70. ročník Matematické olympiády – 2020/2021

Řešení úloh školního kola kategorie P

P-I-1 Přívalový déšť

Zadání této úlohy bylo velmi podobné hledání nejkratší cesty v neohodnoceném grafu, což se dá vyřešit pomocí prohledávání do šířky v lineárním čase (vzhledem k počtu hran grafu). Zde ovšem navíc byly některé vrcholy (křižovatky) zatopené a mohly být použity pouze ke konci nejkratší cesty.

Až dva body šlo získat za řešení hrubou silou. V testech za jeden bod platilo $k = 1$, tehdy stačilo upravit prohledávání do šířky tak, že po nalezení zatopeného vrcholu se podívalo na všechny jeho sousedy, a pokud jedním z nich byl vrchol číslo $n - 1$, nalezli jsme nejkratší cestu.

Další dva body byly možné získat buď za řešení, které předpokládá, že je nejvýše 100 zatopených křižovatek, nebo za řešení, které předpokládá, že $mk \leq 10^6$:

Pokud je málo zatopených křižovatek, je možné z každé z nich spustit prohledávání do šířky a zapamatovat si, jaká nejkratší cesta z ní vede do vrcholu $n - 1$. Poté stačí spustit prohledávání do šířky z vrcholu 0, které nepokračuje ze zatopených vrcholů, a tím najít nejkratší cesty „suchou nohou“ z vrcholu 0 do všech zatopených vrcholů a do vrcholu $n - 1$ (pokud takové existují). Nakonec je třeba tyto dvě vzdálenosti v každém zatopeném vrcholu zkombinovat a spočítat tak odpověď na úlohu. V tomto řešení jsme spustili $z + 1$ prohledávání do šířky, kde z je počet zatopených vrcholů, má tedy časovou složitost nejvýše $\mathcal{O}(zm)$ a paměťovou $\mathcal{O}(n + m)$.

Pokud platí, že $mk \leq 10^6$, můžeme použít trik, kdy z původního (neorientovaného) grafu $G = (V, E)$ vytvoříme nový (orientovaný) graf $G' = (V', E')$ takový, že V' jsou všechny dvojice (v, t), kde $v \in V$ je vrchol G a t je celé číslo z $\{0, 1, \ldots, k\}$. Tyto dvojice interpretujeme tak, že vrchol (v, t) znamená, že je Vašík ve vrcholu v a zbývá mu ještě t kroků, než dostane rýmu (přičemž pokud $t = k$, tak to interpretujeme tak, že ještě nenachladl). V G' vede hrana z (v, t) do (v', t'), právě tehdy když $vv' \in E$ a navíc je splněna jedna z následujících podmínek:

(1) $t = t' = k$ a v není zatopený (toto odpovídá situaci, kdy Vašík do v přišel suchý a suchý z něj i odchází),

(2) $t = k$, $t' = k - 1$ a v je zatopený (toto odpovídá situaci, kdy se Vašík poprvé namočil ve vrcholu v),

(3) $k > t \geq 1$ a $t' = t - 1$ (toto odpovídá situaci, kdy je Vašík už namočený).

Všimněte si, že máme-li $vv' \in E$ a ani jeden z nich není zatopený, pak vedou orientované hrany oběma směry mezi (v, k) a (v', k).

Platí, že $|V'| = (k+1)n$ a $|E'| = 2km$. První nerovnost je zřejmá z konstrukce, druhá plyne z toho, že pro každou hranu $vv' \in E$ a pro každé $1 \leq t \leq k$ vede z (v, t) právě jedna hrana do množiny $\{(v', t') : 0 \leq t' \leq k\}$.

Spusťme nyní prohledávání do šířky z vrcholu $(0, k)$. Z konstrukce G' vyplývá, že nejkratší cesta v G' z $(0, k)$ do (v, t) odpovídá nejkratší cestě v G z 0 do v takové,

že Vašík se na ní namočil přesně $k - t$ vrcholů před v (respektive ještě nenamočil, pokud $t = k$). Vezmeme-li tedy minimum ze vzdáleností z $(0, k)$ do $(n - 1, t)$ pro $t \in \{0, 1, \ldots, k\}$, dostaneme kýžený výsledek.

