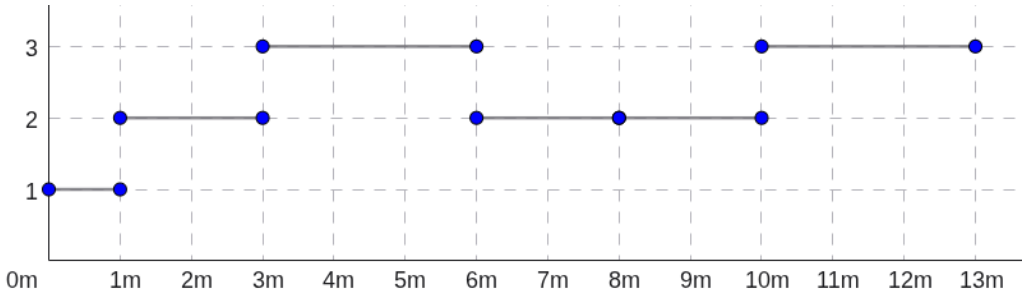
B-I-1 Preteky

Každú cestu pretekára si vieme predstaviť ako postupnosť t zvýšení ($\uparrow$), znížení ($\downarrow$) alebo nezmenení ($-$) rýchlosti, ktorou práve ide.
Vo vzorovom riešení túto postupnosť budeme nazývať rýchlostný *profil* pretekára. Každý rýchlostný profil si vieme zakresliť do grafu, kde na x-ovej osy bude prejdená vzdialenosť a na y-ovej rýchlosť, ktorou pretekár ide. Každý časový úsek v tomto grafe si reprezentujeme jednou úsečkou.



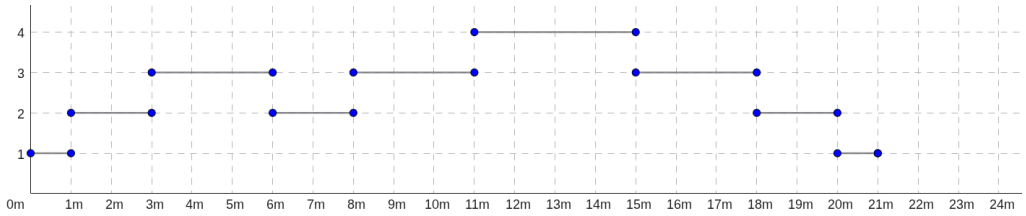
Príklad grafu pre profil $(\uparrow, \uparrow, \uparrow, \downarrow, -, \uparrow)$.

Ako bude vyzerať najlepšie riešenie?

Intuitívne by sa mohlo zdať, že najlepšie je nejaký čas zrýchľovať, potom zopár sekúnd zostať na rovnakej rýchlosti a nakoniec dobrzďovať do cieľa. Táto intuícia ale nie vždy platí. Napríklad pre dĺžku trate 11 a maximálnu cieľovú rýchlosť 1 neexistuje takýto rýchlostný profil, ktorý by preteky dokončil za 6 sekúnd. Existuje ale rýchlostný profil $(\uparrow, \uparrow, \uparrow, \downarrow, -, \downarrow)$, ktorý preteky dokončí za 6 sekúnd.

Naša intuícia sa však nemýlila úplne. Potrebujeme len dovoliť, aby sa rýchlosť mohla nezmeniť nanajvýš v jednej sekunde, keď traktor nejde maximálnou rýchlosťou.
Profil nazveme *kopcovitý*, ak v ňom pretekár najprv zrýchľuje na nejakú rýchlosť V_{max}, na nej nejaký čas zostane, a potom do cieľa len spomaľuje. Navyše v najviac jednej sekunde môže namiesto zrýchlenia alebo spomalenia nezmeniť svoju rýchlosť.

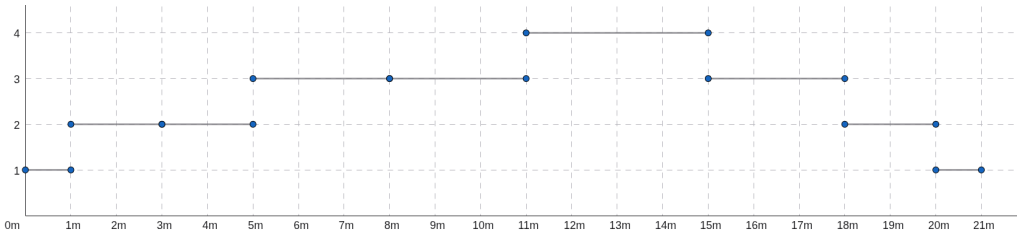
Pozrime sa na traktoristu Ferka, ktorý nemal kopcovitý rýchlostný profil. Postupne si ukážeme, ako ho zmeniť na kopcovitý, s ktorým dokončí preteky za rovnaký alebo lepší čas. Graf Ferkovho rýchlostného profilu pre preteky s dĺžkou 21 a maximálnou povolenou rýchlosťou 1 by mohol vyzerať napríklad takto:



Jeden z možných rýchlostných profilov traktoristu Ferka.

Ako prvé si môžeme všimnúť, že ak zmeníme poradie úsečiek v ľubovoľnom grafe rýchlostného profilu, tak sa celková prejdená vzdialenosť nezmení. Môžeme ich teda preusporiadať tak, aby Ferko najprv zrýchľoval alebo nemenil svoju rýchlosť, potom niekoľko sekúnd zostal na rýchlosti V_{max}, a následne len spomaľoval do cieľa[1]. Tento rýchlostný profil je skoro kopcovitý, len počas zrýchľovania na rýchlosť V_{max} mohol Ferko nezrýchliť vo viac ako jednej sekunde.

[1] Ak by náhodou bola rýchlosť V_{max} menšia ako X, tak tento úsek vynecháme



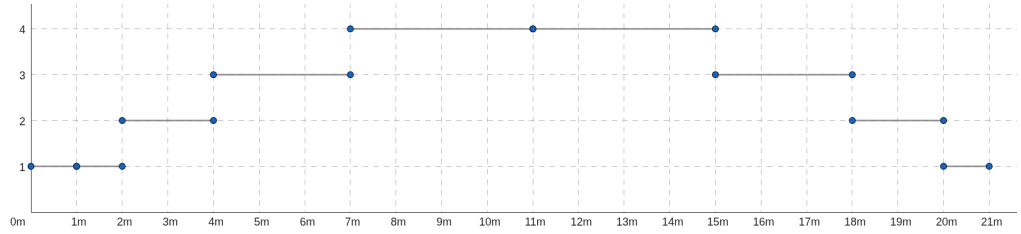
Preusporiadaný graf Ferkovho rýchlostného profilu.

Ak počas zrýchľovania Ferko nezmenil rýchlosť vo viac ako jednej sekunde, tak sme už našli kopcovitý profil. Inak v nejakých časoch t a s nezmenil svoju nemaximálnu rýchlosť. Označme si V_t a V_s rýchlosti, ktorými v týchto sekundách Ferko išiel.

