A-I-1 Zatopené mesto

Ak si uvedomíme, že Kubíkovi sa nikdy neoplatí vrátiť na miesto, kde už bol, ostane nám len konečný počet možných ciest zo školy domov (je ich približne $n!$) a môžeme úlohu za pár bodov vyriešiť hrubou silou – všetky možné cesty vygenerujeme a skontrolujeme.
Na väčší počet bodov bolo potrebné úlohu vedieť vyriešiť v polynomiálnej časovej zložitosti. Ukážeme si dve takéto riešenia.

Priamočiarejšie pomalšie riešenie

Obe riešenia používajú ako nástroj algoritmus prehľadávania grafu do šírky. Detailný popis tohto algoritmu nájdete v Kuchárke KSP: `https://www.ksp.sk/kucharka/bfs/`.

Prvé riešenie je myšlienkovo priamočiarejšie: keď sa pozeráme, ako sa Kubík hýbe po meste, tak jeho aktuálny stav vieme popísať dvoma číslami: prvým bude križovatka, kde sa práve nachádza, a druhým číslo popisujúce, či a ako dlho už je mokrý.
Takto dostávame nový graf. Jeho vrcholy sú všetky možné Kubíkove stavy a jeho hrany zodpovedajú možným nasledujúcim pohybom: Kubík prejde nejakou ulicou, čím sa dostane na novú križovatku a tiež vieme povedať, ako sa mu zmení „mokrosť“.
V takomto grafe nás zaujíma existencia a dĺžka najkratšej cesty zo stavu „Kubík je v škole a je suchý“ do niektorého zo stavov „Kubík je doma a je nanajvýš k minút mokrý“. Túto vieme nájsť prehľadávaním do šírky.
Aký veľký je tento graf, a teda akú má toto riešenie časovú zložitosť? Vrcholov máme $O(nk)$ – je n lokalít a na každej môže byť Kubík v jednom z rádovo k rôznych stavoch mokrosti. Podobne hrán máme dokopy $O(mk)$, keďže aj každou z m ulíc sa môže Kubík vybrať v každom možnom stave mokrosti. Celková časová zložitosť prehľadávania bude teda $O((m+n)k)$. Takéto riešenie malo získať približne polovicu bodov.

Vzorové riešenie

Úlohu vieme riešiť ešte efektívnejšie. Rozdeľme Kubíkovu cestu na dve časti: suchú (po okamih, kedy prvýkrát stúpi na zatopenú križovatku, resp. príde suchý domov) a mokrú (od toho okamihu ďalej). Čo o nich vieme povedať?
Akonáhle je Kubík mokrý, je mu už jedno, ktoré križovatky sú zatopené a ktoré nie. Jediné, čo chce, je dostať sa čo najrýchlejšie domov. Toto si vieme spočítať jedným prehľadávaním do šírky na pôvodnom grafe. Presnejšie, spustíme prehľadávanie do šírky **z Kubíkovho domova** a pre každý vrchol si spočítame, ako najrýchlejšie sa z neho vie Kubík dostať domov.
Podobne vieme druhým prehľadávaním do šírky (tentokrát začínajúc v škole) zistiť pre každú križovatku, či a ako najrýchlejšie ju vieme dosiahnuť suchou cestou. Pri tomto prehľadávaní je zakázané pohnúť sa zo zatopenej križovatky ďalej.
No a teraz už potrebujeme len spojiť obe časti. Postupne pre každú križovatku x spravíme nasledovné: Najskôr skontrolujeme, či sa z nej Kubík vie za x minút dostať domov. Ak áno, sčítame najkratší čas, za ktorý sa vie dostať suchou cestou zo školy sem a najkratší čas, za ktorý sa vie ľubovoľnou cestou dostať odtiaľto domov. Takto dostaneme nejaké platné riešenia. Najlepšie z nich je zjavne celkovo najlepším riešením našej úlohy.
(V predchádzajúcom odseku by stačilo ako x uvažovať Kubíkov dom a všetky zatopené križovatky. Riešenie však rovnako dobre funguje, aj keď za x postupne vyskúšame úplne všetky križovatky.)

