PRASK
VI. ročník, 2019/20
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

Vzorové riešenia 2. kola zimnej časti

1. Práca v továrni dláždení

vzorák napísal(a) Žaba
(max. 15 b za riešenie)

Podúloha a)

Prvou úlohou bolo zistiť počet dláždení plochy 4×4. Nie je ťažké si všetky takéto možnosti nakresliť. Ak si však chceme byť istý, že sme žiadnu možnosť nevynechali, je dobré postupovať systematicky. Jednou z možností je napríklad určiť si, akú najväčšiu dlaždičku v riešení použijeme.

Ak si povieme, že chceme použiť dlaždičku 4×4 máme práve jednu možnosť, pretože táto dlaždička pokryje celú požadovanú plochu. Nasledujú možnosti, ktoré využívajú dlaždičku 3×3. Do plochy 4×4 sa zmestí najviac jedna takáto dlaždička, môžeme ju však položiť na štyri rôzne miesta. Bez ohľadu na to, kam ju uložíme, budeme musieť zvyšok plochy vydláždiť pomocou 1×1 dlaždičiek, na výber teda nemáme, dokopy sú to ďalšie 4 možnosti.

Ak chceme použiť dlaždičku 2×2 musíme si tieto možnosti ešte viac rozdeliť. Takýchto dlaždičiek sa totiž do plochy 4×4 zmestí viacero. Môžeme si teda určiť aj to, koľko takýchto dlaždičiek použijeme. Výhodou je, že keď tento zvolený počet dlaždičiek umiestnime, zvyšok môžeme pokryť už len pomocou dlaždičiek 1×1, nebudeme teda musieť rozlišovať ďalšie možnosti, záleží len na tom, koľkými spôsobmi uložíme požadovaný počet dlaždíc 2.

Ak chceme položiť jednu dlaždicu 2×2 máme 9 možností – ľavý horný roh tejto dlaždice nemôže ísť do posledného riadku a stĺpcu. Pre dve dlaždice 2×2 je týchto možností 16, tu neostáva nič iné ako si ich postupne všetky nakresliť. Pre tri dlaždice je možností 8, okrem tých kde vynecháme jeden z rohov ešte treba nezabudnúť aj na možnosti kde je jedna dlaždička v strede hrany. No a pri použití štyroch dlaždíc máme 1 možnosť.

Rovnako jednu možnosť máme aj vtedy, ak si povieme, že chceme využiť iba dlaždice veľkosti 1×1. Dokopy sme teda napočítali $1 + 4 + 9 + 16 + 8 + 1 + 1 = 40$ možností.

Podúloha b)

Našou úlohou je zistiť počet dláždení plochy $4 \times n$ takých, že v poslednom stĺpci sa nachádzajú iba dlaždice 1. Navyše môžeme využiť trik, v ktorom sa spýtame podobnú otázku našich spolupracovníkov v továrni. Jediné čo je potrebné je, aby sme sa spýtali na menšiu plochu.

Uvedomme si, že ak je povedané, že v poslednom stĺpci sú iba dlaždice 1×1, je tento stĺpec jednoznačne určený. Navyše vôbec neovplyvňuje vyplnenie zvyšku plochy. Ostala nám teda plocha $4 \times n - 1$, ktorú môžeme vyplniť úplne ľubovoľne. Ku každému z týchto vyplnení potom už len stačí pridať posledný stĺpec plný dlaždíc 1×1 a máme jedno z požadovaných vydláždení $4 \times n$. Naša úloha sa teda zmenila na to, že potrebujeme nájsť počet dláždení plochy $4 \times (n - 1)$. Ale práve toto vieme vyriešiť naším trikom, túto otázku jednoducho opäť pošleme do továrne a niekto to vypočíta za nás.

Na vyriešenie tejto úlohy je teda potrebné iba poslať továrni otázku "Koľko existuje rôznych dláždení plochy $4 \times (n - 1)$", počkať si na odpoveď, a potom to isté číslo použiť ako odpoveď na pôvodnú otázku.

Podúloha c)

Zdá sa, že táto podúloha bude veľmi podobná tej predchádzajúcej. A takmer tomu tak je, netreba sa však nechať pomýliť. Ak má byť v poslednom stĺpci dlaždica 4×4 je to skutočne to isté. Je totiž len jediná možnosť ako tam túto dlaždicu uložiť. To čo nám ostane je navyše plocha $4 \times (n - 4)$, na ktorej počet vyplnení sa vieme spýtať pomocou nášeho triku.

Pri dlaždici 3×3 je to o niečo zložitejšie. Jednak sú dve možnosti kam ju vieme v poslednom stĺpci položiť. Navyše však to čo nám ostane nie je zarovnaná plocha veľkosti $4 \times (n - 3)$, buď nad alebo pod dlaždicou 3×3 máme totiž tri prázdne miesta. Uvedomme si však, že na ne nič iné ako dlaždice 1×1 ísť nemôžu. Po ich jednoznačnom vyplnení teda dostaneme plochu $4 \times (n - 3)$, na ktorej počet vydláždení sa môžeme spýtať cez trik. Tu však netreba zabudnúť, že odpoveď, ktorú dostaneme musíme ešte vynásobiť 2, lebo máme dve možnosti ako k nej pridať dlaždicu 3×3 a tri dlaždice 1×1.

Aj pri dlaždici 2×2 to na prvý pohľad vyzerá jednoducho. Sú štyri možnosti ako vyplniť posledné dva stĺpce – jedna dlaždica 2×2 navrchu, v strede, naspodu alebo dve dlaždice – po ktorých vyplnení nám ostane plocha $4 \times (n-2)$, na ktorú sa opýtame pomocou triku. Toto však nestačí, na nejaké možnosti totiž zabúdame.

