P-II-1 Podposloupnost

Naším úkolem bylo nalézt nejdelší souvislý úsek zadané posloupnosti tak, aby aritmetický průměr prvků tohoto úseku byl právě k.

První a nejsnadnější řešení, které nás napadne, je zkusit postupně každý souvislý úsek a spočítat jeho průměr. Stačí tedy použít dva cykly, které budou určovat začátek a konec úseku, ve třetím cyklu pak spočítáme součet prvků tvořících tento úsek. Následně součet vydělíme počtem prvků a zkontrolujeme, zda je roven k.

```
int n, k;
int A[500];                    // Vstupní posloupnost
int vyszac=0, vyskon=-1;       // Začátek a konec výsledného úseku

for (int zac=0; zac<n; zac++)
    for(int kon=zac; kon<n; kon++) {
        int soucet=0;
        // Sčítáme prvky úseku
        for (int i=zac; i<=kon; i++) soucet += A[i];
        if (soucet == k*(kon-zac+1) && vyskon-vyszac < kon-zac) {
            vyszac=zac; vyskon=kon;
        }
    }
```

V programu vidíte, že jsme vlastně dělení nikde nepoužili. Operace dělení je totiž nepřesná. Označíme-li si s součet prvků v úseku a p počet prvků v úseku, můžeme upravit rovnici $s/p = k$ na $s = k \cdot p$. Tím dostaneme jednoduchý program, který všechno počítá naprosto přesně. Jeho časová složitost je $\Theta(n^3)$.

Zlepšení

Podle pravidel hodnocení úlohy dostaneme za řešení s touto časovou složitostí nejvýše 4 body. Je tedy třeba výpočet urychlit. V předchozím programu jsme vykonávali zbytečnou práci, když jsme počítali součet prvků v každém zkoumaném úseku zvlášť. Opakovaně jsme totiž sčítali stejná čísla.

První dva cykly v programu znamenají: nejprve zvolíme začátek úseku *zac* a potom pro tento začátek postupně volíme polohu konce úseku *kon*. Proměnná *kon* se přitom vždy zvyšuje o 1, což znamená, že se zkoumaný úsek prodlužuje o jeden nový prvek, který je na pozici *kon*. Součet aktuálního úseku se tedy liší od součtu předchozího úseku pouze o hodnotu toho prvku, který do úseku právě přibyl. Takto dostáváme řešení, které postupně projde všechny souvislé podposloupnosti a každou z nich zpracuje v konstantním čase. Jeho časová složitost je proto $\Theta(n^2)$.

```
int n,k;
int A[5000];                   // Vstupní posloupnost
int vyszac=0, vyskon=-1;       // Začátek a konec výsledného úseku
```

```
for (int zac=0; zac<n; zac++) {
    // Průběžně sčítáme
    int soucet=0;
    for (int kon=zac; kon<n; kon++) {
        soucet += A[kon];
        if (soucet == k*(kon-zac+1) && vyskon-vyszac < kon-zac) {
            vyszac=zac; vyskon=kon;
        }
    }
}
```

Vzorové řešení

Při hledání ještě lepšího řešení se musíme na naši úlohu podívat trochu jinak. Problémem je, že zkoumáme více parametrů najednou – potřebujeme znát součet úseku i počet jeho prvků. Lepší by bylo, kdyby stačilo sledovat jenom součet úseku a přímo z něj vidět, zda má tento úsek průměr rovný hodnotě k.

Položme si otázku, kdy má nějaký úsek průměr roven právě k. Je jasné, že tam nemůže být příliš mnoho prvků menších než k, ale zároveň ani mnoho prvků větších než k. Musí v něm být určitá rovnováha mezi prvky menšími a většími než k. Například posloupnost $(1, 3, 4, 4)$ má průměr 3, neboť dvě čtyřky vyváží přítomnost jedničky, která je o 2 menší.

Napíšeme si nyní rovnici, kterou se snažíme ověřit pro nějaký zkoumaný úsek čísel a_1 až a_p. Tato rovnice zní: $a_1 + a_2 + \ldots + a_p = pk$, což můžeme přepsat do tvaru $a_1 + a_2 + \ldots + a_p - pk = 0$. Nyní rozložíme člen $-pk$ na p samostatných členů $-k$ a rovnici upravíme do podoby $(a_1 - k) + (a_2 - k) + \ldots (a_p - k) = 0$. Vidíme, že každý prvek zkoumaného úseku se zmenšil právě o k. Proto je-li nyní součet zmenšených hodnot roven 0, pak má úsek průměr k.