Ak $V_t+V_s \leq V_{max}$, tak tieto dva úseky môžeme nahradiť jedným, v ktorom Ferko pôjde rýchlosťou V_t+V_s. Touto úpravou Ferko dokončí preteky o 1 sekundu skorej a jeho celková prejdená vzdialenosť sa nezmení. Rýchlosťou $V_t + V_s$ Ferko počas zrýchľovania niekedy určite išiel, lebo kým zrýchlil na rýchlosť V_{max}, tak išiel všetkými menšími rýchlosťami. Tento časový úsek teda máme kam do profilu pridať.

Ak $V_t + V_s > V_{max}$, tak tieto dva úseky môžeme nahradiť inými dvoma úsekmi. V jednom pôjde rýchlosťou V_{max} a v druhom rýchlosťou $V_t + V_s - V_{max}$. Rýchlosť $V_t + V_s - V_{max}$ je určite menšia ako V_{max}, lebo rýchosti V_t aj V_s sú menšie ako V_{max}. Týmto spôsobom sa Ferkova prejdená dráha určite nezmení a preteky dokončí za rovnaký čas ako pred zmenou.

V oboch prípadoch z Ferkovho rýchlostného profilu zmiznú 2 úseky, v ktorých nezrýchľoval, a pribudne najviac jeden úsek, počas ktorého nebude zrýchľovať (na nemaximálnej rýchlosti). Opakovaním tohto postupu nám budú postupne ubúdať úseky, v ktorých Ferko nebude zrýchľovať. Keď skončíme, tak nám zostane profil, ktorý bude kopcovitý a preteky dokončí za rovnaký alebo lepší čas ako pôvodne.



Graf Ferkovho rýchlostného profilu po jednom odstránení zlých časových úsekov.

Úseky v čase 3 a 5, počas ktorých Ferko dokopy prejde 5 metrov nahradíme jedným úsekom, v ktorom pôjde rýchlosťou 4 m/s a druhým, v ktorom pôjde rýchlosťou 1 m/s.

Ukázali sme si, že každý rýchlostný profil vieme prerobiť na kopcovitý, v ktorom dokončíme preteky za rovnaký alebo lepší čas ako pôvodne. To ale znamená, že existuje nejaký najrýchlejší rýchlostný profil, ktorý je kopcovitý. Pri hľadaní najmenšieho času, za ktorý sa preteky dajú dokončiť sa teda môžeme obmedziť len na kopcovité rýchlostné profily.

Matematická vsuvka

Neskôr v riešení sa nám bude hodiť vedieť vypočítať súčet postupností v tvare $a + (a + 1) + (a + 2) + \cdots + b$. Takýmto postupnostiam sa hovorí *aritmetické postupnosti*. Ak ste o nich už počuli, pokojne môžete túto sekciu preskočiť.

Súčet čísel si najprv prepíšeme tak, aby sme najprv sčítali prvé číslo s posledým, druhé s predposledným a tak ďalej. Ak sme mali v postupnosti párne veľa čísel, tak nám na konci žiadne nezostane. Ak nepárne veľa, tak nám na konci zostane prostredné číslo, presnejšie číslo $\frac{a+b}{2}$.

Môžeme si všimnúť, že každá z takto vytvorených dvojíc má rovnaký súčet $a + b$. Počet čísel v postupnosti je $b - a + 1$. Ak je počet čísel párny, tak súčet postupnosti bude $\frac{(a+b)(b-a+1)}{2}$, lebo máme $\frac{b-a+1}{2}$ dvojíc, kde každá má súčet $a + b$. Ak máme nepárny počet čísel, tak dokopy máme $\frac{b-a}{2}$ dvojíc so súčtom $a + b$ a jedno číslo $\frac{a+b}{2}$. Teda súčet aj v tomto prípade vyjde $\frac{(a+b)(b-a)}{2} + \frac{a+b}{2} = \frac{(a+b)(b-a+1)}{2}$.

Efektívne riešenie

V tomto riešení budeme skúšať, pre ktorú maximálnu rýchlosť V_{max} dokončíme preteky za najkratší čas. Ako sme si v skorších sekciách ukázali, stačí sa pozerať na kopcovité rýchlostné profily. V takýchto profiloch budeme určite najprv V_{max} sekúnd zrýchľovať na rýchlosť V_{max} m/s a potom ďalších $X - V_{max}$ sekúnd spomaľovať na rýchlosť X m/s. Pre tieto úseky si vieme pomocou vzorca na súčet aritmetickej postupnosti spočítať, koľko metrov počas nich prejdeme. Pre prejdenú vzdialenosť D musíme ešte prejsť aspoň $n - D$ metrov. Túto vzdialenosť prejdeme rýchlosťou V_{max}, až na jednu sekundu, v ktorej môžeme ísť aj pomalšie ako V_{max}. To presne odpovedá celočíselnému deleniu so zaokrúhľovaním nahor, teda zvyšok pretekov prejdeme za $\lceil \frac{n-D}{V_{max}} \rceil$ sekúnd.

Treba si už len dať pozor na zopár okrajových prípadov. Ak skúšame rýchlosť V_{max}, ktorá je menšia ako X, tak musíme vynechať úsek, v ktorom spomaľujeme. Ak nám vyjde prejdená vzdialenosť $D > n$, znamená to, že pri danej V_{max} nestíhame dobrzdiť do cieľa. Tým pádom nedobrzdíme ani pri väčších hodnotách V_{max} a nemusíme ich ani skúšať. Tento prípad nastane vtedy, keď je V_{max} zhruba $\sqrt{n}$, čo sa dá vidieť aj z vyššie spomenutého vzorca súčtu aritmetickej postupnosti. To znamená, že časová zložitosť tohto riešenia bude $\mathcal{O}(q\sqrt{n})$, lebo pre každú rýchlosť X nám stačí vyskúšať $\mathcal{O}(\sqrt{n})$ rýchlostných profilov.

Listing programu (Python)

```
def vyries(n, X):
    # Preteky sa urcite daju dokoncit za n sekund
    najlepsie_t = n

    # Prechadzame cez vsetky hodnoty V_max
    for V_max in range(1, n):
        # Pri zrychlovani dokopy prejdeme 1 + 2 + ... + V_max metrov
        D_zrychlovanie = (1 + V_max) * V_max // 2

        # Zrychlujeme V_max sekund
        t_zrychlovanie = V_max

        # Ak V_max > X, tak musime spomalovat
        if V_max > X:
            # Pocas spomalovania prejdeme (V_max - 1) + (V_max - 2) + ... + X metrov
            D_spomalovanie = (X + V_max - 1) * (V_max - X) // 2

            # Spomalovat budeme V_max - X sekund
            t_spomalovanie = V_max - X
        else:
            D_spomalovanie = 0
            t_spomalovanie = 0

        D_rovnake = n - D_spomalovanie - D_zrychlovanie

        # Zvysnu cast drahy prejdeme na rychlosti V_max. Pripocitanim V_max - 1
        # zarucime, ze pri deleni dostaneme hornu celu cast
        t_rovnake = (max(D_rovnake, 0) + V_max - 1) // V_max

        t = t_zrychlovanie + t_spomalovanie + t_rovnake

        najlepsie_t = min(t, najlepsie_t)

        # Ak sme uz nemuseli ziadnu cast cesty ist rovnakou rychlostou, tak
        # dalsie moznosti uz nema zmysel kontrolovat
        if D_rovnake <= 0:
            break

    print(najlepsie_t)


n, q = map(int, input().split())

for _ in range(q):
    X = int(input())
    vyries(n, X)
```