Iné vzorové riešenie

Ešte pohodlnejšiu implementáciu dostaneme, keď si rozmyslíme, že celú úlohu vieme riešiť jedným prehľadávaním zadaného grafu do šírky. Stačí na to spraviť dve úpravy. Prvú už máme: budeme začínať nie zo školy, ale od Kubíkovho domu. No a druhá úprava bude nasledovná: keď sme už počas prehľadávania aspoň k minút od Kubíkovho domu, vieme, že už sme v tej časti jeho cesty, na ktorej musí byť suchý. Preto akonáhle počas prehľadávania spracúvame vrcholy vo vzdialenosti k a viac od Kubíkovho domu, už z nich nesmieme pokračovať do zatopeného vrchola.
Obe verzie vzorového riešenia len niekoľkokrát prehľadajú do šírky graf zadaný na vstupe. Ich časová zložitosť je preto $O(m+n)$.

Listing programu (C++)

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

int n, k;
vector<bool> zatopeny;
vector< vector<int> > e;
vector<int> d;

int main(){
        int m;
        cin >> n >> m >> k;

        zatopeny.resize(n);
        e.resize(n);
        d.resize(n);

        for(int i = 0; i < n; ++i){
                int z;
                cin >> z;
                zatopeny[i] = (z==1);
        }

        for(int i = 0; i < m; ++i){
                int a,b;
                cin >> a >> b;
                e[a].push_back(b); e[b].push_back(a);
        }

        for(int i = 0; i < n; ++i) d[i] = n+47;

        queue<int> q;
        d[n-1]=0;
        q.push(n-1);

        while(!q.empty()){
                int v = q.front();
                q.pop();
                int dv = d[v];

                for(auto w : e[v]){
                        if (d[w]<=dv+1) continue;
                        if (dv>=k && zatopeny[w]) continue;
                        d[w] = dv+1;
                        q.push(w);
                }
        }