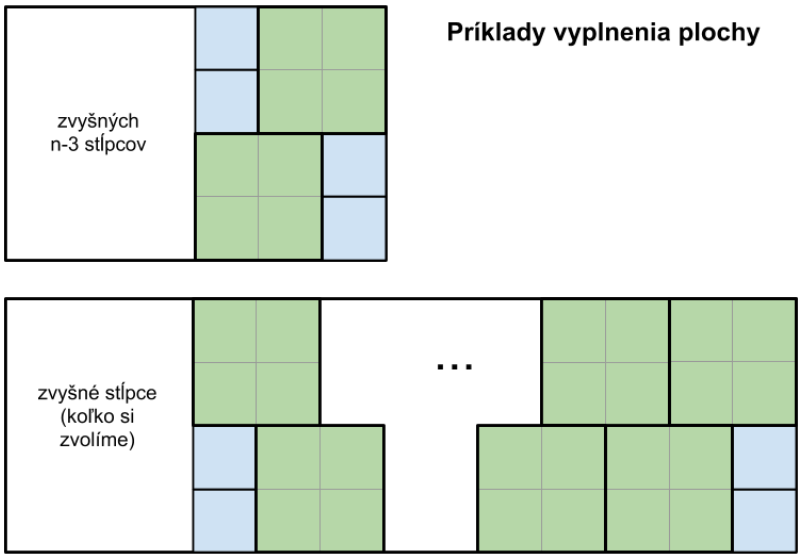
Predstavme si, že dlaždicu 2×2 dáme do posledného stĺpca navrch. Čo môže ísť pod ňu? Ďalšia dlaždica 2×2 alebo štyri dlaždice 1×1. Tieto možnosti sme však už zarátali. Okrem toho však môžeme dať dlaždice 1×1 iba do dvoch voľných políčok posledného stĺpca a vedľa nich uložiť ďalšiu dlaždicu 2×2. V tomto momente však máme aspoň čiastočne zaplnené tri stĺpce a možnosti vyplnenia $4 \times (n-2)$ s touto možnosťou nepočítali. Musíme sa teda zamyslieť čo spraviť. Jedna možnosť je doplniť dve voľné políčka v treťom stĺpci dlaždicami 1×1, zistiť počet dláždení pre plochy $4 \times (n-3)$ a pridať to k možnostiam. Akurát ďalšou možnosťou je tam dať ďalšiu dlaždicu 2×2 a tým si problém posunúť o ďalší stĺpec.

V skutočnosti môžeme dlaždice 2×2 pridávať takto na striedačku ľubovoľne dlho a vedie to k unikátnym možnostiam. V tomto prípade musíme teda započítať všetky tieto možnosti a opýtať sa výrazne viac otázok. Započítame totiž aj možnosti ktoré vzniknú z plochy 4×0, teda keď iba budeme takto nastriedačku dávať dlaždice 2×2, možnosti pre plochy 4×1, plochy 4×2 atď.

Označme si $P(m)$ počet dláždení plochy $4 \times m$. Potom odpoveďou na otázku "Koľko existuje rôznych dláždení plochy $4 \times n$ takých, že v poslednom stĺpci je aspoň jedna dlaždica 2×2" je nasledovná hodnota:

$$4P(n-2) + 2(P(n-3) + P(n-4) + P(n-5) + \cdots + P(1) + P(0))$$

Všimnite si krát dva, ktoré je tam kvôli symetrickosti týchto možností.



Podúloha d)

Táto podúloha sa už rieši ľahko, stačí spojiť to čo sme zistili v predchádzajúcich podúlohách. Všetky možné dláždenia plochy $4 \times n$ totiž vieme rozdeliť na štyri skupiny podľa toho, aké dlaždice dáme do jej posledného stĺpca. A počet rôznych dlaždení pre každú z týchto skupín sme zisťovali v podúlohách b) a c), takže výsledkom tejto podúlohy je jednoducho ich súčet. Opäť, použijúc označenie $P(n)$ pre počet dláždení plochy $4 \times n$, dostaneme výsledok

$$P(n) = \underbrace{P(n-1)}_{\text{pre dlazdicu } 1\times 1} + \underbrace{4P(n-2) + 2(P(n-3) + P(n-4) + \cdots + P(1) + P(0))}_{\text{pre dlazdicu } 2\times 2} + \underbrace{2P(n-3)}_{\text{pre dlazdicu } 3\times 3} + \underbrace{P(n-4)}_{\text{pre dlazdicu } 4\times 4}$$

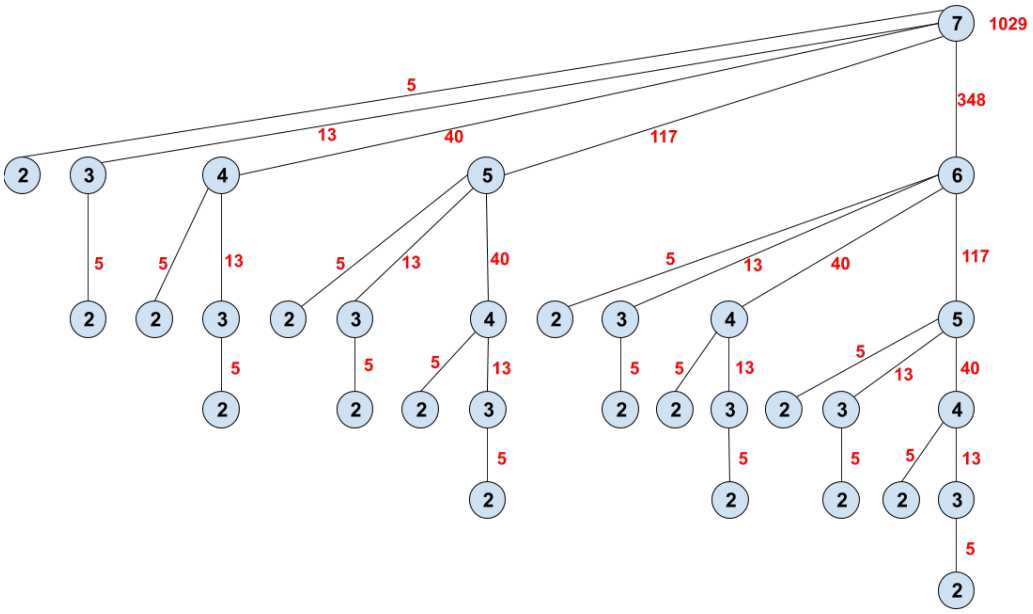
Podúloha e)

