P-II-1 Cesta

Nechť $\vec{s} = s_1, s_2, \ldots, s_N$ je posloupnost specialit dostupných v jednotlivých městech v pořadí dle Filipovy cesty. Nejprve si rozmysleme, jak rozhodnout, zda lze realizovat jeden konkrétní dietní plán $\vec{d} = d_1, d_2, \ldots, d_L$; tedy, zda je $\vec{d}$ *vybranou podposloupností* $\vec{s}$. Pokud se d_1 nevyskytuje v $\vec{s}$, pak dietní plán $\vec{d}$ realizovat nejde. Jinak se král musí někde zastavit na specialitu d_1 a zjevně nic nezkazí tím, že se na ni zastaví v prvním možném městě – řekněme i_1, kde i_1 je první index takový, že $s_{i_1} = d_1$ (kdyby se na ni plánoval zastavit až v nějakém pozdějším městě, pak lze tuto první zastávku přesunout do i_1). Na druhou specialitu d_2 se může zastavit až ve městech za i_1, a opět nic nezkazí tím, že se zastaví v prvním možném. Tuto úvahu opakujeme i pro další města, dokud buď nedojdeme na konec posloupnosti $\vec{d}$ a víme, jak dietní plán realizovat, nebo na konec $\vec{s}$ a dietní plán realizovat nejde. V pseudokódu:

```
int nejbližší_za(int za, int co)
{
  int i = za + 1;
  while (i <= N && s[i] != co)
    i++;
  return i;
}

{
  int za = 0, j;
  for (j = 1; j <= L; j++)
    {
      za = nejbližší_za(za, d[j]);
      if (za > N)
        {
          printf("Nelze realizovat.\n");
          return 0;
        }
    }
  printf("Lze realizovat.\n");
}
```

Tento algoritmus právě jednou projde posloupnosti $\vec{s}$ i $\vec{d}$, jeho časová složitost je tedy $\mathcal{O}(N + L)$. Kdybychom ho postupně aplikovali na všechny dietní plány, dostáváme řešení se složitostí $\mathcal{O}(K(N + L)) = \mathcal{O}(KN + KL)$. Zatímco člen KL je omezený velikostí vstupu, KN může být až kvadratické ve velikosti vstupu.

Povšimněme si, že funkce `nejbližší_za` nezávisí na dietním plánu, pouze na posloupnosti $\vec{s}$. Můžeme si tedy předpočítat nějaké pomocné informace, které zrychlí její vyhodnocování. Jednou z možností je spočítat si hodnoty této funkce pro všechny

možné kombinace `za` a `co` a uložit si je do dvojrozměrného pole `hodnoty`. V algoritmu pak vracíme tyto předpočítané hodnoty v konstantním čase. Složitost rozhodnutí realizovatelnosti jednoho plánu se tím změní na $\mathcal{O}(L)$ a při použití na všechny plány dostáváme složitost $\mathcal{O}(KL)$. Jak spočítáme pole `hodnoty`? Povšimněme si, že pokud `s[za + 1] = co`, pak `hodnoty[za][co] = za + 1`, jinak `hodnoty[za][co] = hodnoty[za + 1][co]`. Pro každou hodnotu `co` můžeme v lineárním čase vyplnit pole `hodnoty` od konce dle těchto vztahů. Možné hodnoty `co` jsou od 1 do M a možné hodnoty `za` jsou od 0 do N. Předpočet nám tak zabere čas $\mathcal{O}(NM)$ a celková časová složitost bude $\mathcal{O}(NM + KL)$. Oproti jednoduchému řešení toto může být rychlejší, ale pouze když počet specialit M je výrazně menší než počet možných dietních plánů K.

Ukážeme si nyní dva způsoby, jak toto řešení vylepšit tak, aby efektivně zvládalo data velikosti uvedené v zadání.

