P-III-4 Vašek a houba

V zadání máme slíbeno, že žádné a_i není větší než $L = 10^9$. To znamená, že ani bitový AND několika čísel ze vstupu není větší než L (vždy platí $a \,\&\, b \leq a, b$). Nechť $1 \leq x \leq L$ a d je celé číslo takové, že $2^d \mid x$ (tak značíme, že číslo 2^d je dělitelem čísla x). Potom jistě i $2^d \leq x \leq L$. Když vezmeme dvojkový logaritmus obou stran nerovnosti, dostaneme $d \leq \log_2 L < 30$ (poslední nerovnost můžeme nahlédnout i bez kalkulačky, například pokud víme, že $2^{10} = 1\,024$, tedy $2^{10} > 10^3$. Umocněním obou stran nerovnosti na třetí dostaneme $2^{30} > 10^9$).

To znamená, že pokud $2^d \mid x$, pak $d \in \{0, 1, \ldots, 29\}$. Takže můžeme vyzkoušet každé takové d a zeptat se, zda existují taková čísla ze vstupu, že největší mocnina dvojky, která dělí jejich bitový AND, je právě 2^d. Potom stačí vzít největší takové d, kde odpověď byla kladná, a k němu i příslušná čísla ze vstupu.

Jak zjistit pro zadané d, zda existuje taková podmnožina $\{b_j\}_{j=1}^{k} \subseteq \{a_i\}_{i=1}^{n}$ čísel ze vstupu, že označíme-li $A = b_1 \,\&\, b_2 \,\&\, \ldots \,\&\, b_k$, tak $2^d \mid A$ a zároveň $2^{d+1} \nmid A$? To, že $2^d \mid A$ znamená, že $A = 2^d y$ pro nějaké přirozené y. Tedy posledních d pozic (tj. pozice $2^0, 2^1, \ldots, 2^{d-1}$) v binárním zápisu A jsou nuly. A naopak $2^{d+1} \nmid A$ znamená, že na pozici 2^d musí být jednička.

To znamená, že každé b_j musí mít v binárním zápisu na pozici 2^d jedničku, jinak by ji tam totiž nemělo ani A. To navádí na jednoduchý hladový algoritmus: Vezmeme všechna a_i taková, že na pozici 2^d mají jedničku, a podíváme se, jestli je jejich bitový AND dělitelný 2^d. Pokud ano, tak jistě není dělitelný 2^{d+1}, protože víme, že na pozici 2^d je jednička. Pokud není dělitelný 2^d, tak to znamená, že na nějaké pozici $2^\ell < 2^d$ je jednička. A tehdy můžeme prohlásit, že neexistuje taková podmnožina čísel ze vstupu, že by jejich bitový AND byl dělitelný 2^d a už nebyl dělitelný 2^{d+1}: Aby byl dělitelný 2^d, tak bychom potřebovali, aby jedno z ANDovaných čísel mělo na pozici 2^d jedničku a na pozici 2^ℓ nulu. Nicméně, my jsme vzali všechna čísla, která mají jedničku na pozici 2^d, a ukázalo se, že všechna mají zároveň jedničku na pozici 2^ℓ.

Náš algoritmus tedy pro každé d od 0 do 29 projde všechna čísla na vstupu, vybere ta, která mají na pozici 2^d jedničku, vezme jejich AND a podívá se, jestli je dělitelný 2^d. Potom ze všech d, pro něž odpověď byla kladná, vybere to největší a vypíše všechna čísla ze vstupu, která mají na příslušné pozici jedničku. Vždycky existuje alespoň jedno d, pro nějž je odpověď kladná (a to nejmenší takové, že existuje číslo na vstupu, které má na dané pozici jedničku), takže náš algoritmus vždy vrátí nějakou odpověď, o níž jsme si navíc rozmysleli, že je správná.

Časová složitost je $\mathcal{O}(n \log L)$.