Časová i paměťová složitost právě popsaného řešení je $\mathcal{O}(km + kn)$, protože musíme zkonstruovat graf G', který má takovou velikost. Taková složitost přitom není pro zisk pěti bodů dostatečná, jelikož zadání slibuje pouze $mk \leq 10^6$, a G tedy může mít například veliké množství izolovaných vrcholů. Tohle se dá ovšem jednoduše vyřešit, když si uvědomíme, že nemusí G' explicitně konstruovat a během prohledávání do šířky z $(0, k)$ generovat pouze ty hrany a vrcholy, které skutečně navštívíme. Tím časová i paměťová složitost klesne na $\mathcal{O}(mk)$.

Pro získání plného počtu bodů si umědomme, že na předchozí řešení lze také nahlížet jako na prohledání G, kde si ale kromě vzdálenosti od počátečního vrcholu musíme navíc pamatovat, jak daleko jsme již ušli od prvního zatopeného vrcholu. Uvažme teď naprosto hypotetickou situaci, že Vašík doma zjistí, že někde cestou ztratil telefon, a tak se ho vydá hledat (vyzbrojen potápěčskou výstrojí a neoprénem, takže ho již prochladnutí netrápí). Pochopitelně se vrací stejnou – nejkratší – cestou, až přijde do školy, kde svůj telefon najde na stole.

Všimněme si nyní toho, že na této opačné cestě platí, že všechny zatopené vrcholy byly ve vzdálenosti maximálně k od vrcholu $n - 1$. A naopak, jakákoliv cesta z vrcholu $n - 1$ do vrcholu 0, která splňuje, že všechny zatopené vrcholy, které navštíví, jsou ve vzdálenosti nejvýše k od vrcholu $n - 1$, splňuje podmínky úlohy. Optimálním řešením je proto spustit prohledávání do šířky z vrcholu $n - 1$ takové, že do zatopených vrcholů vstupujeme pouze, pokud jsou ve vzdálenosti nejvýše k od vrcholu $n - 1$, a vypsat vzdálenost vrcholu 0, kterou tímto postupem najdeme. Časová i paměťová složitost je zřejmě $\mathcal{O}(n + m)$.

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

#define SIZE 1234567

int n, k;
bool zatopeny[SIZE];
vector<int> e[SIZE];

int d[SIZE];

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

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

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

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

        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] < SIZE) cout << d[0] << endl;
        else cout << "-1" << endl;