        if(d[0] < n) cout << d[0] << endl; else cout << "-1" << endl;
}
```

A-I-2 Román

Táto úloha mala viacero rôzne efektívnych riešení. Napríklad sa dalo získať 6 bodov pomocou dynamického programovania: Postupne pre každé i zistíme p_i: koľko najviac vyvážených kníh vieme vyrobiť z prvých i kapitol románu. Toto zistíme tak, že vyskúšame všetky možnosti, koľko kapitol tvorí poslednú knihu. Ak by ich bolo j, tak by platilo $p_i = p_{i-j} + 0$ alebo $p_i = p_{i-j} + 1$ podľa toho, či by podsledná kniha nebola alebo bola vyvážená. Postupne pre každé j zistíme, ako dobré riešenie dostaneme, a za p_i vyberieme najlepšie z nich. Rozpracovaním tohto prístupu dostaneme riešenie s časovou zložitosťou $O(n^2)$.

Riešení za plný počet bodov existuje viacero a veľmi výrazne sa líšia náročnosťou implementácie. V tomto vzorovom riešení si ukážeme postupnosť úvah, ktoré povedú k skutočne jednoducho implementovateľnému riešeniu. Základ použitej techniky spočíva v tom, že sa nesnažíme len nájsť nejaké ľubovoľné optimálne riešenie, ale akoby navyše sa zamýšľame nad tým, ako ho vieme čo najviac zjednodušiť.

Nuly

Začneme tým, že sa zamyslíme nad nulami. Tvrdíme, že vždy existuje optimálne riešenie, v ktorom každú nulu (neutrálnu kapitolu) vydáme ako samostatnú knihu.
Prečo toto môžeme spraviť? Predstavme si úplne ľubovoľné rozdelenie románu na jednotlivé knihy a všimnime si knihu, ktorá má v sebe (aspoň jednu) nulu a okrem nej ešte niečo ďalšie. Celá táto kniha možno je vyvážená a možno nie je. Ak ju ale rozdelíme na menšie knihy tesne pred a tesne za našou nulou, dostaneme dokopy dve

alebo tri nové knihy a z nich určite aspoň jedna bude vyvážená (tá obsahujúca len tú našu nulu). Vidíme, že sme počet vyvážených kníh nezmenšili.
Môžeme teda zobrať ľubovoľné optimálne riešenie a opakovaním vyššie uvedeného postupu ho postupne prerobiť na tiež optimálne riešenie, v ktorom je každá nula v samostatnej knihe.

Bez núl

V tomto okamihu sme vlastne dostali jednoduchšiu úlohu: o každom kuse románu, ktorý nám ostal po spracovaní núl, teraz vieme, že obsahuje samé veselé a smutné kapitoly – teda už máme len 1 a -1.
Opäť, uvažujme ľubovoľné rozdelenie takéhoto románu na knihy. Nové pozorovanie, ktoré môžeme spraviť, je nasledovné. Majme ľubovoľnú vyváženú knihu. Takáto kniha musí obsahovať aspoň jednu dvojicu *bezprostredne po sebe nasledujúcich kapitol*, z ktorých jedna je smutná a druhá veselá. (Ak sú každé dve po sebe idúce kapitoly rovnakého typu, tak sú úplne všetky kapitoly rovnakého typu a teda kniha nie je vyvážená.)
No a teraz môžeme ľubovoľnú vyváženú knihu s viac ako dvomi kapitolami rozdeliť na dve alebo tri menšie knihy: takúto dvojicu 1 a -1, všetko pred ňou a všetko za ňou. Dostaneme opäť aspoň jednu vyváženú knihu, a teda aspoň také dobré riešenie ako to, z ktorého sme začínali.
Pre zadanie, v ktorom sú len veselé a smutné kapitoly, teda platí nasledovné tvrdenie: existuje optimálne riešenie, v ktorom každú vyváženú knihu tvorí práve jedna veselá a jedna smutná kapitola.

Najviac vyvážených kníh

Predchádzajúce pozorovanie nám vlastne hovorí, že keď nájdeme najväčšiu možnú sadu kníh tvorených jednou smutnou a jednou veselou kapitolou, dostaneme optimálne riešenie. Posledným dielikom našej skladačky teda je otázka, ako takúto sadu nájsť.
Odpoveď je veľmi jednoduchá: stačí postupovať pažravo. Postupne si čítame náš román a vždy, keď stretneme po sebe (v ľubovoľnom poradí) ešte nepoužitú veselú a smutnú kapitolu, spravíme z nich oboch knihu.
Zdôvodniť správnosť tohto postupu je ľahké. Môže sa nám niekedy oplatiť takúto knihu nevyrobiť? No zjavne nie – ak by sme najbližšiu vyváženú knihu vyrobili neskôr, nemôžeme dostať lepšie riešenie, lebo po nej nám ostane nepoužitá len vlastná podmnožina tých kapitol, ktoré zatiaľ nepoužilo naše pažravé riešenie.

Zhrnutie

Našli sme špecifický tvar, v ktorom stačí hľadať optimálne riešenie. Odmenou nám je skutočne triviálna implementácia: stačí raz román prečítať a nájsť, koľko núl a koľko disjunktných dvojíc 1 a -1 obsahuje.

Listing programu (C++)

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

int main() {
    int n;
    cin >> n;
    vector<int> a(n);
    for (int i = 0; i < n; ++i) cin >> a[i];

    int odpoved = 0;
    for (int i = 0; i < n; ++i) {
        if (a[i] == 0) ++odpoved;
        else if (a[i] ==  1 && i+1 < n && a[i+1] == -1) { ++odpoved; ++i; }
        else if (a[i] == -1 && i+1 < n && a[i+1] ==  1) { ++odpoved; ++i; }
    }
    cout << odpoved << endl;
}
```