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

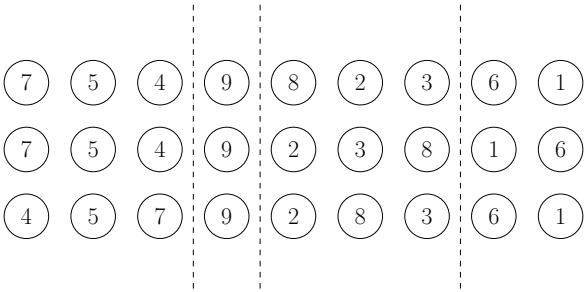
```
int get_best_power(const vector<int> &input) {
    // 2^29 < 10^9 < 2^30, tedy MAX_PW je nejvyšší mocnina dvojky
    // ve vstupních číslech.
    int MAX_PW = 1<<29;

    // Zkoušíme postupně mocniny dvojky od největší možné
    for (int d = MAX_PW; d >= 1; d/=2) {

        // res = (1111....1)_2, tj. pro všechna čísla x <= 10^9 plati x&res = x
        int res = (1<<30) - 1;

        for (auto x : input) {
            // Pokud číslo ze vstupu má na pozici d jedničku,
            // tak ho vyanduj s průběžným výsledkem.
            if (x & d) res &= x;
        }
        if (0 == (res % d)) // Pokud je průběžný výsledek dělitelný d
            return d; // Tak víme, ze d je největší takové a můžeme ho vrátit.
    }
}

int main() {
    vector<int> input;
    int n;

    cin >> n;
    input.resize(n);

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

    int d = get_best_power(input);

    int cnt = 0;
    for (auto x : input)
        if (x & d)
            cnt++;

    cout << cnt << endl;

    for (auto x : input)
        if (x & d)
            cout << x << " ";
    cout << endl;
}
```

P-III-5 Poslední dostih sezóny

Všimněme si, že pokud by se stalo, že nějakých M koní v každém závodě skončilo na posledních M pozicích, tak by si už každý ze zbylých $N - M$ koní na tuto M-tici *věřil* a naopak žádný kůň z této M-tice by si nevěřil na žádného ze zbylých $N - M$ koní. Myšlenkou správného řešení je, že tato podmínka nám již umožní dostatečně charakterizovat, kdo si na koho věří.

Speciálně směřujeme k algoritmu, který koně rozdělí do skupinek jen podle této podmínky (viz obrázek), jinak řečeno najde všechna možná M taková, že pro každého koně platí, že se buď ve všech závodech umístil na pozici M nebo horší, nebo se ve všech závodech umístil na pozici ostře lepší než M. Ukážeme, že pak si

každý kůň bude věřit právě na protivníky ze své skupinky a ze skupinek, které se umístily hůře.



Na začátek si uvědomme, že když pro tři různé koně a, b, c platí, že kůň a si věří na koně b a ten si zase věří na koně c, pak si i a věří na c. Jestliže totiž existuje zároveň posloupnost koní $x_1, x_2, \ldots, x_m$ taková, že a někdy porazil x_1, x_1 někdy porazil x_2, …, x_m někdy porazil b, a posloupnost koní $y_1, y_2, \ldots, y_n$ taková, že b někdy porazil y_1, y_1 někdy porazil y_2, …, y_n někdy porazil c, spojením těchto posloupností dostaneme posloupnost ukazující, že si i a věří na c. Tato užitečná vlastnost se nazývá *tranzitivita.*

Nyní uvažme koně, který v prvním závodě doběhl poslední a všichni si na něj tedy věří. Označme jej k_0 a dále označme $k_1, \ldots, k_\ell$ koně, na které si k_0 věří. Písmenem P budeme značit množinu těchto $\ell + 1$ koní. Dokážeme o nich, že se v každém závodě umístili na posledních $\ell + 1$ příčkách.

Každý si věří na koně k_0 a ten si zase z definice věří na libovolného koně k_i, $1 \leq i \leq \ell$. Z tranzitivity tedy plyne, že na každého koně z P si věří úplně všichni jeho soupeři, speciálně tedy i zbylí koně z P. Na druhou stranu si žádný kůň z P nemůže věřit na nikoho jiného než na soupeře z P, protože pak by si na tohoto koně díky tranzitivitě věřil i k_0. To ale znamená, že žádný kůň z P nikdy neporazil koně, který v P není, jinými slovy všichni koně z P se pokaždé umístili na posledních $\ell+1$ příčkách. Ještě si všimněme, že neexistuje množina koní P' menší než P taková, že by koně z P' v každém závodě skončili na posledních $|P'|$ pozicích. To by totiž podle úvahy z úvodu znamenalo, že kůň k_0 si věří na maximálně $|P'| - 1 < \ell$ protivníků.

Teď už je ale jednoduché kýženou množinu P najít. Budeme procházet všechny závody najednou od posledního N-tého místa až k prvnímu a přitom si postupně pro každé místo i spočítáme, kolik je koní takových, že v každém závodě skončili na i-tém místě nebo hůře. To můžeme implementovat tak, že si pro každého koně budeme pamatovat, v kolika závodech jsme jej již potkali – jakmile bude hodnota odpovídajícího čítače K, nalezli jsme jeho nejlepší možné umístění. Ve chvíli, kdy se stane, že pro i-té pořadí je takových koní $N + 1 - i$, jsme nalezli množinu P a víme, že každý kůň z ní si věří na $|P| - 1$ protivníků.