        return 0;
}
```

P-I-2 Román

Vymyslet řešení, které vyřeší první dvě podúlohy (celkem za 3 body), je jednoduché; jediný zádrhel může být v implementaci. Popis těchto řešení tedy vynecháme.

Šest bodů šlo získat pomocí oblíbeného *dynamického programování*. To je obecná technika spočívající v tom, že řešení celého problému (jak nejlépe rozdělit celou knihu do kapitol?) spočítáme pomocí řešení menších problémů tvaru *jak nejlépe rozdělit prvních i knih do kapitol?*

Konkrétně označme $f(k)$ nejvyšší počet vyvážených kapitol, který můžeme vytvořit z prvních k odstavců Filipova románu. Řešením celého problému je tedy hodnota f pro celou knihu, tj. $f(n)$. Řekněme, že chceme spočítat $f(k)$ z hodnot $f(i)$ pro $i < k$. Potřebujeme vyzkoušet všechny možnosti rozdělení do kapitol, ale výpočet můžeme značně zjednodušit díky tomu, že už známe hodnoty $f(i)$ pro $i < k$.

Napřed se podíváme na případ, kdy je předem dané, že poslední kapitola začíná odstavcem číslo j (a končí odstavcem k). Pokud toto máme pevně dané, je nejvyšší dosažitelný počet vyvážených kapitol roven $f(j-1)+v_{jk}$, kde hodnotu v_{jk} definujeme jako 1, pokud je kapitola s odstavci od j do k (tedy naše poslední kapitola) vyvážená, jinak jako 0. První člen, $f(j-1)$, je podle definice nejvyšší počet vyvážených kapitol vytvořitelných z prvních $j-1$ odstavců, tj. bez naší poslední kapitoly. Hodnota v_{jk} je vlastně počet vyvážených kapitol v poslední kapitole (0 nebo 1) a v součtu dostáváme nejlepší počet vyvážených kapitol pro prvních k odstavců, když poslední kapitola začíná odstavcem j.

A nyní stačí vyzkoušet všechny možnosti délky poslední kapitoly, tj. možné hodnoty pro j. Z těchto kandidátů vezmeme toho s nejvyšší hodnotou $f(j-1) + v_{jk}$. Dostáváme tedy, že $f(k)$ je maximem z hodnot $f(j-1) + v_{jk}$ pro j od 1

do k. Pro úplnost dodejme, že definujeme $f(0) = 0$ (z 0 odstavců žádnou kapitolu nevytvoříme).

Při počítání $f(k)$ zkoušíme k možností, celkově tedy provedeme řádově $n(n+1)/2$ kroků. To nám dává asymptotickou časovou složitost $\mathcal{O}(n^2)$, dostatečně dobrou na 6 bodů.

Optimální řešení

Existuje ale rychlejší a jednodušší řešení než pomocí dynamického programování. Použijeme *hladový algoritmus;* nejprve ho popíšeme a pak ukážeme, proč funguje.

Budeme procházet odstavce od prvního po poslední. U každého odstavce se (nějakým způsobem) podíváme, jestli nejde vytvořit vyvážená kapitola končící tímto odstavcem. Pokud ano, vytvoříme ji. Na odstavce, které jsme viděli, pak zapomeneme, a postup opakujeme se zbylými.

Tedy například pokud je na vstupu pět odstavců s hodnotami `1 1 -1 1 0`, algoritmus bude procházet odstavce dokud nedojde ke třetímu. Tam zjistí, že lze vytvořit vyváženou kapitolu od druhého odstavce po třetí. Přidá tedy následující rozdělení: `1 | 1 -1 | 1 0`. Pak zapomene na první tři odstavce a zbudou mu odstavce `1 0`. Když narazí na odstavec s hodnotou 0, vytvoří z něj vyváženou kapitolu. Celkově tedy dostaneme rozdělení `1 | 1 -1 | 1 | 0` se dvěma vyváženými kapitolami.