Rýchlejšie riešenie

Hoci predchádzajúce riešenie je dosť efektívne na to, aby vyriešilo zadanú úlohu v rozumnom čase, existuje ešte lepšie riešenie.
Pri tomto riešení je potrebné si všimnúť, že sa stačí pozerať len na najväčšiu vzdialenosť, ktorú dokáže pretekár prejsť za t sekúnd tak, aby skončil na rýchlosti najviac X. Ak takto prejde viac ako n metrov, tak sa dá dokázať, že dokáže preteky dokončiť aj normálnym spôsobom za t sekúnd.
Najväčšiu vzdialenosť za t sekúnd pretekár prejde, ak bude najprv zrýchľovať na nejakú rýchlosť V_{max}, potom na nej najviac jednu sekundu zostane a následne bude spomaľovať na rýchlosť X. Teraz si môžeme všimnúť, že ak má pretekár viac času, tak dokáže prejsť väčšiu vzdialenosť. Presnejšie to znamená, že ak vie pretekár dokončiť preteky za t sekúnd, tak ich vie dokončiť aj za $t + 1$ sekúnd.

S touto informáciou môžeme použiť techniku binárneho vyhľadávania. Pre každú rýchlosť X binárne vyhľadáme najmenší čas, za ktorý vieme prejsť viac ako n metrov. Výsledná časová zložitosť tohto riešenia bude $\mathcal{O}(q \log n)$.

B-I-2 Nedočkavosť

Pri praktických úlohách je dobré začať naprogramovaním ľubovoľného (aj pomalšieho) *správneho* riešenia. Keď totiž neskôr programujeme rýchlejšie riešenie, môžeme využiť pomalšie riešenie na kontrolu správnosti. A navyše, často získame cenný vhľad do úlohy.
Najjednoduchšie riešenie, ktoré sa nám ponúka, je priamo naprogramovať postup zo zadania.

Simulujeme zadanie

V zadaní sme mali 3 typy žiakov, `K` – rieši a kontroluje, `n` – rieši a nekontroluje a `#` – nerieši a nekontroluje. Pre našu simuláciu potrebujeme vedieť, ktorí žiaci sa už dozvedeli o zverejnení zadaní.

Môžeme si všimnúť, že aj žiaci typu `K` aj žiaci typu `n` sa po tom, čo sa o zverejnení zadaní dozvedia, správajú úplne rovnako. Pre oba tieto typy žiakov nám teda stačí pridať jeden nový typ žiaka, označme ho `D`. Tento typ bude znázorňovať, že daný žiak sa o zverejnení už dozvedel. Žiaci typu `D` budú preto každú minútu hovoriť o zverejnení úloh všetkým svojim susedom.
Ak má teda žiak typu `D` na začiatku niektorej minúty za suseda žiaka typu `K` alebo `n`, zmení sa tento sused na žiaka typu `D`. Už sa totiž dozvedel, že zadania boli zverejnené a chce o tom povedať ostatným.

Pre žiakov typu `#` nás tiež zaujíma, kedy sa o informácii dozvedeli. Preto si zavedieme ešte jeden typ žiaka, označme ho `Z`, ako zmenený. Takýto žiak o informácií už vie, ďalej ju ale nešíri.

Pozrime sa teraz na samotný postup simulácie. Šírenie informácie budeme simulovať minútu po minúte. Ak si pre každého žiaka správne pamätáme jeho typ, každú minútu prejdeme celú triedu a pre každého žiaka `D` rozšírime informáciu aj na jeho susedov. To znamená, že jeho susedov typu `K` a `n` zmeníme na typ `D` a susedov typu `#` zmeníme na typ `Z`. Keďže priamo simulujeme postup zo zadania, bude počet žiakov, ktorých sme v danej minúte zmenili z `K` alebo `n` na `D`, plus počet tých čo sme zmenili z `#` na `Z`, presne tá hodnota, čo nás zaujíma. Spočítame ju a číslo vypíšeme.

Ostáva doladiť pár detailov.

Dokedy máme postup opakovať?
Ak sa v niektorej minúte m stane, že sa žiaden nový žiak nedozvedel o zadaniach (nanastala zmena na `D` alebo `Z`), vieme, že žiadna zmena nenastane ani v ďalšej minúte.
V minúte $m + 1$ totiž túto informáciu opäť šíria tí istí žiaci typu `D`. A ak to nemali komu novému povedať v minúte m, rovnako to bude platiť aj v každej ďalšej minúte.
Naša simulácia teda môže zastať v okamihu, keď bude počet zmenených žiakov 0.
Čo spraviť v minúte 0?
Náš program rozširuje informáciu od žiakov typu `D`. V minúte 0 však žiadny takí žiaci v mriežke nie sú. Zo zadania však vieme, že v minúte 0 sa informáciu dozvedia práve žiaci typu `K`. Náš program teda zmení všetky `K` na `D` a vypíše ich počet.

Slovne by sme náš algoritmus popísali nasledovne:

1. V minúte 0: Spočítaj, koľko je v mriežke znakov `K`. Toto číslo vypíš. Následne zmeň všetky znaky `K` na `D`.
2. Postupne simuluj minúty od 1, až kým nezastavíš. Pre každú minútu nastav počet zmenených na 0 a prejdi každé políčko v mriežke. Ak je na políčku znak `D`. Pozri sa na každé susedné políčko. Ak je na ňom znak `n` (uvedomte si, že znak `K` tam nebude) zmeň ho na `D` a zvýš počet zmenených o 1. Ak je na ňom znak `#`, zmeň ho na `Z` a zvýš počet zmenených o 1. Ak je po prejdení všetkých políčok počet zmenených 0, ukonči algoritmus. Inak vypíš počet zmenených a opakuj tento krok.

Ak by sme takýto program naprogramovali a spustili na súťažných vstupoch, za pár sekúnd by prešiel na vstupoch 1, 2, 5 a 6. Pri zvyšných by sme však museli čakať dlhšie, niektorých výsledkov by sme sa asi ani nedočkali.

Náš program je teda príliš pomalý. Čo väčšinou znamená, že robíme niečo *zbytočne* alebo *duplicitne*. A ak sa týchto zbytočných výpočtov zbavíme, algoritmus zlepšíme. Pozrime sa teda, čo by sme robiť nemuseli.

V každej minúte rozširujeme informáciu z každého žiaka typu `D`, tak ako káže zadanie. Toto však vôbec nie je potrebné. Ak sme v nejakej minúte m rozšírili informáciu od žiaka `D`, v ďalších minútach od tohto žiaka informáciu rozširovať už nemusíme. Najneskôr v minúte m sa totiž o zadaniach dozvedeli všetci jeho susedia, čím sa zmenili na `D` alebo `Z`. V ďalších minútach by im teda len hovoril to, čo už vedia, čo je *duplicitné*. Vidíme teda, že z každého žiaka stačí informáciu šíriť najviac raz.