Pre zaujímavosť ešte uvádzame minimalistickú implementáciu celého riešenia v Pythone pomocou regulárnych výrazov:

```
import re
print( len( re.findall( '0|␣1␣-1|-1␣1', '␣'+input() ) ) )
```

A-I-3 Celta

Pre každý stánok i chceme vlastne zistiť číslo c_i s nasledovným významom: ak začneme celtu na ľavom konci stánku i, po ľavý koniec ktorého najpravejšieho stánku ju vieme natiahnuť? (Napravo od posledného stánku si ešte predstavíme fiktívny stánok číslo $n+1$, ktorý je odpoveďou, ak vieme celtu natiahnuť až po koniec.)
Ak poznáme číslo c_i, tak počet celtou pokrytých stánkov vypočítame jednoducho ako $c_i - i$, a teda optimálnym riešením úlohy je $\max_i(c_i - i)$.
Toto maximum vieme priamočiaro zistiť hrubou silou. Pre každé i budeme simulovať postupné naťahovanie celty – v cykle sčitujeme dĺžky priečelí stánkov, kým sa zmestia pod celtu. V najhoršom prípade má takéto riešenie časovú zložitosť $O(n^2)$ – postupne pre každý stánok prejdeme všetky napravo od neho.

Binárne vyhľadávanie

Postupnosť dĺžok priečelí si predspracujeme: zavedieme si súradnicovú sústavu a spočítame si pre každý stánok súradnicu, na ktorej začína. Tento výpočet vieme spraviť v lineárnom čase nasledovne: stánok 1 začína na súradnici $p_1 = 0$ a potom pre každé i platí, že stánok $i+1$ začína na súradnici $p_{i+1} = p_i + a_i$. (Hodnotám p_i zvykneme hovoriť *prefixové súčty* poľa a: každé p_i je súčtom nejakého prefixu pôvodnej postupnosti.)
Pomocou takto predpočítaných hodnôt vieme o ľubovoľnom úseku stánkov povedať, aký je dlhý: od začiatku stánku i po začiatok stánku j je vzdialenosť $p_j - p_i$. Hľadaná hodnota c_i je teda rovná najväčšiemu takému j, pre ktoré platí $p_j - p_i \leq k$ (t.j. „pokrytý úsek je nanajvýš taký dlhý ako celta“).
Pre ľubovoľné konkrétne i vieme najväčšie j s touto vlastnosťou nájsť binárnym vyhľadávaním. Takto dostávame riešenie s časovou zložitosťou $\Theta(n \log n)$.

Dva pointre

Existuje aj lineárne riešenie. To bude zodpovedať tomu, že keď nájdeme optimálne riešenie pre celtu začínajúcu na stánku i, tak ju nedáme úplne dole a nezačneme pre stánok $i+1$ odznova, ale ju akoby posunieme o príslušný kus doprava.
Formálnejšie bude toto riešenie vyzerať nasledovne: Všimneme si, že pre každé i platí $c_{i+1} \geq c_i$. Totiž ak vieme od začiatku stánku i celtu natiahnuť až po začiatok stánku c_i, tým skôr ju vieme natiahnuť od začiatku stánku $i+1$ po začiatok stánku c_i. (Máme pri tom vľavo o stánok menej, takže nám na pravom konci ostane o a_i metrov nepoužitej celty viac.) A teda c_{i+1} je buď presne rovná c_i, alebo sa teraz pod celtu ešte zmestia aj ďalšie stánky a vtedy $c_{i+1} > c_i$.
Predstavme si teraz, že na náš rad stánkov ukazujeme dvoma prstami: ľavý prst ukazuje na začiatok stánku i, ktorý práve spracúvame, a pravý prst na začiatok stánku c_i, kde aktuálne končí natiahnutá celta. Na začiatku riešenia postupne posúvame pravý prst, kým nenájdeme hodnotu c_1. Zvyšok riešenia vyzerá tak, že dokola opakujeme nasledovné dva kroky:

- Posuň ľavý prst o stánok doprava. (Ideme hľadať optimálne riešenie pre nasledujúci začiatok.)
- Posúvaj pravý prst doprava, kým stačí celta a neminú sa stánky.