Zbývá si rozmyslet, jak zjistit u momentálního odstavce, zda lze vytvořit vyvážená kapitola, která jím končí. Pokud je odstavec neutrální (hodnota 0), stačí do kapitoly zahrnout pouze tento odstavec. Pokud je momentální odstavec veselý a předchozí smutný nebo naopak, můžeme vytvořit kapitolu o dvou odstavcích. A v jiném případě vyváženou kapitolu vytvořit nelze, protože pokud by to šlo, narazili bychom už dříve na jeden z prvních dvou přpadů; tento fakt necháváme čtenáři k rozmyšlení.

Správnost

Zhruba řešeno, hladový algoritmus funguje správně díky tomu, že vytváří vyvážené kapitoly „co nejrychleji“; jakmile je příležitost v prvních k odstavcích najít vyváženou kapitolu, algoritmus ji odhalí. V této úloze se tímto přístupem nedá nic pokazit a nalezené řešení je tak optimální.

Formálně můžeme správnost dokázat matematickou indukcí podle hodnoty optimálního řešení (počtu vyvážených kapitol). Pokud je hodnota optimálního řešení 0, hladový algoritmus zjevně optimální řešení najde, ať dělá, co dělá. Nyní řekněme, že lze z odstavců vytvořit X vyvážených kapitol. Označme s číslo odstavce, kde končí první vyvážená kapitola nalezená hladovým algoritmem. Pro výše uvedený příklad `1 1 -1 1 0` je tedy $s = 3$. Vezměme nějaké optimální řešení a označme analogicky t číslo odstavce, kde končí první vyvážená kapitola tohoto řešení.

Z toho, jak jsme hladový algoritmus zadefinovali, musí platit $s \leq t$. Označme T hodnotu optimálního řešení, pokud zahodíme prvních t odstavců. Musí platit $T = X - 1$ podle definic X a t. Podobně označme S hodnotu optimálního řešení, pokud zahodíme prvních s odstavců. Platí $S \geq T = X - 1$, protože máme k dispozici

stejné odstavce, jako při useknutí prvních t, a možná nějaké navíc. Zároveň ale S nemůže být víc než $X - 1$, protože jinak bychom mohli z původních odstavců vytvořit $S + 1$ vyvážených kapitol; víme totiž, že v prvních s odstavcích lze nalézt vyváženou kapitolu. Měli bychom tedy $S + 1 > X$ vyvážených kapitol, což je ve sporu s definicí X.

Dohromady jsme tedy dostali $S = X - 1$. Slovy to znamená, že po zahození prvních s odstavců (tj. až po konec první vyvážené kapitoly, kterou hladový algoritmus našel) lze ve zbytku odstavců nalézt $X - 1$ vyvážených kapitol. A nyní použijeme indukční předpoklad: už víme, že hladový algoritmus najde optimální řešení, pokud je jeho hodnota nejvýše $X - 1$. A přesně na takové zadání jej nyní spouštíme. Dokázali jsme tedy, že hladový algoritmus funguje i pokud je hodnota optimálního řešení X, čímž jsme indukci dokončili.

Hladový algoritmus lze snadno implementovat v čase $\mathcal{O}(n)$ a získá plný počet bodů.

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

using namespace std;

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

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

    int vysledek = 0;

    for (int i = 0; i < n; i++) {
        if (a[i] == 0) {
            vysledek++;
        } else if (i < n - 1) {
            // Když najdeme pár opačných odstavců, musíme ještě zvýšit i,
            // abychom nezařadili jeden odstavec do dvou kapitol.
            if (a[i] == -1 && a[i + 1] == 1) {
                vysledek++;
                i++;
            } else if (a[i] == 1 && a[i + 1] == -1) {
                vysledek++;
                i++;
            }
        }
    }

    cout << vysledek << endl;
}
```