Navyše, na to, aby sme zistili, z ktorých žiakov máme informáciu rozširovať, **prechádzame v každej minúte celú mriežku**. Z predchádzajúceho bodu ale vieme, že nám stačí informáciu šíriť raz pre každého žiaka. Pri prechode sa teda pozeráme na *zbytočne* veľa políčok.
Vieme, že žiak informáciu o zadaniach šíri presne minútu po tom, ako sa o nej dozvedel. Ak sa totiž žiak dozvie o zadaniach v minúte m, v minúte $m+1$ o tom povie všetkým svojim susedom a v ďalších minútach to už opakovať nemusí. Žiaci, ktorí budú informáciu posúvať v čase $m+1$ sú teda práve tí, ktorí sa o nej dozvedeli v čase m.

V našom programe si teda vytvoríme dvojrozmerné pole `spracuj[]`. Na pozícii `spracuj[m]` bude zoznam všetkých žiakov, ktorí sa o zadaniach dozvedeli prvýkrát v minúte m, každý žiak bude zapamätaný ako dvojica $(riadok, stĺpec)$, v ktorom sa nachádza.
Vždy, keď náš algoritmus v minúte m zmení žiaka `K` alebo `n` na typ `D`, zaznačíme tohto žiaka do zoznamu `spracuj[m]`. A v minúte $m+1$ budeme informáciu šíriť iba zo žiakov poznačených v `spracuj[m]`.

Takýto vylepšený algoritmus by už mal zbehnúť za pár sekúnd na všetkých súťažných vstupoch. Jeho časová zložitosť je totiž $O(rs)$ a v skutočnosti je to veľmi dobrá implmentácia známeho algortimu *prehľadávania do šírky*.

Listing programu (Python)

```
riadky, stlpce = map(int, input().split())
mriezka = [list(input().strip()) for _ in range(riadky)]

# najskôr riešime minútu 0
vysledky = []
pocet = 0
spracuj = [[]]

for r in range(riadky):
    for s in range(stlpce):
        if mriezka[r][s] == 'K':
            pocet += 1
            mriezka[r][s] = 'D'
            spracuj[0].append((r, s))  # pridávame zmeny v minúte 0

vysledky.append(pocet)

m = 1  # ktorú minútu spracovávame

# susedia políčka
dx = [-1, 0, 1, 0]
dy = [0, -1, 0, 1]
```

```
while True:
    pocet = 0
    spracuj.append([])  # pridáme prázdny zoznam pre aktuálnu minútu

    for (r, s) in spracuj[m - 1]:  # pozeráme žiakov z predchádzajúcej minúty
        for i in range(4):
            # vypočítame pozíciu susedného políčka
            # prejdeme ich v poradí hore, vlavo, dole, vpravo
            x = r + dx[i]
            y = s + dy[i]

            # kontrola hraníc
            if 0 <= x < riadky and 0 <= y < stlpce:
                if mriezka[x][y] == 'n':
                    spracuj[m].append((x, y))
                    mriezka[x][y] = 'D'
                    pocet += 1
                elif mriezka[x][y] == '#':
                    mriezka[x][y] = 'X'
                    pocet += 1

    if pocet == 0:
        break

    vysledky.append(pocet)
    m += 1

print(len(vysledky) - 1)
for x in vysledky:
    print(x)
```

B-I-3 Cesty v mriežke

V tejto úlohe máme mriežku $r \times s$ a našou úlohou je zistiť, koľko rôznych ciest vedie z ľavého horného rohu do pravého dolného rohu, ak sa môžeme hýbať iba doprava alebo dodola.
V zadaní je dokonca popísaný šikovný spôsob ako túto hodnotu počítať: Do každého políčka mriežky si zapíšeme, koľkými spôsobmi sa vieme z ľavého rohu dostať na práve toto políčko. Pre políčko na pozícii x a y túto hodnotu označíme $cesty(x, y)$. Následne vieme, že ak sa chceme dostať na políčko (x, y), máme iba dve možnosti – buď naň prídeme zhora z políčka $(x - 1, y)$ alebo zľava z políčka $(x, y - 1)$. No a výsledný počet možností pre (x, y) je súčet možností pre obe tieto políčka. Platí teda, že $cesty(x, y) = cesty(x - 1, y) + cesty(x, y - 1)$.
Pomocou tohto vzorca vieme hodnoty v mriežke počítať postupne po riadkoch zhora nadol a zľava doprava, až kým nevypočítame hodnotu $cesty(r, s)$.
Súťažnou úlohou bolo tento postup upraviť tak, aby riešil aj iné variácie tejto úlohy.

Podúloha a) – neprechodné políčka

V mriežke je niekoľko neprechodných políčok, cez ktoré prejsť nevieme. Ako to ovplyvní naše riešenie?
Uvedomme si, že základná myšlienka riešenia ostáva nezmenená: Počet možností pre *prázdne políčko* bude rovný súčtu možností pre políčko nad ním a naľavo od neho. Jediné, čo sa zmení je, že počet možností, ako sa dostať na *neprechodné políčko* bude vždy 0.
Opäť teda budeme hodnoty počítať riadok po riadku, akurát na neprechodné políčko zapíšeme 0 a na prázdne políčko súčet dvoch susedných, rovnako ako predtým.
Náš vzorec by sme si teda vedeli zapísať ako:

$$cesty(x, y) = \begin{cases} 0 & \text{ak je políčko } (x, y) \text{ neprechodné,} \\ cesty(x - 1, y) + cesty(x, y - 1) & \text{ak je políčko } (x, y) \text{ prázdne.} \end{cases}$$

1	1	1	1	0	0
1	2	3	4	4	4
1	3	6	0	4	8
0	3	9	9	13	21

Listing programu (Python)

```
# vstupna mriezka . je prazdne policko N neprechodne
mriezka = [
    '....N.',
    '......',
    '...N..',
    'N.....',
]
r, s = len(mriezka), len(mriezka[0])
# prázdne pole s r riadkami a s stĺpcami
cesty = [[0 for _ in range(s)] for _ in range(r)]

cesty[0][0] = 1  # počiatočná hodnota

for x in range(r):
    for y in range(s):
        if mriezka[x][y] == 'N':
            cesty[x][y] = 0
        else:
            if x != 0:  # existuje políčko nadomnou
                cesty[x][y] += cesty[x - 1][y]
            if y != 0:  # existuje políčko nal'avo
                cesty[x][y] += cesty[x][y - 1]

print(cesty[r - 1][s - 1])
```

Podúloha b) – pohyb šikmo doprava-hore

V tejto podúlohe bol pridaný nový smer pohybu. Okrem doprava a dodola sa môžeme hýbať aj *šikmo doprava-hore*. V zadaní však bolo napísané ako postupovať pri pridaní nového smeru pohybu – do počtu možností započítame jednu hodnotu navyše.
V našom prípade sa na políčko (x, y) vieme dostať nielen z políčok $(x-1, y)$ a $(x, y-1)$, ale aj z políčka $(x+1, y-1)$. To je totiž políčko, z ktorého keď sa pohneme šikmo doprava-hore, ocitneme sa opäť na políčku (x, y).
Vzorec výpočtu teda vieme zapísať ako $cesty(x, y) = cesty(x-1, y) + cesty(x, y-1) + cesty(x+1, y-1)$.