Vzorové řešení 1

Zkusme si předpočítat nějaké menší množství informace, umožňující nám rychle určovat hodnoty funkce `nejbližší_za`. Hodnota `nejbližší_za (za, co)` závisí pouze na tom, na kterých pozicích se `co` vyskytuje v posloupnosti $\vec{s}$. Předpočítejme si pro každou hodnotu `co` seznam `pozice[co]` těchto pozic. Hodnota `nejbližší_za (za, co)` je pak nejmenší prvek v seznamu `pozice[co]` větší než `za`, nebo `N+1` jestliže takový prvek neexistuje. Seznam pozic navíc přirozeně dostaneme setříděný v rostoucím pořadí, nejmenší prvek větší než `za` proto můžeme nalézt v logaritmickém čase půlením intervalů.

Seznamy `pozice[co]` snadno určíme jedním průchodem přes $\vec{s}$, jejich vytvoření nám tedy zabere čas $\mathcal{O}(N)$. Každé volání `nejbližší_za (za, co)` zabere čas $\mathcal{O}(\log N)$, výsledná časová složitost vzorového řešení proto bude $\mathcal{O}(N + KL \log N)$. Celková délka seznamů `pozice` je N a dietní plány můžeme načítat průběžně, paměťová složitost je proto $\mathcal{O}(N)$.

Poznamenejme, že seznamy `pozice` by bylo možné reprezentovat i složitějšími datovými strukturami. S použitím van Emde Boasových stromů tak lze dosáhnout časové složitosti $\mathcal{O}((N + KL) \log \log N)$.

```
#include <cstdio>
#include <unordered_map>
#include <vector>
using namespace std;

static int N, M, K, L;

static unordered_map< int, vector<int> > pozice;

static int
nejblizsi_za(int za, int co)
{
  if (za >= N)
    return N;
  if (pozice.count(co) == 0)
    return N;
```

```
  vector<int> &apozice = pozice[co];
  int spodni = 0, horni = apozice.size() - 1;

  if (apozice[horni] <= za)
    return N;
  if (apozice[spodni] > za)
    return apozice[spodni];

  while (horni > spodni + 1)
    {
      int stred = (spodni + horni) / 2;
      if (apozice[stred] <= za)
        spodni = stred;
      else
        horni = stred;
    }

  return apozice[horni];
}

static bool
lze_realizovat(void)
{
  int i, za = -1, co;

  for (i = 0; i < L; i++)
    {
      scanf("%d", &co);
      za = nejblizsi_za(za, co);
    }
  return za < N;
}

int main (void)
{
  int i, co;

  scanf("%d%d%d%d", &N, &M, &K, &L);

  for (i = 0; i < N; i++)
    {
      scanf("%d", &co);
      pozice[co].push_back(i);
    }

  for (i = 0; i < K; i++)
    if (lze_realizovat())
      printf("ano\n");
    else
      printf("ne\n");

  return 0;
}
```

Vzorové řešení 2

V tomto budeme zpracovávat všechny dietní plány dohromady, nebudeme je tedy testovat jeden po druhém. Pro j-tý dietní plán $\vec{d^j} = d_1^j, d_2^j, \ldots, d_L^j$ a i-té město s_i si jako $r_{j,i}$ označme délku nejdelšího počátečního úseku plánu $\vec{d^j}$, který král může mít

realizován po projetí i-tým městem. Budeme postupně procházet města a v každém z nich si určíme hodnoty $r_{j,i}$ pro všechny dietní plány $j = 1, \ldots, K$. Projíždíme-li i-tým městem, pak lze prodloužit realizovaný počáteční úsek pro ty plány, v nichž po již realizovaném úseku následuje specialita s_i. Když $d^j_{r_{j,i-1}+1} = s_i$, pak tedy máme $r_{j,i} = r_{j,i-1} + 1$, jinak $r_{j,i} = r_{j,i-1}$. Plán j je realizovatelný, právě když $r_{j,N} = L$.