P-I-3 Veletrh dortů

Cílem úlohy bylo nalézt pro danou posloupnost její nejdelší souvislou podposloupnost se součtem nejvýše k.

Asi nejjednodušší řešení této úlohy zkusí pro každý možný začátek podposloupnosti $\ell, 1 \leq \ell \leq n$, a každý možný konec $r, \ell \leq r \leq n$, spočítat součet $a_\ell + a_{\ell+1} + \ldots + a_r$ a zapamatuje si nejdelší podposloupnost, jejíž součet nepřesáhl k. Počet souvislých podposloupností je $\mathcal{O}(n^2)$ a pro výpočet součtu prvků každé podposloupnosti je potřeba $\mathcal{O}(n)$ operací. Celková složitost takového algoritmu je proto $\mathcal{O}(n^3)$ a algoritmus si vysloužil 2 body.

Předchozí řešení lze vylepšit za pomoci tzv. prefixových součtů. Nejprve si postupně pro každé i spočítáme součet $s_i = a_1 + a_2 + \ldots + a_i$. Všechny tyto součty lze postupně vypočítat v lineárním čase: Začneme s $s_0 = 0$ a dále pokud jsme již spočítali hodnotu s_i pro $i \geq 0$, snadno spočítáme $s_{i+1} = s_i + a_{i+1}$. Nyní můžeme opět vyzkoušet všechny možné začátky $\ell, 1 \leq \ell \leq n$ a konce podposloupnosti $r, \ell \leq r \leq n$. Součet $a_\ell + a_{\ell+1} + \ldots + a_r$ však nyní snadno spočítáme v konstantním čase jako $s_r - s_{\ell-1}$. Časová složitost takového algoritmu je $\mathcal{O}(n + n^2) = \mathcal{O}(n^2)$ a odpovídá zisku 4 bodů.