Je zjavné, že takéto riešenie je korektné – pre každé i naozaj stále korektne nájdeme optimálne c_i. To, čo ale zjavné nie je, je jeho časová zložitosť. Na tej analýzu nám ale stačí správny uhol pohľadu. Namiesto detailného počítania, pre ktoré i koľkokrát pohneme ktorým prstom, sa pozrime na celý proces riešenia ako na jeden celok. Dôležité je všimnúť si, že každým prstom hýbeme **len doprava**. A keďže stánkov je len n (plus jeden fiktívny), tak **za celé riešenie dokopy** každým prstom pohneme doprava nanajvýš n-krát. Dokopy teda spravíme nanajvýš $2n$ operácií. Každú z nich vieme spraviť v konštantnom čase, celková časová zložitosť tohto riešenia je preto lineárna od n.

Listing programu (Python)

```
N, dlzka_celty   = [ int(_) for _ in input().split() ]
sirky_stankov    = [ int(_) for _ in input().split() ]

zaciatky_stankov = [ 0 ]
for x in sirky_stankov: zaciatky_stankov.append( zaciatky_stankov[-1] + x )
```

```
koniec, optimalny_pocet, optimalny_zaciatok = 0, 0, 0

for zaciatok in range(N):
    # posúvame koniec doprava, kým sa dá a kým celta dočiahne od začiatku po koniec
    while koniec < N and zaciatky_stankov[koniec+1] - zaciatky_stankov[zaciatok] <= dlzka_celty:
        koniec += 1

    # pozrieme sa, či je optimálne riešenie pre tento začiatok doteraz najlepším
    if koniec - zaciatok > optimalny_pocet:
        optimalny_pocet, optimalny_zaciatok = (koniec - zaciatok), zaciatok+1

print(optimalny_pocet, optimalny_zaciatok)
```

A-I-4 Externá pamäť

Podúlohy A a B ilustrujú skutočnosť, ktorú môžeme pozorovať aj v bežnej programátorskej praxi.
Dvojrozmerné pole si často predstavujeme ako tabuľku prvkov. Keď potrebujeme prejsť celé takéto dvojrozmerné pole, z matematického hľadiska je úplne jedno, či ho budeme prechádzať po riadkoch alebo po stĺpcoch. V oboch prípadoch spravíme presne rovnaké množstvo práce: práve raz spracujeme každý prvok. Matematik by teda mohol rozumne očakávať, že je úplne jedno, ktorú z týchto dvoch možností použije, keď potrebuje prechod takýmto poľom naprogramovať.
Ak ale vyskúšame obe možnosti naozaj spraviť v počítačovom programe, praktické testy nám ukážu, že program iterujúci po riadkoch je od programu iterujúceho po stĺpcoch merateľne rýchlejší. A to občas dosť výrazne – autor tohto textu sa už v podobných situáciách občas stretol až s faktorom okolo desať.
Vyskúšajte si to sami! Zaujímavé je skúšať rôzne veľkosti n a pozerať sa, ako sa pomer rýchlostí mení. Tento efekt je pozorovateľný už aj u obyčajných programov, ktoré majú celé pole v pamäti a nijak nepracujú s diskom – totiž už vtedy RAM hrá úlohu nášho „disku“ (externej pamäte), zatiaľ čo menšou a rýchlejšou internou pamäťou je cache procesora. Riešenie podúloh A a B v našom modeli vysvetľuje, ako a prečo vzniká v praxi rozdiel medzi týmito dvoma implementáciami.