Našou úlohou je vypočítať ako bude prebiehať výpočet počtu dláždení pre plochu 4×7 ak vyššie uvedený postup budú využívať všetci zamestnanci továrne. K tomu si však potrebujeme ujasniť niekoľko detailov posielania správ. Napríklad vo vzorci vyššie môžeme vidieť, že tie isté hodnoty potrebujeme vo vzorci viackrát.

Nedáva však zmysel aby sme sa dvakrát pýtali na to, aká je odpoveď pre plochu 4×2. Spýtame sa to len raz a potom zistenú hodnotu použijeme dvakrát.

Takisto, na niektoré jednoduché úlohy nepotrebujeme posielať správu, lebo odpoveď poznáme. Napríklad ak nám vyjde, že sa máme pýtať na počet dláždení plochy, ktorej veľkosť je záporná hodnota, takáto otázka zjavne nemá zmysel a jej odpoveďou je 0. V prípade plochy 4×0 je odpoveď 1 – máme práve jednu možnosť ako vyplniť prázdnu plochu a to nespraviť nič. A rovnako pre plochu 4×1 je len jedna možnosť – použitie dlaždíc 1×1.

Dodržiac tieto pravidlá dostaneme nasledovný obrázok. Čísla v krúžkoch označujú veľkosť problému, ktorý sa akutálne rieši. Čierne čiary znázorňujú posielané správy. Červené čísla zase výsledok, ktorý je poslaný späť. Odpoveďou pre plochu 4×7 je teda 1029.



Podúloha f)

V skutočnosti existujú až dve zlepšenia, veľmi podobné, ktoré sa dajú zapracovať. Vo vzorovom riešení spomenieme obe, ale na získanie bodov stačilo ľubovoľné z nich.

Pri pohľade na obrázok vyššie by malo byť úplne jasné, čo sa robí zbytočne. Neustále dookola počítame tie isté čísla. Počet dláždení pre plochu 4×2 počítame dokopy až 16 krát. Pritom odpoveď je zakaždým rovnaká a nezmení sa ani po stom počítaní. To by možno ešte nebol najväčší problém, lebo však vypočítať tie plochy 4×2 je ľahké. Problém je však pri väčších plochách. Lebo zakaždým keď napríklad počítame plochu 4×5 nepridá sa nám jedno počítanie ale celá tá veľká časť podtým, teda 7 zbytočných výpočtov. A to už problém je.

Ako to teda vyriešiť? Použijeme korkovú tabuľu v továrni a jednoducho si na ňu budeme značiť už vypočítané hodnoty. Prvý zamestnanec, ktorý vypočíta počet dláždení plochy $4 \times m$ to zaznačí na tabuľu. Následne, každý zamestnanec, ktorý dostane za úlohu vypočítať počet dláždení plochy $4 \times x$ sa najskôr pozrie, či už hodnota pre plochu $4 \times x$ nie je na tabuli. Ak áno, jednoducho ju použije a ušetrí sebe aj zvyšným kopec práce. A ak tam nie je, tak ju začne počítať.

V reálnom svete ešte aj tu môže nastať problém. Čo ak začne naraz počítať hodnotu pre plochu $4 \times m$ viacero ľudí. Hodnota ešte nie je na tabuli, lebo ju nikto nezistil, a preto na nej všetci začnú zbytočne pracovať. Potrebovali by sa preto dohodnúť, že prvý kto chce vypočítať plochu $4 \times m$, zaznačí na tabuľu, že na tom pracuje. Ostatní by túto správu videli a počkali na výsledok.

Tento problém je však spôsobený len tým, že viacerí ľudia môžu počítať rôzne veci. V počítači, kde by sme túto techniku chceli použiť sa však môže vykonávať naraz najviac jeden výpočet (ak nezačneme operovať s viacjadrovými procesormi, ale o tom inokedy) a tento problém neexistuje.

Druhé vylepšenie je podobné a týka sa tej škaredej sumy $2(P(n-3)+P(n-4)+\cdots+P(1)+P(0))$, ktorá sa v našom vzorci vyskytuje. Spočítať viacero čísel dokopy je zdĺhavé, úmerne ich počtu, a toto počítanie tiež robí veľa ľudí opakovane. Na tabuľu si teda zamestnanci môžu značiť aj tieto súčty, ktorým sa ostatní určite potešia.