Předchozí kvadratické řešení můžeme dále vylepšit na řešení s časovou složitostí $\mathcal{O}(n \log n)$ pomocí tzv. binárního vyhledávání. Budeme opět procházet všechny možné začátky podposloupnosti $1 \leq \ell \leq n$. Využijeme toho, že všechna a_i jsou kladná čísla a součet $a_\ell + a_{\ell+1} + \ldots + a_r = s_r - s_{\ell-1}$ tak s rostoucím r pouze poroste. Budeme udržovat dvě hodnoty $r_{\min}$ a $r_{\max}$ takové, že součet $s_{r_{\max}} - s_{\ell-1} > k$ a $s_{r_{\min}} - s_{\ell-1} \leq k$. Pro začátek zvolíme $r_{\min} = \ell - 1$ (to odpovídá prázdné posloupnosti) a $r_{\max} = n$ (je-li pro tuto volbu $s_{r_{\max}} - s_{\ell-1} \leq k$, víme, že nejdelší souvislá podposloupnost začínající v ℓ končí v n a jsme pro dané ℓ hotovi). V každém následujícím kroku zvolíme $r_0 = (r_{\max} - r_{\min})/2$ (dělení zaokrouhlíme dolů) a spočítáme, zda $s_{r_0} - s_{\ell-1} \leq k$. Je-li tomu tak, změníme hodnotu $r_{\min}$ na r_0, zatímco v opačném případě změníme hodnotu $r_{\max}$ na r_0. V obou případech i po změně máme zaručeno, že nejdelší podposloupnost začínající v ℓ se součtem nejvýše k končí v nějakém r, pro které platí $r_{\min} \leq r < r_{\max}$. V každém kroku se rozdíl $r_{\max} - r_{\min}$ zmenší na polovinu, takže po $\mathcal{O}(\log n)$ iteracích dostaneme $r_{\min} + 1 = r_{\max}$ a optimální hodnota r pro začátek podposloupnosti r je rovna $r_{\min}$. Tento algoritmus (binární vyhledávání) opakujeme pro každý začátek $1 \leq \ell \leq n$, a výsledná časová složitost je proto $\mathcal{O}(n \log n)$, což si vysloužil 7 bodů.

Nakonec vysvětlíme optimální řešení s časovou složitostí $\mathcal{O}(n)$. Využijeme toho, že jestliže nejdelší souvislý úsek se součtem nejvýše k začínající v ℓ končí v nějaké pozici r, tak nejdelší souvislý úsek se součtem nejvýše k začínající v $\ell+1$ končí nutně v pozici r nebo vyšší.

Naše řešení si bude udržovat dvě proměnné ℓ a r a součet $s_r - s_{\ell-1}$. Na začátku je $\ell = 1$ a $r = 0$ a v průběhu algoritmu budou obě proměnné pouze růst, každá však může nabývat hodnoty nejvýše n. Význam ℓ a r je, že nejdelší podposloupnost se součtem nejvýše k začínající v ℓ končí v r ($\ell = 1, r = 0$ odpovídá prázdné posloupnosti). Algoritmus postupně zvyšuje proměnnou ℓ a pro daný začátek pod-

posloupnosti ℓ vždy zvyšuje r tak dlouho, dokud platí $r \leq n$ a $s_r - s_{\ell-1} \leq k$. Jinými slovy prodlužujeme podposloupnost začínající v ℓ tak dlouho, dokud je její součet nejvýše k. Jakmile $s_r - s_{\ell-1} > k$, zapamatujeme si délku současné podposloupnosti a zvýšíme ℓ. Tímto způsobem pokračujeme, dokud ℓ nedosáhne hodnoty n. Nakonec vrátíme nejdelší takto nalezenou souvislou podposloupnost. Algoritmus pro každé ℓ najde největší možné r tak, aby součet nalezené podposloupnosti byl stále nejvýše k, takže nakonec vrátí délku nejdelší souvislé podposloupnosti se součtem nejvýše k. Jeho časová složitost je $\mathcal{O}(n)$. To proto, že v každém kroku buď zvýšíme hodnotu ℓ, nebo hodnotu r; obě proměnné jsou však na začátku alespoň nula a nikdy nepřesáhnou hodnotu n.