Podúloha A

Môžeme plnohodnotne využiť *pamäť*. Z *disku* postupne čítame bloky a vždy, keď blok prečítame, celý jeho obsah vypíšeme. Časová zložitosť je zjavne $O(n^2)$, komunikačná zložitosť je $O(n^2/b)$, keďže každý z n^2/b blokov len raz načítame do pamäte.
Pri ukážke implementácie pripomíname, že v súlade so zadaním predpokladáme, že n je násobkom veľkosti bloku. (Aj keď v tomto konkrétnom prípade to veľký rozdiel nerobí, keďže aj všeobecné riešenie sa dá implementovať veľmi podobne – postupne čítam bloky do pamäte a vypisujem ich obsah, kým nevypíšem n^2 prvkov.)

Listing programu (Python)

```
# v pamäti už máme:    - konštantu B označujúcu veľkosť bloku na disku
#                      - premennú N obsahujúcu dĺžku strany matice (pričom N je násobkom B)

for riadok in range(N):
    for blok in range(N//B):
        data = Read( (N//B)*riadok + blok )
        for x in data: print(x)
```

Podúloha B

Keď cez dvojrozmerné pole predchádzame po stĺpcoch, situácia je o dosť smutnejšia. Pre n väčšie ako veľkosť jedného bloku platí, že každý prvok, ktorý potrebujeme vypísať, sa nachádza v inom bloku ako ten predchádzajúci, takže budeme musieť do pamäte načítať nový blok, aby sme sa ten nový prvok dozvedeli.
Ak by sme chceli nejak zmenšiť počet čítaní blokov z disku do pamäte, potrebovali by sme si niektoré načítané ale ešte nevypísané prvky vedieť „odkladať na neskôr“. Lenže my môžeme používať len $O(b)$ pamäte a pri takto malom množstve si vieme na neskôr odkladať len tak málo prvkov, že sa to na rýchlosti riešenia nijak merateľne neprejaví.

Presnejšie by sme tento argument mohli spraviť nasledovne: Každý prvok, ktorý si chceme v pamäti odložiť na neskôr, tam bude musieť čakať aspoň n krokov. Keďže dokopy máme n^2 krokov a naraz môže po ľubovoľnom z nich takto čakať v pamäti len rádovo b prvkov, dokopy si zvládneme odložiť a neskôr vypísať nanajvýš $bn^2/n = bn$ prvkov. A pre n rádovo väčšie ako b je n^2 rádovo väčšie ako bn, čiže ide len o zanedbateľný zlomok všetkých prvkov. Skoro každý prvok si teda bude vyžadovať samostatné čítanie z disku.
V nižšie uvedenej implementácii sa preto ani nepokúšame si nič extra pamätať a jednoducho zakaždým, keď pristupujeme k nejakému prvku, najskôr spravíme potrebné čítanie z disku. Toto riešenie má časovú aj komunikačnú zložitosť $O(n^2)$.

Listing programu (Python)

```
for stlpec in range(N):
    for riadok in range(N):
        blok_v_riadku = stlpec // B
        index_v_bloku = stlpec % B
        data = Read( (N//B)*riadok + blok_v_riadku )
        print( data[index_v_bloku] )
```

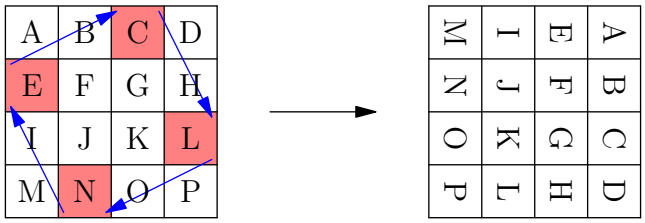
Po riešení podúlohy C si ukážeme, že podúloha B predsa len má aspoň o čosi lepšie riešenie, ak šikovne využijeme aj zápisy na disk. Vyššie uvedené riešenie ale stačilo na plný počet bodov za podúlohu B.