Ak však tento postup skúsime aplikovať na ukážkovú tabuľku, rýchlo narazíme na problém. Hodnota na začiatočnom políčku $(1, 1)$ bude 1. Ďalšie políčko v poradí výpočtu (keďže ideme po riadkoch) je políčko $(1, 2)$. To znamená, že potrebujeme hodnoty z políčok $(1, 1)$ (pohyb doprava), $(0, 2)$ (pohyb dodola) a $(2, 1)$ (pohyb doprava-hore).
Hodnotu $cesty(1, 1)$ sme si pred chvíľou vypočítali. A políčko $(0, 2)$ je mimo mriežky, teda $cesty(0, 2) = 0$. Avšak políčko $(2, 1)$ sa v mriežke nachádza, jeho hodnotu sme však *ešte nepočítali*. Odkazujeme sa teda na hodnotu, ktorú vypočítanú nemáme a to je chyba.
Našťastie hodnotu $cesty(2, 1)$ vieme vypočítať, lebo tá sa neodkazuje na žiadne neznáme hodnoty, musíme ju teda spočítať predtým ako budeme počítať $cesty(1, 2)$.
K správnemu riešeniu teda už potrebujeme spraviť len to, aby sme hodnoty na políčkach počítali *v správnom poradí* tak, aby sme na výpočet hodnoty $cesty(x, y)$ mali už vypočítané všetky potrebné hodnoty. Zo vzorca vyššie vidíme, že pre políčko (x, y) potrebujeme mať vypočítanú jednu hodnotu priamo nad ním a dve hodnoty z vedľajšieho stĺpca. Najjednoduchšie teda bude **počítať mriežku po stĺpcoch** zľava doprava a zhora dole.

1	2	6	22	89
1	4	16	67	288
1	6	29	132	588
1	7	36	168	756

Listing programu (Python)

```
r, s = 4, 5
# prázdne pole s r riadkami a s stĺpcami
cesty = [[0 for _ in range(s)] for _ in range(r)]

cesty[0][0] = 1  # počiatočná hodnota

for y in range(s):
    for x in range(r):
            if x != 0:  # existuje políčko nadomnou
                cesty[x][y] += cesty[x - 1][y]
            if y != 0:  # existuje políčko nal'avo
                cesty[x][y] += cesty[x][y - 1]
            if y != 0 and x != r - 1:  # existuje policko sikmo
                cesty[x][y] += cesty[x + 1][y - 1]

print(cesty[r - 1][s - 1])
```

Podúloha c) – červené políčka

V mriežke sú niektoré políčka označené červenou farbou a nás zaujíma počet ciest z jedného rohu do druhého, ktoré prechádzajú *aspoň* jedným červeným políčkom. Označme si tieto cesty ako „červené“.
Definujme si, čo potrebujeme počítať. Pre každé políčko (x, y) chceme spočítať hodnotu $cervene(x, y)$ reprezentujúcu počet rôznych ciest z ľavého horného rohu na políčko (x, y), ktoré prechádzajú aspoň jedným červeným políčkom. Hodnota $cervene(r, s)$ je potom naša hľadaná odpoveď.
Túto hodnotu vieme pre (x, y) počítať podobne ako v predchádzajúcich úlohách. Ak totiž zoberieme všetky červené cesty končiace na políčku $(x - 1, y)$ z ktorých sa pohenem dodola a všetky červené cesty končiace na políčku $(x, y - 1)$, z ktorých sa poheneme doprava, dostaneme rôzne červené cesty vedúce na políčko (x, y), tieto možnosti teda musíme započítať.
Pre *prázdne* políčka je tento výpočet dokonca správny. Problém nastane pri červených políčkach. Pri nich totiž nastáva špecifická situáciu, s ktorou nepočítame. Cesty, ktoré vedú na červené políčko (x, y), predtým ale nenavštívili žiadne iné červené políčko sú totiž červené. V našom prípade však tieto cesty nepočítame, keďže cesty $cervene(x - 1, y)$ aj $cervene(x, y - 1)$ už červené políčko navštívili.
Potrebovali by sme teda poznať aj hodnoty $necervene(x - 1, y)$ a $necervene(x, y - 1)$, ktoré určujú, koľko ciest vedúcich na dané políčka **neprechádza** cez žiadne červené políčko. Uvedomme si však, že toto je ten istý problém ako v podúlohe a), kde sme nemohli prechádzať cez čierne neprechodné políčka.
Pre červené políčko (x, y) bude platiť, že musíme započítať jednak možnosti $cervene(x - 1, y)$ a $cervene(x, y - 1)$, ale aj tie, pri ktorých sme dovtedy červené políčko nestretli, čo budú hodnoty $necervene(x - 1, y)$ a $necervene(x, y - 1)$.

Trik výsledného riešenie teda spočíva v tom, že nebudeme počítať jednu tabuľku, ale **dve tabuľky** hodnôt. Prvú tabuľku hodnôt $necervene(x, y)$ spočítame rovnako ako v podúlohe a) pomocou vzorca:

$$\mathbf{necervene}(x, y) = \begin{cases} 0 & \text{ak je políčko } (x, y) \text{ červené,} \\ \mathbf{necervene}(x - 1, y) + \mathbf{necervene}(x, y - 1) & \text{ak je políčko } (x, y) \text{ prázdne.} \end{cases}$$

No a túto tabuľku potom využijeme pri výpočte hodnôt $cervene(x, y)$, pre ktoré platí vzorec:

$$cervene(x,y) = \begin{cases} cervene(x-1,y) + cervene(x,y-1) \\ + \textbf{necervene}(x-1,y) + \textbf{necervene}(x,y-1) & \text{ak je políčko } (x,y) \text{ červené,} \\ cervene(x-1,y) + cervene(x,y-1) & \text{ak je políčko } (x,y) \text{ prázdne.} \end{cases}$$

necervene

1	1	0	0	0	0
1	2	2	2	2	2
1	0	2	4	6	0
1	1	3	7	13	13

cervene

0	0	1	1	1	1
0	0	1	2	3	4
0	3	4	6	9	21
0	3	7	13	22	43

Listing programu (Python)

```
# vstupna mriezka . je prazdne policko C cervene
mriezka = [
    '..C...',
    '......',
    '.C...C',
    '......',
]
r, s = len(mriezka), len(mriezka[0])
# prázdne pole s r riadkami a s stĺpcami
necervene = [[0 for _ in range(s)] for _ in range(r)]

necervene[0][0] = 1  # počiatočná hodnota

for x in range(r):
    for y in range(s):
        if mriezka[x][y] == 'C':
            necervene[x][y] = 0
        else:
            if x != 0:  # existuje políčko nadomnou
                necervene[x][y] += necervene[x - 1][y]
            if y != 0:  # existuje políčko nal'avo
                necervene[x][y] += necervene[x][y - 1]

cervene = [[0 for _ in range(s)] for _ in range(r)]

cervene[0][0] = 1 if mriezka[0][0] == 'C' else 0  # počiatočná hodnota

for x in range(r):
    for y in range(s):
        if x != 0:  # existuje políčko nadomnou
            cervene[x][y] += cervene[x - 1][y]
            if mriezka[x][y] == 'C':
                cervene[x][y] += necervene[x - 1][y]
        if y != 0:  # existuje políčko nal'avo
            cervene[x][y] += cervene[x][y - 1]
            if mriezka[x][y] == 'C':
                cervene[x][y] += necervene[x][y - 1]

print(cervene[r - 1][s - 1])
```

Podúloha d) – najčervenejšie cesty

Hľadáme počet takých červených ciest, ktoré prechádzajú cez čo najväčší možný počet červených políčok. Vidíme, že riešenie predchádzajúcej úlohy nám stačiť nebude, v ňom si totiž nepamätáme nič o počte červených políčok, cez ktoré sme prešli. A vlastne ani nevieme, cez koľko najviac červených políčok vieme prechádzať. Pokúsme sa teda najprv vypočítať túto hodnotu.