Nyní už jsme skoro v cíli. Předchozí algoritmus totiž můžeme iterovat, neboli po tom, co najdeme skupinku koní P, která ve všech závodech obsadila posledních $|P|$ příček, na tyto koně můžeme zapomenout a hledat obdobnou skupinku pro zbylé

koně. Musíme si ovšem pamatovat, že k výsledku pro všechny následující skupinky musíme přičíst $|P|$, neboť na nalezenou množinu P si věří všichni ostatní. Tím jsme se dostali k jednoduchému algoritmu, jejž jsme slibovali na začátku řešení. Rozdělujeme postupně koně do skupinek tak, že každý kůň si věří právě na všechny protivníky ze své skupinky a ze skupinek, které se umístily hůře. Časová složitost popsaného algoritmu je $\mathcal{O}(NK)$ a stačí na deset bodů.

Jiné řešení

Úlohu také bylo možno řešit s pomocí teorie grafů. Sestavíme orientovaný graf na N vrcholech, kde každý vrchol odpovídá jednomu koni. Dále mezi vrcholy u a v vytvoříme hranu, pokud v nějakém dostihu kůň u porazil koně v. V každém dostihu máme řádově N^2 takových párů koní, takže výsledný graf bude mít řádově KN^2 hran. Nyní si jen uvědomíme, že kůň u si věří na svého protivníka v právě tehdy, když existuje cesta z odpovídajícího vrcholu u do vrcholu v. Stačí tedy takto vytvořený graf projít z každého vrcholu a při tom zaznamenávat, kolik vrcholů jsme při každém prohledávání navštívili.

Takový algoritmus pracuje v čase $\mathcal{O}(N^3K)$. Můžeme jej snadno vylepšit tak, že do grafu přidáme méně hran. Speciálně stačí do vytvářeného grafu přidat hranu jen v případě, kdy kůň číslo u v nějakém závodě doběhl na pozici těsně před koněm v, tedy v případě, kdy na vstupu těmto koním odpovídají dvě po sobě jdoucí čísla. Potom za každý závod přidáme pouze $N-1$ hran a velikost grafu tak bude řádově KN. Stejný algoritmus pak poběží v čase $\mathcal{O}(N^2K)$ a může získat až pět bodů.

Neefektivita předchozího algoritmu spočívá v tom, že zvlášť počítá odpověď pro každého koně. Pomůžeme si, pokud budeme koně slučovat do skupinek stejně jako v předchozím řešení. Všimněme si, že jestliže máme dva koně a a b takové, že a si věří na b a b si věří na a, pro všechny zbylé koně díky tranzitivitě (viz výše) platí, že si věří na a, právě když si věří na b, a naopak a si na nějakého takového koně věří právě tehdy, když si na něj věří b.

Z toho plyne, že jestliže v našem grafu nalezneme cyklus, víme, že hledaná odpověď je stejná pro všechny koně odpovídající vrcholům tohoto cyklu, a můžeme jej zkontrahovat do jediného vrcholu. Tím myslíme, že celý cyklus nahradíme jediným vrcholem v, který bude odpovídat všem původním vrcholům cyklu. Dále všechny hrany, které vedly do nějakého vrcholu cyklu, přepojíme tak, aby vedly do v, a obdobně se vypořádáme s hranami, které z cyklu vycházely. Nakonec smažeme vzniklé smyčky – hrany vedoucí z vrcholu v do sebe sama. Nesmíme zapomenout si pro každého koně pamatovat, který vrchol v novém grafu mu odpovídá. Opakovanou kontrakcí nakonec dostaneme graf bez cyklů, jehož vrcholy odpovídají takzvaným komponentám silné souvislosti původního grafu a také jsou to přesně skupinky koní z předchozího řešení.