Ak by sme riešenie tejto úlohy naprogramovali, vyzeralo by nasledovne. Ak viete aspoň trochu programovať, pozrite sa ako elegantne sa to dalo riešiť a ako naša továreň predstavovala "rekurzívnu funkciu". A ak sa s pojmom "rekurzia" stretnete niekedy v budúcnosti, skúste si pre lepšiu predstavu vybaviť továreň na dlaždičky :)

Listing programu (Python)

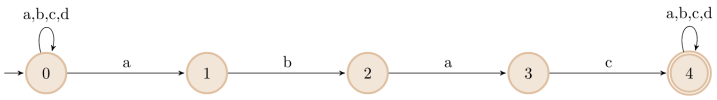
```
# -1 znamena, ze dana hodnota este nebola vypocitana
tabula_vysledky = [-1]*100
tabula_sucty = [1, 2]

# potrebujeme vypocitat pocet dlazdeni plochy 4xn
def tovaren(n):
    if n < 0:
        return 0
    if n < 2:
        return 1
    # niekto tuto ulohu pocital pred nami
    if tabula_vysledky[n] != -1:
        return tabula_vysledky[n]
    pocet = tovaren(n-1) + 4*tovaren(n-2) + 2*tovaren(n-3) + tovaren(n-4)
    if n-3 >= 0:
        pocet += 2*tabula_sucty[n-3]
    # vypocitame sucet prvych n hodnot
    tabula_sucty.append(tabula_sucty[-1] + pocet)
    # zapamatame si a posleme odpoved
    tabula_vysledky[n] = pocet
    return pocet

n = int(input())
print(tovaren(n))
```

2. Rizikové Receptomaty

vzorák napísal(a) Adam
(max. 15 b za riešenie)

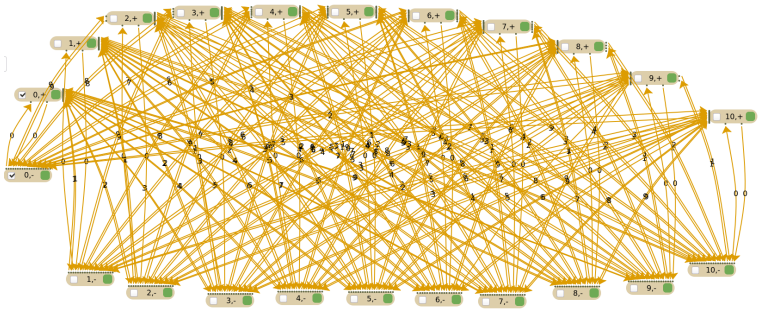
Podúloha a)

Keďže náš receptomat je nerozhodný, stačí nám držať ukazovateľ na počiatočnom stave a do ďalšieho prejsť pri vyhovujúcom znaku. Posledný stav má prechod do seba so všetkými znakmi – keď raz zistíme že sa na vstupe podreťazec `abac` nachádza, chceme si túto informáciu držať až do konca (pomocou akceptovaného stavu).



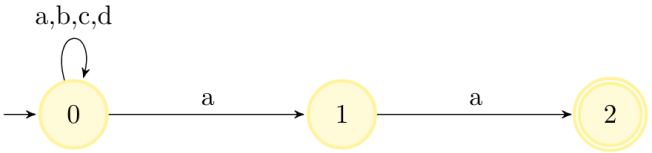
Podúloha b)

Toto bola ťažšia úloha ako sa na prvý pohľad zdalo. Deliteľnosť číslom 11 je ekvivalentná s deliteľnosťou rozdielu súčtov cifier na párnych a nepárnych miestach (napr. pre 6 ciferné číslo nás zaujíma (súčet cifier na prvom, treťom a piatom mieste) mínus (súčet cifier na druhom, štvrtom a šiestom mieste)). Budeme si teda držať aktuálny zvyšok po delení 11 a striedavo pričítavať/odčítavať cifry (príklad: číslo 14641 = +1-4+6-4+1 = 0, číslo je deliteľné 11). Budeme mať teda 11 stavov pre reprezentáciu aktuálneho zvyšku po delení, ale potrebujeme si aj pamätať či odčítavame alebo pričítavame (teda 2*11 = 22 stavov). Následne medzi nimi už len urobíme všetky prechody.



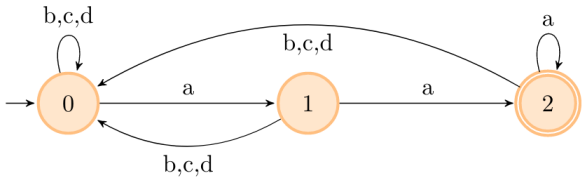
Podúloha c)

Tento receptomat je podobný ako v *Podúlohe a)*, ale akceptujúci stav necyklí pre iné znaky. Receptomat sa tak dostane do akceptujúceho stavu len vtedy, ak sú posledné 2 znaky `aa`.



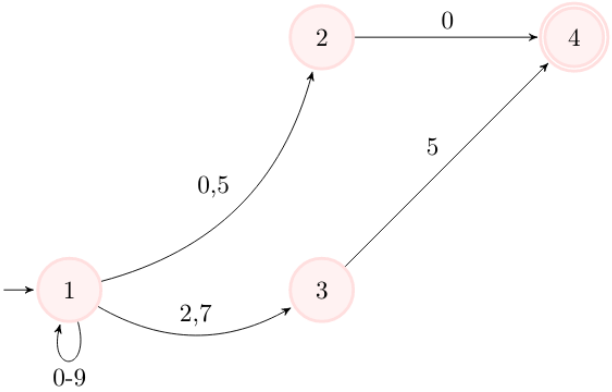
Podúloha d)

Tento receptomat je podobný ako z *Podúlohy d)* minulého kola, ale potrebujeme aby automat akceptoval len recepty končiace na tento reťazec. Posledný stav teda nebude cykliť pre ľubovoľný prechod ako v podúlohe z minulého kola, ale len pre `a`. Potom nám stačí spraviť prechody pre nesprávne znaky do počiatočného stavu a pre správne znaky spraviť jednoduchú (lineárnu) cestu.