Realizácia čítaní a zápisov

Na tomto mieste sa ešte oplatí na chvíľu zamyslieť nad otázkou o časovej zložitosti predchádzajúceho riešenia. Totiž počas neho robíme až $O(n^2)$ čítaní z disku a pri každom z nich potrebujeme presunúť b údajov z disku do pamäte. Nebolo by teda fér hovoriť, že kvôli týmto operáciám má tento druhý algoritmus aj horšiu časovú zložitosť – až $\Theta(bn^2)$?
Ani bolo, ani nebolo. Asi najlepšie je spraviť práve to, čo sme spravili my pri definícii modelu, v ktorom riešite súčasné úlohy – čiže presúvanie dát po blokoch medzi diskom a pamäťou budeme rátať zvlášť. Pri odhade časovej zložitosti budeme za každý Read aj Write rátať len konštantne veľa času za realizáciu samotného volania funkcie. Konkrétna realizácia týchto operácií a tiež čas, ktorý trvá ich vykonanie, totiž závisí od konkrétneho hardvéru, ktorý náš program využíva. Je samozrejme možné, že niekedy bude každá takáto operácia realizovaná sekvenčne, a vtedy by náš odhad časovej zložitosti zodpovedal realite. No je rovnako dobre možné zostrojiť hardvér, ktorý bude vedieť dáta z bloku na disku do pamäte kopírovať paralelne, „všetky naraz“ a vtedy by sa čítania a zápisy na *asymptotickej* časovej zložitosti neprejavili.
No a aby sme boli úplne presní, môže sa stať, že tá konštanta, ktorá popisuje trvanie jedného čítania, resp. jedného zápisu, bude rádovo väčšia ako konštanty v presnej časovej zložitosti zvyšku programu. Zahrnúť oboje do toho istého asymptotického odhadu by mohlo byť zbytočne nepresné, obzvlášť v situáciách ako táto, kde očakávame, že celkový čas strávený prácou s diskom bude väčší ako celkový čas strávený vo zvyšku programu. Ak v praxi potrebujeme odhadnúť reálny čas behu takéhoto programu, môžeme si dovoliť časovú zložitosť len asymptoticky odhadnúť, zatiaľ čo komunikačnú zložitosť sa oplatí vypočítať čo najpresnejšie.

Podúloha C

Na pole v pamäti sa môžeme dívať ako na sadu dlaždíc rozmerov $b \times b$. (Rozdelíme si teda aj riadky aj stĺpce poľa do skupín po b.) Máme $m = n/b$ riadkov a tiež m stĺpcov dlaždíc. Do pamäte sa nám v tejto podúlohe zmestí konštantný počet takýchto dlaždíc.
Pre dlaždice si môžeme zaviesť novú súradnicovú sústavu: aj riadky zhora dole, aj stĺpce zľava doprava očíslujeme od 0 po $m - 1$.
Je ľahké nahliadnuť, že keď celé pole otočíme o 90 stupňov doprava, otočíme tým o 90 stupňov doprava každú dlaždicu. Navyše dlaždice zmenia miesta: dlaždica z riadku r a stĺpca s sa presunie do riadku s a stĺpca $m-1-r$.
Pre nepárne m máme v strede poľa jednu dlaždicu, ktorá sa len otočí na mieste. Všetky ostatné dlaždice vieme rozdeliť do štvoríc. V rámci každej štvorice sa prvá dlaždica presunie na miesto druhej, druhá na miesto tretej, tretia na miesto štvrtej a tá zase na miesto prvej.
Naša implementácia bude vyzerať tak, že každú takúto štvoricu dlaždíc spracujeme samostatne, a to nasledovne: Všetky štyri dlaždice načítame do pamäte. V pamäti každú zvlášť otočíme o 90 stupňov doprava. Následne ich všetky štyri zapíšeme naspäť na disk, ale cyklicky zrotované.