Navíc platí, že se pro žádnou dvojici vrcholů u, v nestane, že by nevedla cesta ani z u do v, ani z v do u – to proto, že se pro žádnou dvojici odpovídajících koní a, b nemůže stát, že by si nevěřil ani a na b, ani b na a. Z této vlastnosti plyne, že se vrcholy tohoto grafu dají uspořádat do řady tak, aby všechny hrany vedly zleva doprava, což odpovídá tomu, že stejným způsobem byly v předchozím řešení seřazeny

skupinky koní. Nakonec takto seřazené vrcholy, opět obdobně jako v předchozím řešení, můžeme postupně procházet a počítat, na kolik soupeřů si každý kůň věří. Zbývá tedy pouze navrhnout algoritmus, který pro zadaný orientovaný graf najde jeho komponenty silné souvislosti. To už je ale standardní problém teorie grafů, který se dá řešit v lineárním čase například pomocí Tarjanova algoritmu. Nakonec jsme se tedy opět dostali k algoritmu s časovou složitostí $\mathcal{O}(NK)$.

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

vector< vector<int> > zavody;
vector<int> nalezenych, vysl;
int N, K;

int main() {
    std::ios::sync_with_stdio(false);
    cin >> N >> K;

    zavody.resize(K, vector<int>(N));
    nalezenych.resize(N+1, 0);
    vysl.resize(N+1, 0);

    for (int i=0; i<K; ++i)
        for (int j=0; j<N; ++j)
            cin >> zavody[i][j];

    int hotovych = 0;
    int poslnalez = N-1;

    for (int i=N-1; i>=0; --i) {
        // Procházíme všechny závody najednou od posledního místa k prvnímu.
        for (int j=0; j<K; ++j) {
            // Koním, kteří se někdy umístili na i-té pozici,
            // zvýšíme odpovídající čítač o jedna.
            if (++nalezenych[zavody[j][i]] == K)
                ++hotovych;
        }
        if (i+hotovych == N) {
            // Jestliže jsme narazili na konec aktuální skupinky,
            // pro všechny koně v ní uložíme výsledek.
            while (poslnalez >= i) {
                vysl[zavody[0][poslnalez]]=hotovych-1;
                --poslnalez;
            }
        }
    }

    for(int i=1; i<=N; ++i)
        cout << vysl[i] << " \n"[i==N];
}
```

P-III-6 Vykopávky

Nejprve nalezneme a očíslujeme místnosti na mapě: postupně procházíme čtverce mapy a když narazíme na prázdný a ještě neoznačený, pak si pro všechny z něj dostupné čtverce označíme, že patří do nově nalezené místnosti (například prohle-

dáním do hloubky). Pro každý čtverec si tedy budeme pamatovat číslo místnosti, do které patří.

Pro libovolné r a c uvažme obdélník o rozměrech $(M+2)\times(N+2)$ s protilehlými rohy tvořenými čtverci $(r-1, c-1)$ a $(r+M, c+N)$. Jako $X(r,c)$ si označme útvar, který z tohoto obdélníka vznikne odebráním jeho rohových čtverců $(r-1, c-1)$, $(r+M, c-1)$, $(r+M, c+N)$ a $(r-1, c+N)$. Když profesor Jones vykope díru o rozměrech $M \times N$ s protilehlými rohy tvořenými čtverci (r,c) a $(r+M-1, c+N-1)$, pak může prozkoumat právě všechny místnosti, které obsahují alespoň jeden čtverec z „osekaného obdélníka" $X(r,c)$. Mohli bychom si ted jednoduše hrubou silou spočítat pro všechna $1 \le r \le R-M+1$ a $1 \le c \le C-N+1$ počet místností, které protínají útvar $X(r,c)$, a vzít maximum; to by nám zabralo čas $\mathcal{O}(RCMN)$.

Zkusme tento postup vylepšit: osekané obdélníky $X(r,c)$ a $X(r,c+1)$ se od sebe liší pouze v $2M+2$ čtvercích (na svislých hranách obdélníka plus u osekaných rohů). Můžeme tedy zkusit informaci o místnostech protnutých $X(r,c)$ zkusit přepočítat na informaci o místnostech protnutých $X(r,c+1)$. K tomu si stačí pro každou místnost pamatovat, kolik z jejích čtverců je v aktuálně uvažovaném útvaru. Přidáme-li do útvaru jeden nový čtverec, stačí si zvýšit počítadlo u místnosti obsahující tento čtverec (není-li daný čtverec plný) a naopak. Obdobně si udržujeme počet protnutých místností – zvedne-li se počítadlo pro nějakou místnost z 0 na 1, zvýšíme si počet protnutých místností, a klesne-li na 0, zmenšíme si ho. Takto zvládneme informaci pro $X(r,c)$ přepočítat na informaci pro $X(r,c+1)$ v čase $\mathcal{O}(M)$.

Jelikož zkoušíme nejvýše RC pozic kopané díry, výsledná časová složitost je $\mathcal{O}(RCM)$, což je v nejhorším případě (když M je řádově stejně velké jako R) $\mathcal{O}(R^2C)$. Tento odhad ještě můžeme drobně vylepšit, jestliže si v případě že $N < M$ nejprve celou mapu transponujeme tak, abychom z řádků udělali sloupce a naopak; to nám dá časovou složitost $\mathcal{O}(RC\min(M,N))$, resp. $\mathcal{O}(RC\min(R,C))$.