Nech $pocet(x, y)$ určuje cez koľko najviac červených políčok viem prechádzať na ceste z ľavého horného rohu do políka (x, y). Čo platí pre túto hodnotu? Opäť bude musieť súvisieť so susednými políčkami. Akákoľvek cesta

do (x, y) prechádza buď cez $(x-1, y)$ alebo $(x, y-1)$.
Hodnota $pocet(x-1, y)$ určuje cez koľko najviac červených políčok vieme prejsť ak sa chceme dostať nad (x, y). A $pocet(x, y-1)$ cez koľko najviac čevených políčok vieme prejsť ak sa chceme dostať naľavo od (x, y). Je potom jasné, že z týchto dvoch možností si chceme vybrať tú **väčšiu**. Tá menšia nás už zaujímať predsa nemusí.
Bude teda platiť, že $pocet(x, y) = \max(pocet(x-1, y), pocet(x, y-1))$. Navyše, ak je samotné políčko (x, y) červené, túto hodnotu chceme zväčšiť o 1, keďže potrebujeme započítať aj prechod cez toto políčko.
Celkový vzorec teda vyzerá nasledovne:

$$\mathbf{pocet}(x, y) = \begin{cases} \max(\mathbf{pocet}(x-1, y), \mathbf{pocet}(x, y-1)) + 1 & \text{ak je políčko } (x, y) \text{ červené,} \\ \max(\mathbf{pocet}(x-1, y), \mathbf{pocet}(x, y-1)) & \text{ak je políčko } (x, y) \text{ prázdne.} \end{cases}$$

Vráťme sa teraz k pôvodnej úlohe, v ktorej chceme spočítať počet červených ciest prechádzajúcich cez najväčší počet červených políčok. Nech $najcervensie(x, y)$ označuje práve tento počet pre políčko (x, y). Je jasné, že táto hodnota by nejakým spôsobom mala súvisieť s hodnotami $najcervensie(x-1, y)$ a $najcervensie(x, y-1)$, problém ale je, že najčervenšie cesty na políčko $(x-1, y)$ môžu ísť cez iný počet červených políčok ako cesty na políčko $(x, y-1)$, nemôžeme ich teda iba tak spočítať dokopy.
Vďaka hodnotám $pocet(x, y)$ ale presne vieme, cez koľko červených políčok prechádzajú. Cesty $najcervensie(x-1, y)$ totiž musia prechádzať cez práve $pocet(x-1, y)$ červených políčok. Zrazu teda vieme tieto hodnoty rozlišovať. Porovnáme teda hodnoty $pocet(x-1, y)$ a $pocet(x, y-1)$. Ak je jedna z nich ostro väčšia ako druhá, hodnota $najcervensie(x, y)$ bude rovná hodnote $najcervensie()$ z daného dominantného smeru, keďže menej červené cesty započítať nechceme. A ak sa oba počty rovnajú, v takom prípade chceme zobrať súčet hodnôt $najcervensie()$ z oboch smerov.

$$najcervensie(x, y) = \begin{cases} najcervensie(x-1, y) & \text{ak } \mathbf{pocet}(x-1, y) > \mathbf{pocet}(x, y-1), \\ najcervensie(x, y-1) & \text{ak } \mathbf{pocet}(x-1, y) < \mathbf{pocet}(x, y-1), \\ najcervensie(x-1, y) + najcervensie(x, y-1) & \text{ak } \mathbf{pocet}(x-1, y) = \mathbf{pocet}(x, y-1). \end{cases}$$

pocet

0	0	1	1	1	1	1	2
0	0	1	1	2	2	2	3
0	1	1	1	2	3	3	3
1	1	1	1	2	3	3	3
1	1	2	2	3	3	3	3

najcervensie

1	1	1	1	1	1	1	1
1	2	1	2	3	3	3	4
1	3	4	6	3	6	6	10
1	4	8	14	3	6	12	22
1	5	13	13	16	22	34	56

Listing programu (Python)

```
# vstupna mriezka . je prazdne policko C cervene
mriezka = [
    '..C....C',
    '....C..C',
    '.C...C..',
    'C.......',
    '..C.C...',
]
r, s = len(mriezka), len(mriezka[0])
# prázdne pole s r riadkami a s stĺpcami
pocet = [[0 for _ in range(s)] for _ in range(r)]

pocet[0][0] = 1 if mriezka[0][0] == 'C' else 0  # počiatočná hodnota
```

```
for x in range(r):
    for y in range(s):
        if x != 0:  # existuje políčko nadomnou
            pocet[x][y] = max(pocet[x][y], pocet[x - 1][y])
        if y != 0:  # existuje políčko nal'avo
            pocet[x][y] = max(pocet[x][y], pocet[x][y - 1])
        if mriezka[x][y] == 'C':
            pocet[x][y] += 1

najcervensie = [[0 for _ in range(s)] for _ in range(r)]

for x in range(r):
    for y in range(s):
        if x == 0 and y == 0:  # počiatočná hodnota
            najcervensie[0][0] = 1
            continue
        if x == 0:
            najcervensie[x][y] = najcervensie[x][y - 1]
        elif y == 0:
            najcervensie[x][y] = najcervensie[x - 1][y]
        elif pocet[x - 1][y] < pocet[x][y - 1]:
            najcervensie[x][y] = najcervensie[x][y - 1]
        elif pocet[x - 1][y] > pocet[x][y - 1]:
            najcervensie[x][y] = najcervensie[x - 1][y]
        elif pocet[x - 1][y] == pocet[x][y - 1]:
            najcervensie[x][y] = najcervensie[x - 1][y] + najcervensie[x][y - 1]

print(najcervensie[r - 1][s - 1])
print(pocet)
print(najcervensie)
```

B-I-4 Krtko staviteľ

Podúloha a)

Myšlienka je nasledovná: Najprv vytvoríme miestnosti `hlinahlinahlina`, čím zabezpečíme že budeme mať reťazec `hlina` aspoň 3-krát. Potom budeme mať jedno pravidlo, ktoré nám dovolí pridať miestnosti `hlina`, a druhé, ktoré nás zbaví nedokončenej miestnosti, čím dostavia noru. Výsledné pravidlá teda môžu vyzerať nasledovne:

$$\texttt{Z} \to \texttt{hlinahlinahlinaA}$$

$$\texttt{A} \to \texttt{hlinaA} \mid \epsilon$$

Na začiatku Krtko nemá na výber, teda po prvom kroku dostane $(\texttt{hlina})^3\texttt{A}$. Potom Krtko ešte niekoľkokrát použije prvé pravidlo pre `A`, a nakoniec musí použiť druhé pravidlo, lebo to ako jediné vytvorí dokončenú noru. Pred použitím pravidla $\texttt{A} \to \epsilon$ mala nora tvar $(\texttt{hlina})^\texttt{n}\texttt{A}$ pre nejaké $\texttt{n} \geq 3$. Každá nora z portfólia bude teda mať požadovaný tvar.
Ostáva ukázať, že každú požadovanú noru vie Krtko vytvoriť. To je ale jednoduché. Ak chce vytvoriť noru $(\texttt{hlina})^\texttt{k}$, stačí keď najprv použije pravidlo $\texttt{Z} \to \texttt{hlinahlinahlinaA}$, potom $\texttt{k} - 3$ krát použije pravidlo $\texttt{A} \to \texttt{hlinaA}$ a nakoniec použije pravidlo $\texttt{A} \to \epsilon$.