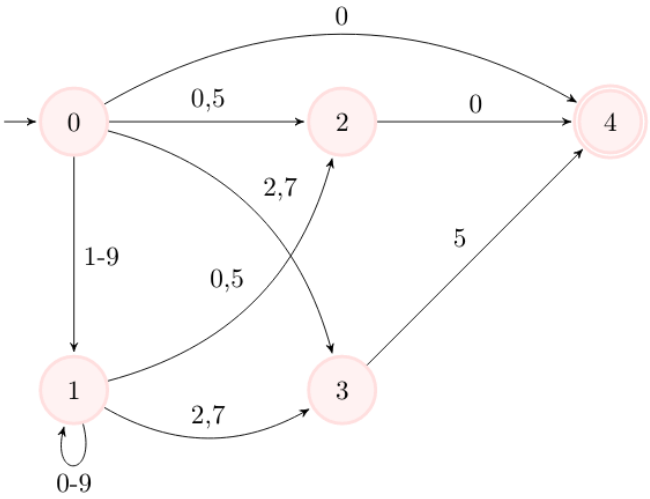


Podúloha e)

Číslo je deliteľné 25, ak sa končí na `00`, `25`, `50` alebo `75`. Vytvoríme teda receptomat, ktorý bude akceptovať tieto 4 zakončenia (to sú stavy 1, 2, 3 a 4).

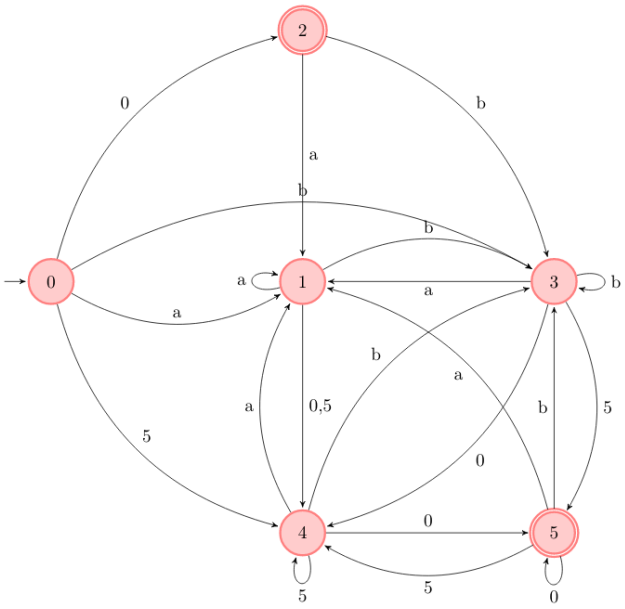


Okrem toho musíme ešte ošetriť prípad `0` (na to si pridáme stav 0).



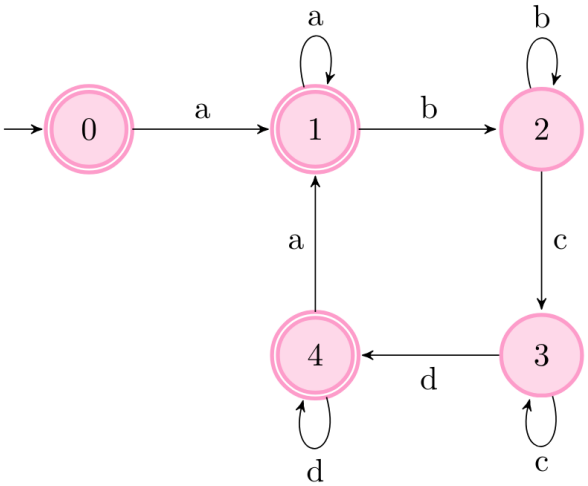
Podúloha f)

Začneme tým, že si ošetríme recept `0`. Ten je ošetrený stavom 2. Potom si budeme sledovať, či prišli znaky `00`, `25`, `50` alebo `75` (pomocou stavov 3, 4 a 5). Keď nám v ľubovoľnom stave príde nejaký zo znakov "1,3,4,6,8,9", vrátime sa do stavu podobnému počiatočnému, ale bez stavu 2 – ten ošetruje len recept `0`. V schéme je prechod "2,7" zjednodušený na "b" a "1,3,4,6,8,9" na "a", "5" a "0" ostávajú nezmenené (čiže napr. "0,a,b" by znamenalo "0,1,2,3,4,6,7,8,9").



Podúloha g)

Receptomat akceptuje recepty v tvare `(a!b!c!d!)*a*`, kde výkričník znamená "opakuj tento znak 1 alebo viac krát" a hviezdička znamená "opakuj tento znak 0 alebo viac krát". Na začiatku teda musíme mať akceptujúci stav, lebo aj prázdny recept je akceptovaný. Ďalej potrebujeme prechod, ktorý nám zaručí aspoň jeden znak `a`. Následne v našom pôvodnom receptomate len "pootočíme" šípky – tým spôsobíme, že z jednotlivých stavov nepôjdu 2 prechody s rovnakým znakom. Receptomat teda bude deterministický. Nakoniec ešte spravíme aj stav 4 akceptujúci, lebo recept sa nemusí končiť znakom `a`.



3. A tak sa Denis stratil

vzorák napísal(a) Prefix
(max. 15 b za riešenie)

Riešenia jednotlivých podúloh

1. Heslo sa nachádza v súbore. Keď zadáme `cat heslo.txt`, vidíme, že tam je množstvo náhodných znakov. Máme dve možnosti - pomocou príkazu `strings` vieme vyhľadať len čitateľné texty. Tento príkaz má aj parameter `-n`, ktorý špecifikuje, koľko minimálne znakov musí mať text. Takže vieme použiť príkaz `strings -n 10 heslo.txt`. Druhá možnosť bola použiť `grep` s vhodným regulárnym výrazom.