```
#include <stdio.h>
#include <stdlib.h>

#define MAXKL 1000
#define MAXMN 1000
#define MAXF (MAXKL*MAXKL)

int R, C, M, N;
char mapa[MAXKL+2][MAXKL+2];
int mistnost[MAXKL+2][MAXKL+2];
int zasobnik[MAXF][2];
int velikost_zasobniku;
int prekryv[MAXF+1];
int pocet_mistnosti;
int akt_mistnosti, max_mistnosti;

void nacti_vstup(void)
{
  scanf("%d%d%d%d", &R, &C, &M, &N);
  for (int i = 1; i <= R; i++)
    scanf("%s", mapa[i] + 1);
}
```

```
void prohod(char *x, char *y)
{
  char t;
  t=*x; *x=*y; *y=t;
}

void transponuj(void)
{
  int i, j, t;
  int RC = R < C ? R : C;

  for (i = 0; i <= R + 1; i++)
    for (j = 0; j <= C + 1; j++)
      if (i > RC + 1 || j > RC + 1 || i < j)
        prohod(&mapa[i][j], &mapa[j][i]);

  t=R; R=C; C=t;
  t=M; M=N; N=t;
}

void zkus_jit(int x, int y)
{
  if (!mistnost[x][y] && mapa[x][y] == '.')
    {
      mistnost[x][y] = pocet_mistnosti;
      zasobnik[velikost_zasobniku][0] = x;
      zasobnik[velikost_zasobniku][1] = y;
      velikost_zasobniku++;
    }
}

void najdi_mistnosti(void)
{
  for (int i = 1; i <= R; i++)
    for (int j = 1; j <= C; j++)
      if (!mistnost[i][j] && mapa[i][j] == '.')
        {
          pocet_mistnosti++;

          zkus_jit(i, j);
          while (velikost_zasobniku > 0)
            {
              velikost_zasobniku--;
              int x = zasobnik[velikost_zasobniku][0];
              int y = zasobnik[velikost_zasobniku][1];
              zkus_jit(x - 1, y);
              zkus_jit(x + 1, y);
              zkus_jit(x, y - 1);
              zkus_jit(x, y + 1);
            }
        }
}

void zvetsi_prekryv(int i, int j)
{
  if (mistnost[i][j] && ++prekryv[mistnost[i][j]] == 1)
    akt_mistnosti++;
}
```

```
void zmensi_prekryv(int i, int j)
{
  if (mistnost[i][j] && --prekryv[mistnost[i][j]] == 0)
    akt_mistnosti--;
}

void vyzkousej_radek(int r)
{
  int j, k;

  for (j = 1; j <= N; j++)
    for (k = -1; k <= M; k++)
      zvetsi_prekryv(r + k, j);
  for (k = 0; k < M; k++)
    zvetsi_prekryv(r + k, N + 1);
  if (akt_mistnosti > max_mistnosti)
    max_mistnosti = akt_mistnosti;

  for (j = 1; j <= C - N; j++)
    {
      for (k = 0; k < M; k++)
        {
          zmensi_prekryv(r + k, j - 1);
          zvetsi_prekryv(r + k, j + N + 1);
        }
      zmensi_prekryv(r - 1, j);
      zmensi_prekryv(r + M, j);
      zvetsi_prekryv(r - 1, j + N);
      zvetsi_prekryv(r + M, j + N);

      if (akt_mistnosti > max_mistnosti)
        max_mistnosti = akt_mistnosti;
    }
  for (j = C - N + 1; j <= C; j++)
    for (k = -1; k <= M; k++)
      zmensi_prekryv(r + k, j);
  for (k = 0; k < M; k++)
    zmensi_prekryv(r + k, C - N);
}

void vyzkousej_diry(void)
{
  for (int i = 1; i <= R - M + 1; i++)
    vyzkousej_radek(i);
}

int main(void)
{
  nacti_vstup();
  if (M > N)
    transponuj();

  najdi_mistnosti();
  vyzkousej_diry();
  printf("%d\n", max_mistnosti);
  return 0;
}
```