

1. nezačínajú 0 (zabezpečené pravidlami zo $\mathtt{Z}$)
2. zodpovedajú číslam deliteľným 7 (zabezpečené $\mathtt{A}_0, \ldots, \mathtt{A}_6$ a pravidlami z/do nich)
3. zodpovedajú číslam deliteľným 5 (zabezpečené pravidlami $\mathtt{A}_0 \to \mathtt{0}$ a $\mathtt{A}_3 \to \mathtt{5}$)

Poznámka autorov: Dá sa na to pozerať aj tak, že ak $\mathcal{A}_5$ je portfólio čísel deliteľných 5 a $\mathcal{A}_7$ je portfólio čísel deliteľných 7, tak vlastne hľadáme pravidlá pre portfólio $\mathcal{A}_5 \cap \mathcal{A}_7$. Vo vzorovom riešení domáceho kola sme si už ukazovali, ako sa taká sada pravidiel dá vo všeobecnosti zostrojiť. Táto konštrukcia sa dá zovšeobecniť aj pre pomalého krtka (uvažovali sme tam normálny tvar pravidiel, kde na pravej strane je *práve jedna* dokončená miestnosť alebo ε – čo je *skoro* to isté, ako pomalý krtko). Otázka znie, ako veľmi sa nám nafúkne počet typov nedokončených miestností. Dá sa rozmyslieť si, že ak a je počet typov miestností potrebný pre $\mathcal{A}_5$ a b je počet typov miestností potrebný pre $\mathcal{A}_7$, potom pre $\mathcal{A}_5 \cap \mathcal{A}_7$ použijeme $a \cdot b$ typov miestností. Ak teda optimalizujeme pravidlá a počet typov nedokončených miestností pre niektoré z $\mathcal{A}_5$ alebo $\mathcal{A}_7$, tak sa to pretaví aj do lepšej (čítaj: potrebujúcej menej typov nedokončených miestností) sady pravidiel pre $\mathcal{A}_5 \cap \mathcal{A}_7$.

Podúloha b)

Zjednodušená verzia: sedem a-čok za sebou

Zamyslime sa najprv nad jednoduchšou verziou úlohy, kde sa chceme vyhnúť podúseku $\mathtt{a}^7$. T.j. nesmieme nikdy vygenerovať sedem a-čok za sebou. V type nedokončenej miestnosti si môžeme pamätať, koľko a-čok za sebou sme už vygenerovali; ak ich je už 6, tak nedovolíme postavenie ďalšieho a-čka. Budeme mať teda 7 typov nedokončených miestností $\mathtt{A}_0, \mathtt{A}_1, \ldots, \mathtt{A}_6$. Pre prvých 6 dovolíme postavenie a-čka; tým sa počet a-čok za sebou zvýši o jeden. To zodpovedá pravidlám

$$\begin{aligned}
\mathtt{A}_0 &\to \mathtt{a}\mathtt{A}_1,\\
\mathtt{A}_1 &\to \mathtt{a}\mathtt{A}_2,\\
&\dots\\
\mathtt{A}_4 &\to \mathtt{a}\mathtt{A}_5,\\
\mathtt{A}_5 &\to \mathtt{a}\mathtt{A}_6.
\end{aligned}$$

Pre $\mathtt{A}_6$ nebudeme mať žiadne pravidlo, ktorý by postavilo $\mathtt{a}$ – chceme, aby vygenerovaná časť nory nikdy neobsahovala $\mathtt{a}^7$. Otázka znie, čo keď chceme postaviť iné dokončené miestností? Počet a-čok za sebou sa zmení na nula. Budeme mať teda pravidlá

$$\begin{aligned}
\mathtt{A}_0 &\to [\mathtt{b}-\mathtt{z}]\mathtt{A}_0,\\
\mathtt{A}_1 &\to [\mathtt{b}-\mathtt{z}]\mathtt{A}_0,\\
&\dots\\
\mathtt{A}_6 &\to [\mathtt{b}-\mathtt{z}]\mathtt{A}_0.
\end{aligned}$$

A nakoniec pravidlá pre začiatok a koniec nory. Na začiatku nemáme žiadne za sebou idúce a-čka, takže budeme mať jediné pravidlo $\mathtt{Z} \to \mathtt{A}_0$. Čo sa konca týka, počas konštrukcie nikdy nemáme sedem a-čok za sebou, môžme teda konštrukciu kedykoľvek ukončiť. To zodpovedá pravidlám $\mathtt{A}_i \to \varepsilon$ pre každé $i = 0, 1, \dots, 6$.