Hodnoty $r_{j,i-1}$ pro aktuální hodnotu i si samozřejmě stačí pamatovat v jednorozměrném poli `r[j]` a při počítání $r_{j,i}$ toto pole přepsat. Dle předchozího odstavce se v něm o 1 zvýší právě ty položky, pro něž $d^j_{r[j]+1} = s_i$. Kdybychom postupovali přímočaře a testovali všechny hodnoty j, dostaneme řešení s časovou složitostí $\mathcal{O}(NK)$. Místo toho si v poli `useky[s]` budeme pro každou specialitu s pamatovat seznam všech hodnot j takových, že $d^j_{r[j]+1} = s$. Pak stačí projít indexy j obsažené v `useky[`s_i`]` a pro ně zvýšit `r[j]` o 1. Tyto indexy pak samozřejmě musíme přesunout do správného seznamu v poli `useky`, odpovídajícímu nové hodnotě $d^j_{r[j]+1}$.

Tento upravený postup v i-tém městě stráví čas úměrný tomu, kolik hodnot v poli `r` se zvýší. Toto pole má K položek a každá z nich je menší nebo rovná L, takže celkem za celou dobu běhu algoritmu můžeme hodnoty v poli r zvýšit nejvýše $O(KL)$-krát. Nesmíme ale zapomenout na čas strávený procházením seznamu měst délky N a inicializací pole `useky` velikosti M, výsledná časová složitost je tedy $\mathcal{O}(KL + N + M)$. Stejná je i paměťová složitost.

Poznamenejme, že tuto složitost lze ještě trochu vylepšit. Počet různých specialit ve vstupu je nejvýše $KL + N$, můžeme je tedy přečíslovat na čísla od 1 do $KL + N$, čímž se zbavíme závislosti na M. Rozmyslete si, že takové přečíslování lze provést v čase úměrném velikosti vstupu (například pomocí radix sortu) a výsledná časová složitost celého řešení je $\mathcal{O}(KL+N)$. Pro přehlednost tento krok ve vzorovém programu neprovádíme.

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

static int N, M, K, L;
static vector<int> mesta;
static vector< vector<int> > diety;
static vector<int> r;
static vector< vector<int> > useky;

int main (void)
{
  int j, i, co;

  scanf("%d%d%d%d", &N, &M, &K, &L);
  for (i = 0; i < N; i++)
    {
      scanf("%d", &co);
      mesta.push_back(co);
    }