Podúloha b)

Najprv chceme budovať miestnosti `a`, potom raz miesnosť `b` a ďalej miestnosti `a`. Aby sme vedeli rozlíšiť, či sme už postavili miesnosť `b`, budeme používať rôznu nedokončenú miesnosť – miestnosť `Z` ak sme `b` ešte nepostavili a miestnosť `A` ak sme už miestnosť `b` postavili.
Naše pravidlá teda budú vyzerať:

$$\texttt{Z} \to \texttt{aZ} \mid \texttt{bA}$$

$$\texttt{A} \to \texttt{aA} \mid \epsilon$$

Všimnite si, že ak máme na konci miestnosť `Z`, môžeme buď pridať miestnosť `a` a nechať nedokončenú miestnosť `Z`, alebo zapísať to jediné `b`, ale tým zmeniť nedokončenú miestnosť na `A`.

Je zjavné, že každá dostavaná nora bude pozostávať iba z miestností `a` a `b`. Akonáhle Krtko niekedy postaví nedokončenú miesnosť `A`, už nikdy nepostaví miestnosť `b`. Jediný spôsob ako vytvoriť miestnosť `b` je pravidlo $\texttt{Z} \to \texttt{bA}$, ktoré vytvorí nedokončenú miesnosť `A`. Žiadna dostavaná nora preto nebude mať viac ako jednu miesnosť `b`. Bez použitia pravidla $\texttt{Z} \to \texttt{bA}$ môžeme využívať iba pravidlo $\texttt{Z} \to \texttt{aZ}$. Nikdy teda noru nedostaviame, ak nezapíšeme aspoň jedno `b`.
Každá nora z portfólia $\mathcal{B}$ má tvar $\texttt{a}^k\texttt{ba}^l$. Vieme ju vyrobiť tak, že najprv použijeme k-krát pravidlo $\texttt{Z} \to \texttt{aZ}$, potom raz pravidlo $\texttt{Z} \to \texttt{bA}$, l-krát pravidlo $\texttt{A} \to \texttt{aA}$ a nakoniec pravidlo $\texttt{A} \to \epsilon$. Všimnite si, že naše navrhnuté pravidlá dobre fungujú aj v prípade ak sa k alebo l rovná 0.

Podúloha c)

Ak nechceme dve miestnosti `b` za sebou, miestnosť `b` musí byť poslednou miestnosťou, alebo po nej musí nasledovať miestnosť `a`. Sada pravidiel teda bude

$$\texttt{Z} \to \texttt{aZ} \mid \texttt{baZ} \mid \texttt{b} \mid \epsilon$$

Určite nepostavíme nevhodnú noru, keďže vždy keď staviame miestnosť `b` buď hneď skončíme, alebo rovno za ňou postavíme miestnosť `a`.
Teraz ukážeme, že dokážeme vytvoriť každú noru. Pozrime sa na vhodné nory odzadu. Ak končí miestnosťou `b`, tak ako posledné použijeme pravidlo $\texttt{Z} \to \texttt{b}$. Teraz musíme vytvoriť nejakú časť nory, ktorá končí miestnosťou `a`. Podľa toho, či posledné dve miestnosti sú `ba` alebo `aa` použijeme pravidlá $\texttt{Z} \to \texttt{baZ}$ alebo $\texttt{Z} \to \texttt{aZ}$. Teraz opäť máme časť nory končiacu na `a`, ale kratšiu. Postupným opakovaním teda nájdeme postup (odzadu) ako postaviť každú noru.

Podúloha d)

Ak by sme chceli iba nory, ktoré začínajú aj končia na `a`, mohli by sme mať nasledovnú sadu pravidiel:

$$\texttt{Z} \to \texttt{aA} \mid \texttt{a}$$
$$\texttt{A} \to \texttt{aA} \mid \texttt{bA} \mid \texttt{cA} \mid \texttt{a}$$

Všimnite si, že pravidlo $\texttt{Z} \to \texttt{a}$ potrebuje kvôli tomu, že aj nora `a` patrí do nášeho portfólia.
Táto sada pravidiel určite vytvorí noru začínajúcu miestnosťou `a`, a jediné pravidlo, ktoré nevytvorí nedokončenú miesnosť dá ako poslednú miesnosť `a`.
Okrem prvého kroku navyše vždy môžeme vytvoriť ľubovoľnú miesnosť, takže vieme vyrobiť všetky vhodné nory.
Aby sme našli pravidlá pre $\mathcal{D}$, využijeme spôsob zo študijného textu pre zjednotenie. Dostaneme teda nasledovné pravidlá:

$$\texttt{Z} \to \texttt{aA} \mid \texttt{a} \mid \texttt{bB} \mid \texttt{b} \mid \texttt{cC} \mid \texttt{c}$$
$$\texttt{A} \to \texttt{aA} \mid \texttt{bA} \mid \texttt{cA} \mid \texttt{a}$$
$$\texttt{B} \to \texttt{aB} \mid \texttt{bB} \mid \texttt{cB} \mid \texttt{b}$$
$$\texttt{C} \to \texttt{aC} \mid \texttt{bC} \mid \texttt{cC} \mid \texttt{c}$$

Podúloha e)

Našou úlohou bolo nájsť sadu pravidiel, ktorej portfólio obsahuje tie nory, ktoré patria aj do $\mathcal{C}$ aj do $\mathcal{D}$. Pomerne ľahko vieme určiť, že to musia byť teda tie nory, ktoré sa skladajú z miestností `a` a `b` (miestnosti `c` sa nevyskytujú v $\mathcal{C}$), žiadne dve miestnosti typu `b` nie sú pri sebe a navyše tieto nory začínajú a končia rovnakou miestnosťou. Na vytvorenie týchto pravidiel stačí jemne upraviť naše riešenia z častí c) a d). Konkrétne:

$$\texttt{Z} \to \texttt{aA} \mid \texttt{baB} \mid \texttt{b}$$
$$\texttt{A} \to \texttt{aA} \mid \texttt{baA} \mid \epsilon$$
$$\texttt{B} \to \texttt{aB} \mid \texttt{baB} \mid \texttt{b}$$

Všimnite si, že kým začiatok s `a` je trochu jednoduchší, pri `b` musíme rozobrať viacero možností, napríklad aj tú, že výsledná nora obsahuje iba jedno písmeno `b`.