2. Vidíme, že heslo je 2 krát base64 enkódované. Na dekódovanie ho teda "pipeneme" cez príkaz `base64`. Kompletný príkaz je teda `echo 'Ympkek1HRnZNbXd3YkRGdU9EbHFNUT09Cg==' | base64 -d | base64 -d`

1. Vidíme heslo, ale po prečítaní nevieme zistiť, čo v ňom je a ani grep ho nevie nájsť. Príkaz na zistenie typu súboru je jednoduchý - `file`. Takže vieme použiť `file heslo` a zistíme, že sa jedná o JPEG obrázok. Keďže na termináli nevieme otvárať obrázky, tak môžeme použiť trik, ktorý bol popísaný v predošlej úlohe a stiahnuť si ho na počítač.

4. Toto bola len demonštrácia toho, že v suid programoch vieme mať iné práva. Keďže máme spustený suid príkazový riadok ako používateľ level4, vieme prečítať obsah súboru pomocou `cat`.

5. Prečítame si program. Napriek tomu, že je v jazyku C je celkom priamočiary. Program si od používateľa vypýta číslo, a ak je toto číslo rovné kľúču, program spustí `system(„cat ./heslo.txt")`, čím prečíta a vypíše heslo.

6. Opäť sme v podobnej situácii ako v predošlom príklade. Teraz ale nevidíme výstup programu, takže nám nestačí napísať `cat heslo.txt`. Existuje veľa možných riešení, napríklad spustiť príkaz `chmod a+r heslo.txt`, čím sme povolili všetkým používateľom čítať súbor.

7. Napriek tomu, že heslo nevieme prečítať vieme zmeniť jeho názov. Takže `mv heslo.txt x` a následne pomocou programu prečítame `x`.

8. Autor programu nám chcel zamedziť čítať súbory z iných priečinkov tým, že nám zadal začiatok cesty `cat ./tuniejeheslo/`. Našťastie sa mu to nepodarilo, pretože vieme z toho vytvoriť príkaz `cat ./tuniejeheslo/../tujeheslo/heslo.txt`.

9. Pomocou príkazu `sleep` vieme počkať pár sekúnd, nevieme ale čítať súbory. Vieme to vyriešiť napr. znakom `;`. Ak vložíme `1;bash`, tak z toho vznikne `sleep 1;bash`. Teda najprv počkáme sekundu a potom sa nám otvorí príkazový riadok s právami používateľa `level9`. Problém, s ktorým ste sa mohli stretnúť boli medzery - `scanf` skončí čítanie pri medzere, čo znamená že napr. `1;cat heslo.txt` nefungovalo a bolo treba nájsť nejakú cestu okolo toho.

10. Nevieme síce vylistovať obsah priečinku, to ale neznamená, že nevieme čítať súbory v ňom. Musíme však poznať ich názov. Našťastie možnosti pre názov tu nie je veľa. `for m in {00..12}; do for d in {00..31}; do cat „./obsah_tohto_priecinku_je_tajny/backup-$d-$m-18" 2>/dev/null; done; done;` prejde cez všetky možnosti v roku 2018. Pomocou `2>/dev/null` nebudeme vypisovať chybové hlášky a vypíše sa nám heslo.

4. Skladanie rebríkov

vzorák napísal(a) Kalerab
(max. 0 b za riešenie)

Fungujúce, pomalé riešenie

Jedno z prvých riešení, čo by nám mohli napadnúť, je vyskúšať každú dvojicu a pozrieť sa, či má požadovanú výšku. Treba sa však aj pozrieť, či nie je samostatný rebrík, ktorý by mal správnu výšku. V najhoršom prípade

overíme všetky dvojice rebríkov. Dvojíc je spolu $(n-1)+(n-2)+\ldots+3+2+1$, pretože prvý rebrík môžeme spárovať s $(n-1)$ zvyšnými, druhý s $(n-2)$, atď. Ide teda o súčet čísel od 1 po $(n-1)$[1]

$$1+2+3+\cdots+N=\frac{N(N+1)}{2}$$

Takýto počet operácií potom zaokrúhlime na N^2 a povieme, že časová zložitosť je $O(N^2)$. Pamäťová zložitosť je $O(N)$, pretože náš program si okrem konštantne veľa pomocných premenných pamätá iba vstupných N čísel.

Listing programu (Python)

```
ciel, pocet_rebrikov = list(map(int, input().split(" ␣ "))) # nacitame ciel a pocet rebrikov do premennych

rebriky = []
for vyska in input().split(): # nacitame vysky rebrikov
    rebriky.append(int(vyska))
je_riesenie = False # zacneme s predpokladom, ze riesenie neexistuje

if ciel in rebriky:
    je_riesenie = True # ak je rebrik spravnej vysky medzi rebrikmi, nic netreba skladat
else:
    for i in range(pocet_rebrikov): # prvy rebrik dvojice
        for j in range(i + 1, pocet_rebrikov): # druhy rebrik, zacneme rebrikom i+1, lebo vsetky predtym sme skusali
            if rebriky[i] + rebriky[j] == ciel:
                je_riesenie = True
                break
        if je_riesenie:
            break

if je_riesenie:
    print('ano')
else:
    print('nie')
```

Optimálne riešenie

Toto riešenie ťaží z faktu, že rebríky sú zoradené. Teraz môžeme implementovať takzvaných bežcov. Budú dvaja, nazvem ich ľavý a pravý. Každý bežec vždy stojí na jednom rebríku. Ľavý začne na prvom rebríku, pravý na poslednom. Predpokladáme, že rebríky sú zoradené vzostupne, teda najmenší je prvý. Postupne sa budú pohybovať smerom k sebe. Vždy sa pozrieme na súčet výšok rebríkov, na ktorých bežci práve stoja. Ak je to požadovaná výška, tak sme skončili a vypíšeme áno. Ak to tak nie je, tak musíme posunúť bežca. Ak je súčet výšok rebríkov viac, ako chceme, musíme tento súčet zmenšiť. Keďže bežci sa môžu pohybovať iba smerom k sebe, musí sa posunúť pravý bežec, pretože celkový súčet sa má zmenšiť, a najväčšie rebríky sú napravo. Naopak, ak je súčet malý, posunie sa ľavý bežec. Toto opakujeme, až kým sa bežci nestretnú, alebo kým nenájdeme správnu kombináciu.