  diety.resize(K);
  r.resize(K);
  useky.resize(M + 1);
```

```
  for (j = 0; j < K; j++)
    {
      r[j] = 0;
      for (i = 0; i < L; i++)
        {
          scanf("%d", &co);
          diety[j].push_back(co);
        }
      useky[diety[j][0]].push_back(j);
    }
  for (i = 0; i < N; i++)
    {
      vector<int> zvys = useky[mesta[i]];
      useky[mesta[i]].clear();
      for (vector<int>::iterator zj = zvys.begin(); zj != zvys.end(); ++zj)
        {
          r[*zj]++;
          if (r[*zj] < L)
            useky[diety[*zj][r[*zj]]].push_back(*zj);
        }
    }
  for (j = 0; j < K; j++)
    if (r[j] == L)
      printf("ano\n");
    else
      printf("ne\n");
  return 0;
}
```

P-II-2 Vyvážená strava

Neúspěšné pokusy

Úlohy na stromech se často řeší rekurzivní metodou rozděl a panuj. Kdybychom ji ale chtěli přímočaře aplikovat na tento problém, tak brzy narazíme. Na první pohled to vypadá, že když otrháme co nejméně jablek tak, aby levý podstrom byl vyvážený a také pravý podstrom byl vyvážený, tak nic nezkazíme. Problém je v tom, že bychom nyní potřebovali otrhat ještě některá jablka v tom podstromu, ve kterém nám zbylo více jablek. To ale nemůžeme udělat nějak náhodně, protože musíme zachovat vyváženost celého podstromu. Může se dokonce stát, že přesně tolik jablek vůbec otrhat nepůjde a museli bychom jich otrhat více. Museli bychom tedy střídavě jablka otrhávat z obou podstromů, dokud nebudou mít stejný počet jablek (lze si rozmyslet, že ve skutečnosti se podstrom, ze kterého otrháváme, změní nanejvýš dvakrát). Kdybychom takto rekurzivně postupovali v každém podstromě, dostaneme řešení sice správné, zato ale velmi složité a neefektivní.

O něco efektivnější je následující řešení. Vycházet budeme ze stejné metody rozděl a panuj, jenom si pro každý podstrom spočítáme, jaké všechny počty jablek v něm mohou zbýt tak, aby byl vyvážený. U každého vrcholu stromu budeme mít navíc pole čísel obsahující tyto počty. Sloučení dvou podstromů je potom jasné – podíváme se na tyto dva podstromy a když jsou v nich dvě stejná čísla, tak do

výsledného pole přidáme jejich součet. Výsledek potom bude největší číslo u vrcholu s nejnižší výškou. Toto řešení má časovou složitost $\mathcal{O}(JN)$, kde N je počet vrcholů stromu a J celkový počet jablek. To je neuspokojivé, pokud je počet jablek velký.

Vzorové řešení

Nejprve ukážeme, že počty jablek na vyvážené binární jabloni nemohou být úplně náhodné, ale jsou svázané poměrně silnou podmínkou, a díky tomu velmi snadno navrhneme efektivní algoritmus řešící zadanou úlohu.

Představme si vyváženou binární jabloň, na které roste celkem J jablek. Nejprve se podíváme na nejnižší uzel (výšky 0) a pro něj ze zadání platí, že v levém podstromě i v pravém podstromě roste shodně $J/2$ jablek. Analogicky v každém uzlu se počet jablek, který je v jednotlivých podstromech, vždy dělí dvěma. Z toho přímočaře plyne, že v nejvyšším listu, který leží ve výšce H, roste $J/2^H$ jablek. Speciálně je počet všech jablek ve stromě dělitelný číslem 2^H a bude tedy tvaru $B2^H$ pro nějaké celé číslo B. Na druhou stranu pro každý takový počet můžeme jablka na strom jednoznačně rozmístit tak, že půjdeme od spodu stromu nahoru a v každém uzlu pošleme polovinu jablek doprava, zatímco druhou polovinu doleva. Z toho plyne, že v každém listu výšky h roste $B2^{H-h}$ jablek.

Pro danou jabloň je 2^H konstantní a hledáme tedy pouze správnou hodnotu B. Zřejmě čím větší bude B, tím více jablek bude na vyvážené jabloni a tím méně jich bude potřeba ze zadaného stromu otrhat. Hledáme tedy největší takové B, aby v každém listu výšky h rostlo v zadané jabloni alespoň $B2^{H-h}$ jablek. Uvažujme nějaký list ve výšce h, ve kterém roste a jablek. V tomto listu je na konci $B2^{H-h}$ jablek, a proto $B \leq \frac{a}{2^{H-h}}$. Z každého listu takto získáme nějaké omezení shora na B a na závěr vybereme největší takové B, které všem vyhovuje (= minimum z dolních celých částí všech omezení). Potom už stačí pouze odečíst $B2^H$ od původního počtu jablek a máme hledaný nejmenší počet jablek, který je potřeba otrhat, aby byl vzniklý strom vyvážený.

Implementační detaily a časová složitost

Zjistění hloubky stromu a počtu jablek ve všech listech je obyčejný průchod stromu, který stihneme v lineárním čase vzhledem k počtu vrcholů. Potom je potřeba jabloň podruhé projít a pro každý list si spočítat odhad na hledané B. To zvádneme celočíselným vydělením, jenom je potřeba dát si pozor, abychom zbytečně pomalu nepočítali potřebné mocniny dvou. Časová složitost této druhé fáze je opět lineární vzhledem k N a na závěr provádíme už pouze konstantně mnoho operací, tedy časová složitost celého algoritmu je $\mathcal{O}(N)$. V paměti budeme potřebovat uložit celý strom a při jeho procházení si vystačíme s asymptoticky stejným množstvím paměti. Kromě toho už potřebujeme paměti pouze konstantní množství a celková paměťová složitost algoritmu je tedy také $\mathcal{O}(N)$.