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

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

  // Nejdelší souvislý úsek se součtem nejvýše k
  // začíná na pozici p a má délku d
  int d = -1, p = -1;

  int left = 0, right = -1, sum = 0;

  // Procházíme začátky všech souvislých úseků
  for (; left < n; ++left) {
    while (right + 1 < n && sum + a[right + 1] <= k) {
      // Inkrementujeme konec úseku right, dokud je součet od left do right nejvýše k
      sum += a[right + 1];
      ++right;
    }
    if (right - left + 1 > d) {
      // Úsek mezi left a right má součet nejvýše k
      d = right - left + 1;
      p = left;
    }
    // left se inkrementuje, pročež přepočítáme součet mezi left a right
    sum -= a[left];
  }

  cout << d << " " << p + 1 << endl;
}
```

P-I-4 Matice na disku

Podúloha a) – výpis po řádcích

Vypsání matice po řádcích je přímočaré. Využijeme toho, jak jsme matici uložili na disk: průchod maticí po řádcích odpovídá průchodu po sobě jdoucími prvky na disku. Stačí tedy použít příklad ze studijního textu. Procházíme celkem N^2 prvků, takže to zvládneme v čase $\mathcal{O}(N^2)$ a komunikační složitosti $\mathcal{O}(N^2/B)$.

Všimněte si, že komunikační složitost je nejlepší možná: jelikož každý blok obsahuje B prvků, přečtením méně než N^2/B bloků bychom nějaké prvky nutně přeskočili.

Podúloha b) – výpis po sloupcích

Rozmysleme si, co se stane, pokud budeme matici přímočaře procházet po sloupcích.

Jelikož délka řádku N je násobkem velikosti bloku B, každý řádek se skládá z N/B celých bloků. Tím pádem všechny prvky v prvním sloupci leží v navzájem různých blocích. Čtení prvního sloupce tedy vyžaduje přečíst N bloků.

Pak čteme druhý sloupec. Ten je uložený ve stejných blocích jako první sloupec. Jenže pokud je paměť mnohem menší než matice, většinu z těchto bloků jsme už museli z paměti odstranit. Proto i druhý sloupec bude vyžadovat přečíst řádově N bloků.

Podívejme se na příklad pro $N = 8$ a $B = 4$:

11 12 13 14	15 16 17 18
21 22 23 24	25 26 27 28
31 32 33 34	35 36 37 38
41 42 43 44	45 46 47 48
51 52 53 54	55 56 57 58
61 62 63 64	65 66 67 68
71 72 73 74	75 76 77 78
81 82 83 84	85 86 87 88

Podobně můžeme argumentovat pro každý další sloupec, takže komunikační složitost našeho algoritmu dosáhne $\mathcal{O}(N^2)$. Jelikož každé čtení bloku trvá $\mathcal{O}(B)$, spotřebujeme celkem $\mathcal{O}(N^2B)$ času. (Původní verze studijního textu neříkala, kolik času trvá práce s diskem. V reakci na dotazy účastníků jsme to doplnili.)

Uvažme ještě případ, kdy je paměť dost velká, aby se do ní vešlo NB prvků. Tehdy by po projití prvního sloupce mohly zůstat všechny potřebné bloky v paměti, což by stačilo i na dalších $B - 1$ sloupců. Komunikační složitost by se tedy snížila na $\mathcal{O}(N^2/B)$ a časová na $\mathcal{O}(N)$.

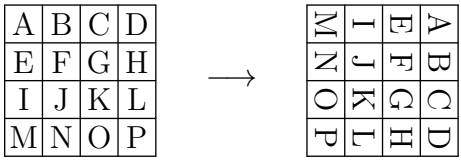
Ještě dodejme, že myšlenky z podúlohy c) se dají použít i na efektivnější řešení této podúlohy. K tomu se vrátíme na konci našeho řešení.

Podúloha c) – otočení po směru hodinových ručiček

Na otočení matice bychom potřebovali číst po řádcích a zapisovat po sloupcích (nebo opačně). Už ale víme, že přístup po řádcích je rychlý a po sloupcích pomalý.

Podmatici velikosti $B \times B$, na disku zarovnanou na bloky, bychom ale otočit uměli. Zadání nám totiž slibuje, ze $M \geq B^2$, takže můžeme podmatici celou načíst z disku do paměti, tam ji otočit a pak zase zapsat zpět na disk. To zvládneme v čase $\mathcal{O}(B^2)$ a komunikační složitosti $\mathcal{O}(B)$.

Celou matici budeme rotovat následovně: Rozkrájíme ji na podmatice $B \times B$, každou podmatici zvlášť otočíme a nakonec podmatice přeházíme. Hezky to je vidět na příkladu:



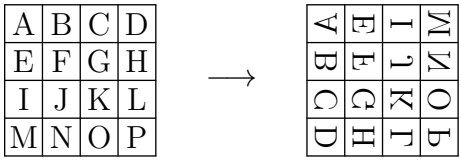
Přeházení podmatic bychom přitom mohli dělat na místě, ale to je zbytečně pracné. Místo toho na disku vyhradíme místo na druhou matici a otočené podmatice budeme zapisovat do ní. Pokud chceme výsledek na místě první matice, pak na závěr algoritmu prostě překopírujeme druhou matici do první.

Jak dlouho to celé potrvá? Stačí vynásobit složitost zpracování jedné podmatice počtem podmatic, což je $(N/B) \times (N/B) = N^2/B^2$. Celkem tedy vyjde časová složitost $\mathcal{O}(N^2/B^2 \cdot B^2) = \mathcal{O}(N^2)$ a komunikační složitost $\mathcal{O}(N^2/B^2 \cdot B) = \mathcal{O}(N^2/B)$. Obojí je nejlepší možné.

Efektivnější řešení podúlohy b)

Vraťme se ještě k podúloze b). Ukážeme, že pokud bychom stejně jako v c) věděli, že $M \geq B^2$, lze také použít myšlenku rozdělení na bloky.

Matici budeme chtít *transponovat* – tím se myslí prohození indexů, tedy prvek z pozice (i, j) skončí na pozici (j, i). Transpozice se dá také provést rozdělením na bloky, transponováním každého bloku zvlášť v paměti a zapsáním v jiném pořadí. Jak na to, opět naznačíme obrázkem:



Časová a komunikační složitost vyjde stejná jako u podúlohy c). Nakonec transponovanou matici přečteme po řádcích, což je ekvivalentní s přečtením původní matice po sloupcích. I úlohu b) tedy umíme vyřešit v čase $\mathcal{O}(N^2)$ a komunikační složitostí $\mathcal{O}(N^2/B)$ za předpokladu, že $M \geq B^2$.

Kdyby tento předpoklad nebyl splněn, mohli bychom místo podmatic $B \times B$ použít podmatice $\sqrt{M} \times \sqrt{M}$. Podmatic této velikosti je $(N/\sqrt{M})^2 = N^2/M$. Každá z nich se vejde do paměti a její přečtení a zapsání stojí $\mathcal{O}(\sqrt{M})$ přístupů na disk, čili $\mathcal{O}(B\sqrt{M})$ času. Celkem tedy vyjde časová složitost $\mathcal{O}(N^2 B/\sqrt{M})$ a komunikační složitost $O(N^2/\sqrt{M})$.