V našom riešení pomocou bežcov pri jednom kroku while cyklu buď nájdeme riešenie, alebo posunieme jedného z bežčov. V najhoršom prípade sa tak bežci stretnú po $(n-1)$ prechodoch while cyklu, takže časová zložitosť tohto riešenia je $O(N)$. Pamäťová zložitosť je rovnako ako pri pomalšom riešení $O(N)$ pretože si musíme zapamätať veľkosti rebríkov. Rýchlejšie riešenie ako $O(N)$ nevymyslíme, pretože program musí aspoň načítať vstup, ktorý obsahuje N čísel, takže už samotné načítanie vstupu nám zaberie $O(N)$ krokov.

Listing programu (C++)

```
#include <iostream>
#include <vector>
#include <map>

using namespace std;

int main()
{
    int vyska, pocet_rebrikov;
    vector<int> rebriky;
    cin >> vyska >> pocet_rebrikov; // nacitame vysku a pocet rebrikov
    bool je_riesenie = false; // oznacuje, ci sme uz spravnu kombinaciu nasli
    for (int i = 0; i < pocet_rebrikov; i++) // nacitame vysky rebrikov
    {
        int r;
        cin >> r;
        if (r == vyska) // ak je toto vyska, ktoru hladame, mame vysledok
          je_riesenie = true;
        rebriky.push_back(r);
    }
```

[1] Pre súčet prvých N prirodzených čísel platí $1+2+...+N=\frac{N(N+1)}{2}$. Skúsme pre párne N sčítať prvé číslo s posledným, druhé s predposledným, atď. Výsledok bude zakaždým $N+1$. Dvojíc s takýmto súčtom sme vytvorili N, a preto je celkový výsledok $\frac{N(N+1)}{2}$. Skúste si rozmyslieť podobnou úvahou, prečo vzorec platí aj pre nepárne N.

```
    int lavy = 0, pravy = pocet_rebrikov - 1;

    if(!je_riesenie)
    {
        while (lavy < pravy) // kym sa nestretnu
        {
            // ak sme nasli spravnu kombinaciu, mame odpoved
            if (rebriky[lavy] + rebriky[pravy] == vyska)
            {
                je_riesenie = true;
                break;
            }
            if (rebriky[lavy] + rebriky[pravy] >= vyska) // posunieme bezcov
                pravy--;
            else
                lavy++;
        }
    }

    if (je_riesenie)
        cout << "ano" << endl;
    else
        cout << "nie" << endl;
    return 0;
}
```

Listing programu (Python)

```
# nacitame ciel a ppocet rebrikov do premennych
ciel, pocet_rebrikov = list(map(int, input().split("␣")))
rebriky = []

# nacitame vysky rebrikov
for vyska in input().split():
    rebriky.append(int(vyska))
# zacneme s predpokladom, ze riesenie neexistuje
je_riesenie = False
# ak je rebrik spravnej vysky medzi rebrikmi, nic netreba skladat
if ciel in rebriky:
    je_riesenie = True
else:
    lavy = 0 # lavy bezec zacne na zaciatku, pravy na konci pola
    pravy = pocet_rebrikov - 1
    # opakujeme kym sa bezci nestretnu alebo kym nenajdeme dvojicu
    while lavy < pravy:
        # ak sme nasli spravnu kombinaciu, mame odpoved
        if rebriky[lavy] + rebriky[pravy] == ciel:
            je_riesenie = True
            break
        # posunieme bezcov podla velkosti
        if rebriky[lavy] + rebriky[pravy] > ciel:
            pravy = pravy - 1
        else:
           lavy = lavy + 1

if je_riesenie:
    print('ano')
else:
    print('nie')
```

5. Kvalita výberu

vzorák napísal(a) Dano
(max. 15 b za riešenie)

Zjednodušene povedané, na vstupe sme dostali mriežku čísiel a niekoľko obdĺžnikových plôch na tejto mriežke. Našou úlohou bolo pre každú zadanú obdĺžnikovú plochu zrátať súčet čísiel na nej.

Priamočiare riešenie

Prvé, čo by nám malo napadnúť je, že každý zadaný obdĺžnik prejdeme políčko po políčku a takto si ho celý nasčítame.

Každný obdĺžnik teda spracujeme v $O(n^2)$ a obdĺžnikov bolo q, takže časová zložitosť tohto riešenia je $O(qn^2)$. Pamätáme si iba mriežku, takže pamäť bude $O(n^2)$.