```
#include <stdio.h>
#define MAXN 1000000

// Popis stromu
int levy[MAXN], pravy[MAXN], N;
char typ[MAXN];
```

```
// Obsahuje pouze 0 nebo 1, podle toho, jestli je i-tý vrchol vnitřní
int vnitrni_vrchol[MAXN];

long long jablek[MAXN], celkem_jablek = 0; // nebo jiný větší datovy typ
long long mocniny2[MAXN], B;
int vyska_stromu = 0;

// Poprvé projdeme jabloň pouze abychom zjistili její výšku a počet jablek
void dfs1(int vrchol, int vyska) {
  if (typ[vrchol] == 'L') {
    if (vyska > vyska_stromu) vyska_stromu = vyska;
    celkem_jablek += jablek[vrchol];
  } else {
    dfs1(levy[vrchol], vyska + 1);
    dfs1(pravy[vrchol], vyska + 1);
  }
}

// Podruhé hledáme omezení na B a z nich vybíráme minimum
void dfs2(int vrchol, int vyska) {
  if (typ[vrchol] == 'L') {
    long long omezeni = jablek[vrchol] / mocniny2[vyska_stromu - vyska];
    if (omezeni < B) B = omezeni;
  } else {
    dfs2(levy[vrchol], vyska + 1);
    dfs2(pravy[vrchol], vyska + 1);
  }
}

int main(int argc, char *argv[]) {
  scanf("%d\n", &N);
  if (N == 1) { printf("0\n"); return 0; } // Ošetříme speciální případ

  // Načítání vstupu
  for (int i = 0; i < N; i++) {
    scanf("%c", typ + i);
    if (typ[i] == 'U') {
      scanf("%d%d\n", levy + i, pravy + i);
      // Indexujeme od 0
      vnitrni_vrchol[--levy[i]] = vnitrni_vrchol[--pravy[i]] = 1;
    } else scanf("%lld\n", jablek + i);
  }

  int nejnizsi; // V korektně zadaném vstupu existuje jediný takový vrchol
  for (nejnizsi = 0; typ[nejnizsi] == 'L' || vnitrni_vrchol[nejnizsi]; nejnizsi++);

  dfs1(nejnizsi, 0); // Zjistíme výšku stromu a počet jablek ve stromě

  // Předpočítáme si mocniny 2, abychom je potom měli v konstantním case
  mocniny2[0] = 1;
  for (int i = 0; i < vyska_stromu; i++) mocniny2[i + 1] = 2 * mocniny2[i];

  B = celkem_jablek; // Více to být rozhodně nemůže
  dfs2(nejnizsi, 0); // Hledáme hodnotu B