Všeobecné riešenie

Vyššie uvedený postup vieme zovšeobecniť. Noru chceme zostrojovať miestnosť po miestnosti tak, aby sme si vždy boli istí, že slovo $w = w_1 w_2 \dots w_n$, ktorému sa chceme vyhnúť, v dokončenej časti nory nie je. Zároveň si chceme popri konštrukcii pamätať informáciu o posledných miestnostiach, aby sme vedeli povedať, aké ďalšie dokončené miestnosti môžeme postaviť. (Napríklad ak sa chceme vyhnúť $\mathtt{a}^7$ a posledných šesť miestností sú samé $\mathtt{a}$, tak vieme, že nesmieme postaviť $\mathtt{a}$.)

Určite by stačilo, ak by sme si pamätali posledných $n-1$ dokončených miestností, vďaka tomu budeme vždy vedieť aké písmeno vieme doplniť aby sme nevytvorili zakázaný reťazec w. Na to by sme však potrebovali 26^{n-1} typov nedokončených miestností, čo je priveľa.
Ako sme však videli v jednoduchšej verzii, v skutočnosti nám stačí si pamätať akú dlhú časť zakázaného slova w máme aktuálne postavenú na konci nory. Presnejšie, budeme mať jeden typ nedokončenej miestnosti pre každé k od 0 po $n-1$, pričom k označuje *koľko najviac* z posledných dokončených miestností je zhodných s prvými k znakmi slova w (toto označíme prefix slova w).
Vďaka tomu budeme vedieť, že ak je $k = n-1$, tak nemôžeme postaviť miestnosť typu w_n, inak môžeme postaviť ľubovoľnú z miestností.

Otázkou ostáva, či si vieme túto informáciu korektne udržiavať vždy po dostavení ďalšej miestnosti x.
Nech sme vedeli, že posledných k miestností sa zhoduje s prefixom w. Za týmito miestnosťami sa pridala miestnosť x. Určite sa teda vieme pozrieť na týchto $k+1$ posledných znakov a vypočítať si, koľko najviac z nich sa rovná prefixu w a to použiť ako novú hodnotu k. Takto však vypočítame nanajvýš hodnotu $k+1$ – buď sa písmeno x rovná w_{k+1} a len si ten počet zväčšíme, alebo nejakú informáciu zahodíme, lebo najdlhšia zhoda bude kratšia.
Nemôže sa však stať, že pridaním písmena x v skutočnosti dĺžka zhodujúceho úseku stúpne o viac ako 1? Napríklad na $k+2$?
To by ale znamenalo:

- Pred pridaním x sme mali na konci nory $w_1 w_2 \dots w_k$.
- Po pridaní x máme na konci nory $w_1 w_2 \dots w_{k+1} w_{k+2}$.

Ak označíme miestnosť pred w_1 ako q, tak po pridaní x zároveň platí:

- Posledných $k+2$ hotových miestností je $q w_1 w_2 \dots w_k x$.

- Posledných $k+2$ hotových miestností je $w_1 w_2 \dots w_{k+1} w_{k+2}$.