Listing programu (C++)

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

using namespace std;

const int MAXN = 2001;
int rola[MAXN][MAXN];

int main() {
    ios_base::sync_with_stdio(false); cin.tie(0); cout.tie(0); // rychle nacitavanie vstupu
    int n;
    cin >> n;
```

```
    for(int i = 0; i < n; i++)
        for(int j = 0; j < n; j++)
            cin >> rola[i][j];

    int q, r1, c1, r2, c2;
    cin >> q;
    while(q--) {
        long long odpoved = 0;
        cin >> r1 >> c1 >> r2 >> c2;
        for(int r = r1; r <= r2; r++)
            for(int c = c1; c <= c2; c++)
                odpoved += rola[r][c];
        cout << odpoved << '\n';
    }

    return 0;
}
```

Zrýchlenie

Využijeme známy trik známy ako prefixové súčty, alebo prefixové sumy. V čom spočíva? To sa dozviete v tomto článku v KSP kuchárke.

Keď už vieme, čo sú to prefixové súčty, tak by nemal byť problém použiť ich na riešenie tejto úlohy.

Zrátame si prefixové súčty pre každý riadok mriežky samostatne. Keď teraz chceme zistiť výsledok pre nejaký obdĺžnik, tak nemusíme prechádzať všetky jeho políčka. Stačí nám prejsť všetky jeho riadky. Pre každý jeho riadok totiž z prefixových súčtov vieme zrátať, koľko daný riadok pridá ku súčtu.

Ako sme si pomohli? Každý obdĺžnik teraz už vieme spočítať v čase $O(n)$. Zrátanie prefixových súčtov nám dokopy na začiatku zaberie čas $O(n^2)$. Obdĺžnikov je stále q, takže celkovo sme čas zlepšili na $O(n^2 + qn)$. Pamäť nemáme ako zlepšovať, je stále $O(n^2)$.

Listing programu (C++)

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

using namespace std;

const int MAXN = 2001;
int rola[MAXN][MAXN];

int main() {
    ios_base::sync_with_stdio(false); cin.tie(0); cout.tie(0); // rychle nacitavanie vstupu
    int n;
    cin >> n;
    for(int i = 0; i < n; i++)
        for(int j = 0; j < n; j++)
            cin >> rola[i][j];

    // Zratame prefixove sucty pre kazdy riadok
    for(int i = 0; i < n; i++)
        for(int j = 1; j < n; j++)
            rola[i][j] += rola[i][j - 1];

    int q, r1, c1, r2, c2;
    cin >> q;
    while(q--) {
        long long odpoved = 0;
        cin >> r1 >> c1 >> r2 >> c2;
        for(int r = r1; r <= r2; r++) {
            odpoved += rola[r][c2]; // pripocitame prefix riadku az po koniec obdlznika
            if(c1 > 0)
                odpoved -= rola[r][c1 - 1]; // odpocitame prefix riadku, ktory este nepatri do obdlznika
        }
        cout << odpoved << '\n';
    }

    return 0;
}
```

Vzorové riešenie

Od predchádzajúcej myšlienky už ku vzorovému riešeniu nechýba veľa. Stačí myšlienku prefixových súčtov rozšíriť do dvoch rozmerov. Môžete si skúsiť vymyslieť, ako by to mohlo fungovať. Ak sa vám nechce vymýšľať, môžete si prečítať článok o 2D prefixových súčtoch z KSP kuchárky.

Vzorové riešenie tejto úlohy je potom už jednoduché. Na mriežke si zrátame 2D prefixové súčty a na každú otázku potom budeme vedieť odpovedať v $O(1)$. Zrátanie 2D prefixových súčtov nám bude trvať $O(n^2)$. Celkový čas teda bude $O(n^2 + q)$. Pamäť ostane $O(n^2)$.

Listing programu (C++)

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

```cpp
using namespace std;

const int MAXN = 2001;
int rola[MAXN][MAXN];

int main() {
    ios_base::sync_with_stdio(false); cin.tie(0); cout.tie(0); // rychle nacitavanie vstupu
    int n;
    cin >> n;
    for(int i = 0; i < n; i++)
        for(int j = 0; j < n; j++)
            cin >> rola[i][j];
    // Zratame 2D prefixove sucty
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            if(i != 0)
                rola[i][j] += rola[i - 1][j];
            if(j != 0)
                rola[i][j] += rola[i][j - 1];
            if(i != 0 && j != 0)
                rola[i][j] -= rola[i - 1][j - 1];
        }
    }

    int q, r1, c1, r2, c2;
    cin >> q;
    while(q--) {
        // Priamociaro vyuzijeme 2D prefixy pre ziskanie odpovede
        long long odpoved = 0;
        cin >> r1 >> c1 >> r2 >> c2;
        odpoved += rola[r2][c2];
        if(r1 > 0)
            odpoved -= rola[r1 - 1][c2];
        if(c1 > 0)
            odpoved -= rola[r2][c1 - 1];
        if(r1 > 0 && c1 > 0)
            odpoved += rola[r1 - 1][c1 - 1];
        cout << odpoved << '\n';
    }

    return 0;
}
```

Listing programu (Python)

```python
n = int(input())
rola = []
for _ in range(n):
    rola.append([int(x) for x in input().split()])

# Zratame 2D prefixove sucty
for r in range(n):
    for c in range(n):
        if r > 0:
            rola[r][c] += rola[r - 1][c]
        if c > 0:
            rola[r][c] += rola[r][c - 1]
        if r > 0 and c > 0:
            rola[r][c] -= rola[r - 1][c - 1]

q = int(input())
res = []
for _ in range(q):
    # Z 2D prefixovych suctov priamo zistime odpoved
    r1, c1, r2, c2 = [int(x) for x in input().split()]
    ans = rola[r2][c2]
    if r1 > 0:
        ans -= rola[r1 - 1][c2]
    if c1 > 0:
        ans -= rola[r2][c1 - 1]
    if r1 > 0 and c1 > 0:
        ans += rola[r1 - 1][c1 - 1]
    res.append(ans)

print(*res, sep='\n')
```