  printf("%lld\n", celkem_jablek - B * mocniny2[vyska_stromu]);
  return 0;
}
```

P-II-3 Zaokrouhlování

Nejprve si uvědomme, že stačí uvažovat případ, kdy čísla v matici jsou kladná a menší než 1 – kdyby se v matici vyskytovalo například 5,31, můžeme ho nahradit číslem 0,31. Součty řádků i sloupců zůstanou celočíselné a řešení pro novou matici bude stejné jako pro původní.

Zaokrouhlením kladného čísla menšího než 1 dostaneme nulu nebo jedničku. Úlohu proto lze ekvivalentně zformulovat takto: Mějme matici o rozměrech $M \times N$ a pro každý její sloupec a řádek zadané přirozené číslo. Naplňte matici nulami a jedničkami tak, aby počet jedniček v jednotlivých řádcích/sloupcích odpovídal zadaným číslům.

Základní idea řešení takto upravené úlohy je jednoduchá. Budeme postupovat řádek po řádku a v každém řádku budeme postupovat hladově. Vždy víme kolik potřebujeme v řádku umístit jedniček a z možných sloupců vybereme ty, kterým zbývá doplnit nejvíce jedniček. Tím intuitivně zabráníme tomu, abychom se dostali do situace, kdy zbývá doplnit do nějakého sloupce více jedniček než je počet zbývajících nezpracovaných řádků.

Přímá implementace řešení může vypadat následovně. Sloupce si setřídíme podle toho, kolik jedniček je v nich potřeba. Naplníme první řádek hladově, zaktualizujeme počty zbývajících jedniček ve sloupcích. Sloupce znovu setřídíme a pokračujeme dokud nevyřešíme všechny řádky. Výpočet se tedy v každém řádku skládá z třídení N hodnot, což zvládneme v čase $\mathcal{O}(N \log N)$. Protože řádků je M, je celková složitost tohoto přístupu je $\mathcal{O}(MN \log N)$. Problémem je třídení v každém kroku. Pokusme se ho udělat rychleji.

Nejprve si všimněme, že počty jedniček ve sloupcích jsou z roszahu $0 \ldots M$ (v opačném případě úloha zjevně nemá řešení). Můžeme tedy použít bucketsort, čímž se nám složitost změní na $\mathcal{O}(M(M+N))$. Protože si na začátku algoritmu můžeme matici transponovat, lze předpokládat, že $M \leq N$, a proto máme $\mathcal{O}(M(M+N)) = \mathcal{O}(MN)$.

Jinou možností (tu používáme ve vzorovém řešení) je si všimnout, že ve sloupcích, které jsme použili, se zmenší počty zbývajících jedniček právě o 1. A protože je bereme v setříděném pořadí, zůstanou setříděné i když je zmenšíme o 1. Získáme tedy 2 posloupnosti sloupců – první jsou ty, které jsme změnili, a ty máme setříděné. Druhé jsou ty co jsme nezměnili, ale ty jsou také setříděné. Stačí je tedy slít v celkovém čase $\mathcal{O}(N)$, čímž také získáme řešení běžící v optimálním čase $\mathcal{O}(MN)$.

Jak je to s pamětí? Zdá se, že řešení nemůžeme vypisovat rovnou, protože nevíme dopředu zda úloha bude mít řešení. Máme tedy dvě možnosti. Buď využijeme faktu, že libovolnou matici, kterou můžeme dostat na vstupu, lze vždy zaokrouhlit (což je možné dokázat a čehož jsme pro větší názornost využili v našem řešení), nebo v prvním průchodu pouze zjístíme zda řešení existuje a vypisovat výsledek budeme při druhém průchodu. Díky tomu si výsledek nemusíme ukládat do paměti, ale můžeme ho rovnou vypsat, a tak nám stačí pouze paměť $\mathcal{O}(N+M)$.

Zbývá dokázat, že popsaný algoritmus funguje. Pokud algoritmus najde řešení, tak je zjevně správné. Zbývá tedy dokázat, že pokud existuje alespoň jedno řešení

úlohy, tak nějaké řešení nalezne i náš algoritmus. Nechť tedy existuje řešení $\check{R}$, ale náš algoritmus žádné nenajde. Nechť r je číslo prvního řádku, na kterém se náš algoritmus během výpočtu odchýlí od $\check{R}$. Je snadné si uvědomit, že takový řádek se bude lišit od $\check{R}$ v sudém počtu pozic. Označme si i a j čísla takových sloupců, že $\check{R}$ má na r-tém řádku v i-tém sloupci 1 a j-tém 0, zatímco náš algoritmus je přiřadí naopak. To ale znamená, že v r-tém řádku zbývalo v j-tém sloupci alespoň tolik jedniček co v i-tém (plyne z toho, že náš algoritmus volí sloupce hladově podle zbývajícího počtu jedniček). Po r-tém řádku jich tedy musí zbývat v i-tém sloupci méně a tedy v $\check{R}$ musí existovat řádek r' takový, že $r' > r$ a v i-té pozici má 0 a j-té má 1. Zjevně pokud tyto čtyři pozice v $\check{R}$ zinvertujeme získáme opět korektní řešení $\check{R}'$. To se ale na řádku r liší od našeho v méně políčcích, takže můžeme místo $\check{R}$ použít $\check{R}'$ a pokračovat analogicky pro všechny odchylky a všechny řádky. Pokud tedy existovalo $\check{R}$, tak algoritmus musel dojít k řešení, které bylo z $\check{R}$ odvoditelné úpravami, které zachovávají celkový počet jedniček v jednotlivých řádcích/sloupcích, což je ve sporu s tím, že žádné řešení nenašel.