Potom ale $q w_1 w_2 \dots w_k = w_1 w_2 \dots w_{k+1}$, čo je v spore s tým, že pred pridaním x bol najdlhší zhodný prefix dlhý k. Takže dĺžka najdlhšieho úseku nevie skočiť o $+2$ alebo viacej. Ak teda poznáme najdlhši prefix pred pridaním x, tak máme dosť informácie na výpočet najdlhšieho prefixu aj po pridaní x. V podstate stačí iba vyskúšať všetky možné prefixy do dĺžky $k+1$ a zistiť, či pasujú. Ukážeme si to na príklade, ktorý je predmetom podúlohy, a to $w = \texttt{ananas}$:

- Vždy, keď pridáme miestnosť inú ako $\texttt{a}, \texttt{n}, \texttt{s}$, tak nový najdlhší prefix bude ε. (Žiaden prefix `ananas` nebude pasovať, lebo `ananas` takúto cudziu miestnosť neobsahuje.)
- Ak najdlhší prefix je ε, po pridaní miestnosti sú iba dve možnosti na najdlhší prefix: $\texttt{a}$ a ε. Ak pridáme $\texttt{a}$, tak to bude to prvé, inak to bude to druhé.
- Ak najdlhší prefix je $\texttt{a}$, po pridaní miestnosti môže byť najdlhší prefix $\texttt{an}, \texttt{a}$ alebo ε. Prvé dostaneme pridaním $\texttt{n}$, druhé pridaním $\texttt{a}$ a tretie pre ľubovoľnú inú miestnosť.
- Ak najdlhší prefix je $\texttt{an}$, po pridaní miestnosti môže byť najdlhší prefix $\texttt{ana}, \texttt{an}, \texttt{a}, \varepsilon$. Prvé dostaneme pridaním $\texttt{a}$. Druhé dostať nevieme, lebo to by znamenalo $\texttt{an}x = q\texttt{an}$ pre nejaké x, q, ale druhá miestnosť nesedí. Tretie dostať nevieme, lebo by sme museli pridať $\texttt{a}$ – ale vtedy dostaneme už dlhší prefix `ana`. Prefix ε dostaneme pre všetky ostatné miestnosti.
- ...
- Ak najdlhší prefix je `anana`, po pridaní miestnosti môže byť $\texttt{ananas}, \texttt{anana}, \texttt{anan}, \texttt{ana}, \texttt{an}, \texttt{a}, \varepsilon$.
  - `ananas`: Dostaneme pridaním $\texttt{s}$. My ale chceme zakázať vygenerovanie podúseku `ananas`, tak v tomto prípade pre miestnosť $\texttt{s}$ pravidlo nespravíme (neumožníme ju postaviť ako ďalšiu).
  - `anana`: Nevieme dostať, lebo to by znamenalo $\texttt{anana}x = q\texttt{anana}$ – ale napr. druhá miestnosť nesedí.
  - `anan`: Dostaneme pridaním $\texttt{n}$.
  - `ana`: Nevieme dostať.
  - `an`: Dostali by sme pridaním $\texttt{n}$, ale vtedy dostaneme dlhší prefix `anan`.
  - $\texttt{a}$: Dostaneme pridaním $\texttt{a}$.
  - ε: Dostaneme pridaním ľubovoľnej inej miestnosti.

Označme si nedokončené miestnosti $\texttt{A}_\varepsilon, \texttt{A}_\texttt{a}, \texttt{A}_\texttt{an}, \dots, \texttt{A}_\texttt{anana}$. Vyššie uvedené úvahy zodpovedajú nasledovnej sade pravidiel:

$$
\begin{aligned}
\texttt{A}_\varepsilon &\to \texttt{a}\texttt{A}_\texttt{a} \mid [\texttt{b}-\texttt{z}]\texttt{A}_\varepsilon \\
\texttt{A}_\texttt{a} &\to \texttt{n}\texttt{A}_\texttt{an} \mid \texttt{a}\texttt{A}_\texttt{a} \mid [\texttt{b}-\texttt{m}]\texttt{A}_\varepsilon \mid [\texttt{o}-\texttt{z}]\texttt{A}_\varepsilon \\
\texttt{A}_\texttt{an} &\to \texttt{a}\texttt{A}_\texttt{ana} \mid [\texttt{b}-\texttt{z}]\texttt{A}_\varepsilon \\
\texttt{A}_\texttt{ana} &\to \texttt{n}\texttt{A}_\texttt{anan} \mid \texttt{a}\texttt{A}_\texttt{a} \mid [\texttt{b}-\texttt{m}]\texttt{A}_\varepsilon \mid [\texttt{o}-\texttt{z}]\texttt{A}_\varepsilon \\
\texttt{A}_\texttt{anan} &\to \texttt{a}\texttt{A}_\texttt{anana} \mid [\texttt{b}-\texttt{z}]\texttt{A}_\varepsilon \\
\texttt{A}_\texttt{anana} &\to \texttt{n}\texttt{A}_\texttt{anan} \mid \texttt{a}\texttt{A}_\texttt{a} \mid [\texttt{b}-\texttt{m}]\texttt{A}_\varepsilon \mid [\texttt{o}-\texttt{r}]\texttt{A}_\varepsilon \mid [\texttt{t}-\texttt{z}]\texttt{A}_\varepsilon
\end{aligned}
$$