Všeobecný prienik

Úloha e) je však v skutočnosti oveľa zaujímavejšia ako sa na prvý pohľad zdá. Dá sa totiž ukázať, že existuje pomerne algoritmický postup ako skonštruovať prienik ľubovoľných dvoch portfólií. Ak teda máme portfóliá $\mathcal{X}$ a $\mathcal{Y}$, pre ktoré existujú nejaké sady krtkových pravidiel, vieme nájsť aj pravidlá pre $\mathcal{X} \cap \mathcal{Y}$.
Pre jednoduchosť dôkazu zavedieme *normálny tvar pravidiel*. Povieme, že budeme akceptovať iba také pravidlá, ktoré vytvoria vždy *práve jednu dokončenú* miestnosť alebo ϵ. To znamená, že pravidlo $\texttt{Z} \to \texttt{aA}$ je dovolené, ale pravidlo $\texttt{Z} \to \texttt{abB}$ alebo $\texttt{Z} \to \texttt{a}$ už nie.
Na prvý pohľad sa môže zdať, že sme naše pravidlá oslabili, lebo zrazu nevieme zapísať to čo predtým. Nie je to však tak. Vieme totiž zapísať aj komplikovanejšie pravidlá, akurát využitím viacerých medzipravidiel.
Predstavme si, že máme pravidlo $\texttt{X} \to \texttt{hlinaY}$. Toto pravidlo by sme chceli prepísať do normálneho tvaru. To ale predsa vieme spraviť nasledovnou sadou pravidiel:

$$\texttt{X} \to \texttt{hA}$$

$$\texttt{A} \to \texttt{lB}$$

$$\texttt{B} \to \texttt{iC}$$

$$\texttt{C} \to \texttt{nD}$$

$$\texttt{D} \to \texttt{aY}$$

Je zrejmé, že výsledkom bude to isté, akurát teraz sme potrebovali viacero nedokončených miestností. Kým tých však máme dostatok (napr. môžeme vytvárať aj číslované nedokončené miesnosti $A_1, A_2 \dots$), vieme takto „rozbiť“ ľubovoľne dlhé pravidlo.
Pravidlá typu $\texttt{Z} \to \texttt{a}$ vieme nahradiť dvojicou pravidiel $\texttt{Z} \to \texttt{aA}$ a $\texttt{a} \to \epsilon$. A namiesto $\texttt{Z} \to \texttt{A}$ pridáme miestnosti `Z` rovnaké pravidlá ako má `A`.
Takýto normálny tvar je síce nešikovný na samotné písanie pravidiel, uľahčí nám však dokazovanie, keďže stačí rozmýšľať nad oveľa menšou sadou možností.
Majme portfóliá $\mathcal{X}$ a $\mathcal{Y}$ a k nim zodpovedajúce sady pravidiel. Chceme ukázať, že v takom prípade existuje sada pravidiel aj pre $\mathcal{X} \cap \mathcal{Y}$. Upravme si sady pravidiel do normálneho tvaru, pričom sada pravidiel pre $\mathcal{X}$ bude okrem nedokončenej miestnosti `Z` používať už len miestnosti označené $\texttt{X}_\texttt{i}$ a sada pravidiel pre $\mathcal{Y}$ len miestnosti označené $\texttt{Y}_\texttt{i}$.
Zoberme si teraz ľubovoľnú noru w, ktorá patrí do $\mathcal{X} \cap \mathcal{Y}$. Dĺžku tejto nory si označíme n. Keďže nora w leží v $\mathcal{X}$, existuje postupnosť pravidiel z našej sady, ktorej aplikovaním vytvoríme túto noru. A keďže pravidlá normálneho tvaru vytvárajú vždy jednu nedokončenú miestnosť a na záver ϵ, vieme, že táto postupnosť bude mať dĺžku $n+1$ (n za každé písmeno a 1 za ϵ). Z tejto postupnosti aplikovania pravidiel si vieme zapísať, aká nedokončená miestnosť bola na konci nory počas jej výstavby, čím dostaneme postupnosť: $Z, X_{p_1}, X_{p_2} \dots X_{p_n}$.
Keďže nora w patrí aj do $\mathcal{Y}$, vieme zopakovať rovnakú úvahu a dostať postupnosť nedokončených miestností $Z, Y_{p_1}, Y_{p_2} \dots Y_{p_n}$.
Z tohto vieme usúdiť, že ak prvé písmeno nory w bolo `a`, tak v prvej sade existuje pravidlo $\texttt{Z} \to \texttt{a}\texttt{X}_{p_1}$ a v druhej $\texttt{Z} \to \texttt{a}\texttt{Y}_{p_1}$. A dokonca aj všeobecnejšie, ak bolo i-te písmeno nory w práve `a` (ale rovnako to platí aj pre zvyšné písmená), tak existujú pravidlá $\texttt{X}_{p_{i-1}} \to \texttt{a}\texttt{X}_{p_i}$ a $\texttt{Y}_{p_{i-1}} \to \texttt{a}\texttt{Y}_{p_i}$.
Našou úlohou je však vytvoriť sadu pravidiel pre $\mathcal{X} \cap \mathcal{Y}$. Chceli by sme preto nejakým spôsobom tieto rovnaké pravidlá prepojiť, aby pri stavaní nôr touto sadou musel krtko akoby naraz stavať aj podľa sady pre $\mathcal{X}$ aj pre $\mathcal{Y}$.
To sa dá spraviť jedným pomerne šikovným trikom. Nedokončené miestnosti pre pravidlá prieniku si označme $Z_{i,j}$, ktoré budú predstavovať situáciu, že na konci nory mám nedokončenú miestnosť $\texttt{X}_i$ podľa pravidiel pre $\mathcal{X}$ a nedokončenú miestnosť $\texttt{Y}_j$ podľa pravidiel pre $\mathcal{Y}$.
Následne sa pozrieme na všetky pravidlá pre $\texttt{X}_i$ a $\texttt{Y}_j$ a ak majú obe pravidlá, ktoré vytvárajú tú istú dokončenú miestnosť, teda $\texttt{X}_i \to \texttt{a}\texttt{X}_k$ a $\texttt{Y}_i \to \texttt{a}\texttt{Y}_l$, tak do našej prienikovej sady pravidiel pridáme $\texttt{Z}_{i,j} \to \texttt{a}\texttt{Z}_{k,l}$. A toto spravíme pre všetky dvojice nedokončených miestností a možných písmen.

Uvedomte si, že týmto trikom donútime našu novú sadu pravidiel, aby v podstate naraz simulovala aj aplikáciu pravidiel pre $\mathcal{X}$ aj $\mathcal{Y}$.
Skúste si to rozmyslieť a prípadne vyskúšať na príklade z úlohy e). Nie je to najjednoduchší dôkaz, ale keď mu porozumiete, zistíte, že je veľmi elegantný a pekný.

ŠTYRIDSIATY PRVÝ ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Michal Farnbauer, Martin Belluš, Ján Gottweis
Recenzia: Michal Anderle
Slovenská komisia Olympiády v informatike
Vydal: NIVAM – Národný inštitút vzdelávania a mládeže, Bratislava 2025