```
#include <cstdio>
#include <vector>
#include <algorithm>

struct SPopisSloupce {
        int sloupec;
        int zbyvaJednicek;
};

bool operator<(const SPopisSloupce &a, const SPopisSloupce &b)
{
        return a.zbyvaJednicek > b.zbyvaJednicek;
}

int main()
{
        int N, M;
        scanf("%d %d", &M, &N);

        std::vector<int> souctyRadku(M), souctySloupcu(N);
        for (int i=0; i<M; i++) {
                for (int j=0; j<N; j++) {
                        int cela, desetinna;
                        scanf("%d.%d", &cela, &desetinna);
                        // Toto si můžeme dovolit, protože jsou čísla
                        // zadaná s přesností na dvě desetinná místa
                        souctyRadku[i] += desetinna;
                        souctySloupcu[j] += desetinna;
                }
        }

        std::vector<int> jednicekVRadku(M), jednicekVeSloupci(N);

        // Kolik jedniček je třeba umístit do jednotlivých řádků
        for (int i=0; i<M; i++)
                jednicekVRadku[i] = souctyRadku[i]/100;

        // Totéž pro sloupce, ale rovnou provádíme první fázi třídění
        std::vector<std::vector<int> > prihradkySeSloupci(M+1);
```

```
        for (int j=0; j<N; j++) {
                jednicekVeSloupci[j] = souctySloupcu[j]/100;
                prihradkySeSloupci[jednicekVeSloupci[j]].push_back(j);
        }

        // Nyní jenom projdeme přihrádky a dostaneme setříděné sloupce
        std::vector<SPopisSloupce> serazeneSloupce;
        for (int i=M; i>=0; i--) {
                for (int j=0; j<prihradkySeSloupci[i].size(); j++) {
                        SPopisSloupce sloupec = { prihradkySeSloupci[i][j], i };
                        serazeneSloupce.push_back(sloupec);
                }
        }

        // Zpracováváme řádek po řádku
        for (int i=0; i<M; i++) {
                std::vector<int> vysledek(N);

                // Jedničky dáváme k maximálním sloupcům
                int sloupec = 0;
                while(jednicekVRadku[i]-- > 0) {
                        serazeneSloupce[sloupec].zbyvaJednicek--;
                        vysledek[serazeneSloupce[sloupec].sloupec] = true;
                        ++sloupec;
                }

                // Znovu "setřídíme" sloupce, tj. slijeme setříděná pole sloupců
                std::vector<SPopisSloupce> znovuSerazeneSloupce(N);
                std::merge(
                        serazeneSloupce.begin(), serazeneSloupce.begin()+sloupec,
                        serazeneSloupce.begin()+sloupec, serazeneSloupce.end(),
                        znovuSerazeneSloupce.begin()
                );
                std::swap(serazeneSloupce, znovuSerazeneSloupce);

                // Jednička odpovídá 'N' a nula odpovídá 'D'
                for (int i=0; i<N; i++)
                        printf("%c", (vysledek[i]==1) ? 'N' : 'D');
                printf("\n");
        }

        return 0;
}
```

P-II-4 Log-space programy

a) K řešení této úlohy stačí použít „školní“ algoritmus pro násobení čísel. Postupně tedy od konce počítáme číslice výsledku a udržujeme si přenos do vyšších řádů.