Vľavo pole rozdelené na dlaždice, vpravo to isté pole otočené o 90 stupňov doprava.
Farebne je znázornená jedna štvorica dlaždíc, šípky ukazujú, ako si vymenia miesta.

Každý prvok zo vstupu raz načítame, raz otočíme v rámci jeho dlaždice a raz vypíšeme. Každý blok z disku raz prečítame a raz zapíšeme. Preto máme optimálnu aj časovú zložitosť $O(n^2)$, aj komunikačnú zložitosť $O(n^2/b)$. (V programe uvedenom nižšie robíme pre pohodlie zvlášť otáčanie dlaždíc na mieste a zvlášť výmenu štvoríc, takže každý blok načítame aj zapíšeme dvakrát.)

Listing programu (Python)

```
def otoc_doprava(dlazdica):
    # otočíme doprava dlaždicu rozmerov BxB
    return [ [ dlazdica[B-1-s][r] for s in range(B) ] for r in range(B) ]

def nacitaj_dlazdicu(driadok, dstlpec):
    return [ Read( N*driadok + (N//B)*riadok + dstlpec ) for riadok in range(B) ]

def zapis_dlazdicu(driadok, dstlpec, dlazdica):
    for riadok in range(B):
        Write( N*driadok + (N//B)*riadok + dstlpec, dlazdica[riadok] )

# každú dlaždicu zvlášť otočíme doprava na mieste
for driadok in range(N//B):
    for dstlpec in range(N//B):
        dlazdica = nacitaj_dlazdicu(driadok, dstlpec)
        dlazdica = otoc_doprava(dlazdica)
        zapis_dlazdicu(driadok, dstlpec, dlazdica)

# každú štvoricu dlaždíc cyklicky zrotujeme
blokov = N//B
riadkov, stlpcov = blokov // 2, (blokov+1) // 2
for driadok in range(riadkov):
    for dstlpec in range(stlpcov):
        suradnice = [ (driadok,dstlpec), (dstlpec,blokov-1-driadok),
                      (blokov-1-driadok,blokov-1-dstlpec), (blokov-1-dstlpec,driadok) ]
        stvorica = [ nacitaj_dlazdicu(r,s) for r,s in suradnice ]
        for i in range(4): zapis_dlazdicu( *suradnice[(i+1)%4], stvorica[i] )

# pre kontrolu vypíšeme novú maticu po riadkoch
for riadok in range(N):
    for blok in range(N//B): print( * Read( (N//B)*riadok + blok ), end = ' ' )
    print()
```

Na záver ešte podotkneme, že keby sme mali v podúlohe B k dispozícii $O(b^2)$ pamäte, vedeli by sme vyriešiť podúlohu B v rovnakej zložitosti ako podúlohu A. Stačilo by postupom podobným tomu z riešenia podúlohy C maticu na disku *transponovať* (t.j. preklopiť podľa hlavnej diagonály) a následne by stačilo použiť riešenie podúlohy A.

Ak máme k dispozícii len $O(b)$ pamäte, zmestí sa nám do nej dlaždica so stranou dĺžky $d = \sqrt{b}$. Takýchto dlaždíc budeme mať $(n/d)^2 = n^2/b$. Ak by sme pomocou nich robili transpozíciu vstupnej matice, budeme pre každú dlaždicu robiť d čítaní (pre každý jej riadok ten blok, ktorý ho obsahuje) a d zápisov. Dokopy by sme teda spravili $n^2 d/b = n^2/\sqrt{b}$ čítaní a zápisov blokov. Samotná transpozícia v pamäti aj výpis prvkov si vyžiadajú po $O(n^2)$ krokov. Dostali sme teda riešenie s časovou zložitosťou $O(n^2)$ a komunikačnou zložitosťou $O(n^2/\sqrt{b})$.

TRIDSIATY ŠIESTY ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Michal Forišek
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava 2020