Pre úplnosť potrebujeme ešte pravidlá pre začiatok a koniec. Na začiatku je najdlhší prefix ε, stačí teda pravidlo $\texttt{Z} \to \texttt{A}_\varepsilon$. Ukončiť konštrukciu smieme v ľubovoľnom momente, čo zodpovedá pravidlám $N \to \varepsilon$ pre každé $N = \texttt{A}_\varepsilon, \texttt{A}_\texttt{a}, \texttt{A}_\texttt{an}, \dots, \texttt{A}_\texttt{anana}$.

Podúloha c)

Najprv sa zamyslime, v čom by mohol byť problém. Ak by sme neboli obmedzení na pomalého krtka, tak zakončiť noru postupnosťou presne 47 a-čok nie je problém:

1. Buď je nora dlhá presne 47 (vtedy to je presne nora $\texttt{a}^{47}$),
2. alebo je nora dlhšia ako 47 miestností. Vtedy určite končí 47 a-čkami a 48-sma miestnosť od konca je rôzna od $\texttt{a}$ (t.j. $[\texttt{b}-\texttt{z}]$).

Prvý prípad zachytíme pravidlom $\mathtt{Z} \to \mathtt{a}^{47}$. Druhý prípad zachytíme pravidlami

$$\mathtt{Z} \to [\mathtt{a}-\mathtt{z}]\mathtt{X},$$
$$\mathtt{X} \to [\mathtt{a}-\mathtt{z}]\mathtt{X} \mid [\mathtt{b}-\mathtt{z}]\mathtt{a}^{47}.$$

Ak sme obmedzení na pomalého krtka, tak pravidlo $\mathtt{Z} \to \mathtt{a}^{47}$ (a ani $\mathtt{X} \to [\mathtt{b}-\mathtt{z}]\mathtt{a}^{47}$) nevieme použiť. Vo vzorovom riešení domáceho kola sme si ukazovali, že ak sme ochotní nafúknuť počet typov nedokončených miestností, tak sa takéto pravidlá dajú nahradiť postupnosťou viacerých:

$$\mathtt{Z} \to \mathtt{a}\mathtt{Z}_1, \quad \mathtt{Z}_1 \to \mathtt{a}\mathtt{Z}_2, \quad \ldots, \quad \mathtt{Z}_{45} \to \mathtt{a}\mathtt{Z}_{46}, \quad \mathtt{Z}_{46} \to \mathtt{a}.$$

Problém ale je, že my sme obdmedzení na 26 typov nedokončených miestností. Takže toto obmedzenie bude kľúčový dôvod, prečo sa to nedá.

Dokazujme sporom: predpokladajme, že máme sadu pravidiel, ktorej portfólio je práve $\mathcal{C}$. Ukážeme, že potom táto sada pravidiel vygeneruje aj nejakú noru $x \notin \mathcal{C}$ (čo bude hľadaný spor). Zrejme $a^{47} \in \mathcal{C}$. Túto noru teda vieme zostrojiť nejakou postupnosťou pravidiel, začínajúc s nedokončenou miestnosťou $\mathtt{Z}$. Táto postupnosť bude vyzerať takto:

$$\mathtt{Z} \quad \to^* \quad \mathtt{a}N_1 \quad \to^* \quad \mathtt{a}^2N_2 \quad \to^* \quad \mathtt{a}^3N_3 \quad \to^* \quad \ldots \quad \to^* \quad \mathtt{a}^{46}N_{46} \quad \to^* \quad \mathtt{a}^{47},$$

kde N_i sú (nejaké) nedokončené miestnosti a $\to^*$ znamená, že z ľavej strany dostaneme pravú stranu na niekoľko pravidiel (t.j. nie nutne presne jedno pravidlo). To platí, pretože síce môžeme v pravidle nedokončiť žiadnu miestnosť, nevieme ale dokončiť viacero miestností naraz. Teda v nore budú dokončené miestnosti $\mathtt{a}$ pribúdať jedna po druhej, nikdy nie viacero naraz.

Nedokončené miestnosti $N_1, \ldots, N_{46}$ nie sú známe, my ale vieme, že všetkých typov nedokončených miestností je len 26. To znamená (z Dirichletovho princípu), že niektorý typ sa v našom postupe objavuje aspoň dvakrát; označme ho D. Náš postup teda vyzerá aj takto:

$$\mathtt{Z} \quad \to^* \quad \mathtt{a}^iD \quad \to^* \quad \mathtt{a}^{i+j}D \quad \to^* \quad \mathtt{a}^{47},$$

pre nejaké $j > 0$. No ale potom vieme časti medzi týmito D-čkami vynechať, a stále dostaneme korektný postup zodpovedajúci nejakej postupnosti pravidiel:

$$\mathtt{Z} \quad \to^* \quad \mathtt{a}^iD \quad \to^* \quad \mathtt{a}^{47-j},$$

Presnejšie, od okamihu, kedy vyrobíme $\mathtt{a}^iD$, budeme v tomto druhom postupe používať tie pravidlá, ktoré sme v prvom postupe použili až od $a^{i+j}D$. Toto vieme spraviť, lebo začíname z tej istej nedokončenej nory D. To znamená, že do portfólia musí patriť aj nora $\mathtt{a}^{47-j}$, čo je hľadaný spor: $\mathtt{a}^{47-j}$ nekončí na 47 a-čok.

Poznámka autorov: Táto úvaha v trochu všeobecnejšej forme hovorí, že nezávisle od sady pravidiel, na dostatočne dlhých norách sa nám niektorý typ nedokončenej miestnosti D zopakuje. To dá veľa informácie o portfóliu: jednak môžeme úsek medzi D-čkami vypustiť (ako sme videli v podúlohe c)), alebo ho môžeme aj ľubovoľne veľakrát zopakovať. Dalo by sa teda povedať, že portfólio je vždy v nejakom zmysle „periodické“. Môžte si skúsiť týmto spôsobom dokázať, že neexistuje sada pravidiel (dokonca ani pre rýchleho krtka!), ktorej portfólio by bolo:

1. $\{\mathtt{a}^n\mathtt{b}^n \mid n \in \mathbb{N}\}$
2. $\{\mathtt{a}^{n^2} \mid n \in \mathbb{N}\}$
3. $\{\mathtt{a}^p \mid p \text{ je prvočíslo}\}$
4. $\{w \mid \text{slovo } w \text{ je palindróm, teda rovnaké odpredu a odzadu}\}$

Krtkovia, nory, konštrukčné pravidlá a portfólia sú dobre preštudovanou oblasťou informatiky; samozrejme v literatúre majú iné pomenovania. Oblasť informatiky, ktorá ich študuje, sa volá *formálne jazyky a automaty*; nory sú *slová*, portfóliá sú *jazyky*, krtkovia a ich konštrukčné pravidlá zodpovedajú *regulárnym gramatikám*.

Jazyky, pre ktoré existuje regulárna gramatika, sa volajú *regulárne jazyky* a sú to presne tie jazyky, ktoré vieme rozpoznávať *konečným automatom*.

ŠTYRIDSIATY PRVÝ ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Truc Lam Bui, Ján Hozza, Jakub Šimo
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: NIVAM – Národný inštitút vzdelávania a mládeže, Bratislava 2026