```
var n : integer;                                { vstup: velikost čísel }
    A, B : array [1..n] of integer;             { vstup: násobená čísla }
    C : array [1..2*n] of integer;              { výstup: výsledek násobení }
    rad, i, j, soucet, prenos: integer;
begin
  prenos := 0;
  for rad := 1 to 2 * n do
```

```
    begin
      soucet := prenos;
      for i := 1 to rad do
        begin
          j := rad + 1 - i;
          if (i <= n) and (j <= n) then
            soucet := soucet + A[i] * B[j];
        end;
      C[rad] := soucet mod 10;
      prenos := soucet div 10;
    end;
end.
```

Musíme si rozmyslet, že proměnné `prenos` a `soucet` nabývají pouze hodnot polynomiálně velkých v n. Povšimněme si, že `prenos` je vždy nanejvýš roven hodnotě `soucet` z předchozí iterace a že hodnota proměnné `soucet` se v každé iteraci vnějšího cyklu zvýší o nejvýše $81n$. Hodnoty v proměnných `prenos` a `soucet` jsou tedy vždy nejvýše $162n^2$ (přesnější rozbor ukazuje, že jsou dokonce vždy menší než $90n$).

b) K řešení úlohy použijeme známý postup na hledání cesty v bludišti – na každé křižovatce zatoč vpravo. Snadno si rozmyslíme, že v bludišti bez cyklů takto postupně projdeme celou komponentu bludiště dostupnou z počátečního bodu (každou chodbu v obou směrech) a vrátíme se na místo, kde jsme začali. Výhodou tohoto postupu je, že si nemusíme pamatovat žádné informace kromě toho, kterou chodbou a ve kterém směru jsme šli naposledy, takže ho lze přímočaře převést na log-space program.

Začneme tedy na libovolné hraně vycházející z vrcholu u a budeme obcházet graf popsaným postupem, dokud buď nedorazíme do v (pak u a v jsou ve stejné komponentě souvislosti) nebo dokud se nevrátíme na počáteční hranu ve stejném směru (pak u a v jsou v různých komponentách). Abychom mohli implementovat příkaz „zahni doprava“, potřebujeme mít nějaké pořadí na hranách okolo vrcholu – můžeme použít například jejich pořadí ve vstupním poli.

```
var n, m : integer;                  { vstup: počet vrcholů a hran }
    A, B : array [1..m] of integer;  { vstup: hrany }
    u, v : integer;                  { vstup: počáteční a cílový vrchol }
    s : integer;                     { výstup: zda u a v jsou ve stejné komponentě }

function nasledujici(z, p : integer) : integer;
var i : integer;
begin
  i := p;
  repeat
    i := i mod m + 1;
  until (A[i] = z) or (B[i] = z);
  nasledujici := i;
end;

function vpravo(z, k : integer) : integer;
{ Vrátí číslo x takové, že hrana (z,x) následuje po (z,k) v pořadí kolem vrcholu z }
var p : integer;
begin
```

```
  p := 1;
  repeat
    p := nasledujici(z, p);
  until (A[p] = k) or (B[p] = k);
  p := nasledujici(z, p);
  if A[p] = z then
    vpravo := B[p]
  else
    vpravo := A[p];
end;

var prvni_z, prvni_k, akt_z, akt_k, p: integer;
    h : boolean;

begin
  if u = v then
    begin
      s := 1;
      exit;
    end;

  { Nejprve ověřme, zda z u vede alespoň jedna hrana,
    jinak u a v nejsou ve stejné komponentě. }
  h := false;
  for p := 1 to m do
    if (A[p] = u) or (B[p] = u) then
      h := true;
  if not h then
    begin
      s := 0;
      exit;
    end;

  { Najdeme počáteční hranu z u. }
  p := nasledujici(u, 1);
  prvni_z := u;
  if A[p] = u then
    prvni_k := B[p]
  else
    prvni_k := A[p];

  { A obcházíme strom. }
  akt_z = prvni_z;
  akt_k = prvni_k;

  repeat
    if akt_z = v then
      begin
        s := 1;
        exit;
      end;
    p := vpravo(akt_k, akt_z);
    akt_z := akt_k;
    akt_k := p;
  until (prvni_z = akt_z) and (prvni_k = akt_k);

  s := 0;
end.
```