Jinými slovy řečeno, když od každého prvku odečteme k, snížíme tím také průměr libovolného úseku o k. Tedy úseky, které měly v původní posloupnosti průměr k, odpovídají úsekům, které mají v upravené posloupnosti průměr 0. Proč nám to pomůže? Posloupnost má totiž průměr 0 právě tehdy, když má součet 0. A při hledání úseků, které mají součet 0, už nemusíme hledět na jejich délku.

Zadanou posloupnost si tedy upravíme tak, že od každého prvku odečteme k. Od této chvíle budeme slovem „posloupnost“ označovat tuto novou posloupnost, ve které hledáme úseky se součtem 0.

Zbývá provést ještě jeden myšlenkový krok. Součet úseků je možné počítat i jiným způsobem. Naplníme si pole $P[0 \ldots n]$ takovými hodnotami, aby prvek $P[i]$ udával součet prvních i prvků naší posloupnosti. Prvky takového pole P nazýváme prefixové součty dané posloupnosti. Hodnoty pole P snadno spočítáme v čase $\mathcal{O}(n)$ jedním průchodem: $P[0] = 0$ a každé další $P[i]$ určíme jako součet $P[i-1]$ a následujícího prvku posloupnosti. Když potom potřebujeme zjistit součet úseku od pozice *zac* do pozice *kon*, tuto hodnotu spočítáme jako $P[kon] - P[zac - 1]$.

Zvolíme pevný konec úseku *kon*. Které začátky pro tento zvolený konec určují úseky se součtem 0? Začátek *zac* je vhodný právě tehdy, když $P[kon] - P[zac - 1] = 0$, tzn. když $P[zac - 1] = P[kon]$. Abychom dostali úsek co nejdelší, chceme nalézt nejmenší *zac* s touto vlastností. Zjišťovat přítomnost nějakého konkrétního

prvku umíme snadno třeba pomocí vyvažovaného binárního stromu – v jazyce C++ je implementovaný jako `set` nebo `map`.

Celý algoritmus bude probíhat následovně. Nejprve odečteme od každého prvku původní posloupnosti číslo k. Postupně budeme volit stále větší a větší *kon*, které bude určovat, kde má končit náš úsek. Zároveň s tím si budeme pamatovat součet všech již zpracovaných prvků, tedy hodnotu $P[kon]$. V `map`u budeme mít pro každou předchozí prefixovou sumu uloženo nejmenší číslo *zac* takové, že prvních *zac* prvků má tento součet. Pro každé *kon* se podíváme do mapy, zda jsme již aktuální prefixový součet $P[kon]$ někdy dříve viděli. Jestliže ano, dozvěděli jsme se právě nejlepší začátek odpovídající aktuálnímu konci úseku. Jestliže ne, tak si pro hodnotu $P[kon]$ zapamatujeme, že poprvé byla dosažena na pozici *kon*. Nakonec už jenom vypíšeme nejdelší nalezený úsek.

Celková časová složitost programu bude $\mathcal{O}(n \log n)$, neboť pro každý konec se jen konstanta-krát podíváme do `map`u, jehož operace trvají $\mathcal{O}(\log n)$ času. Paměťová složitost je $\mathcal{O}(n)$.

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

int main() {
    // Načteme vstup a rovnou upravíme posloupnost
    int n, k, A[100047];
    scanf("%d %d",&n,&k);
    for (int i=0; i<n; i++)
        { scanf("%d",&A[i]); A[i] -= k; }

    // Procházíme posloupnost
    map<int,int> M;
    int prefix=0, vyszac=0, vyskon=-1;
    M[0] = -1;          // Zarážka (prefixové součty jsou nezáporné)
    for (int i=0; i<n; i++) {
        prefix += A[i];
        // Uvědomme si, že stejné číslo má být na pozici zac-1, takže přidáme ještě 1
        if (M.find(prefix) != M.end() && vyskon-vyszac+1 < i-M[prefix])
            { vyszac=M[prefix]+1; vyskon=i; }
        if (M.find(prefix) == M.end())
            M[prefix]=i;
    }
    printf("%d %d\n",vyszac+1,vyskon+1);
    return 0;
}
```

Alternativní řešení

V posledním kroku vůbec nebylo třeba používat pokročilou datovou strukturu. (Je to ale pohodlné, proto jsme takové řešení uvedli nejdříve.) Místo `map`u vystačíme i s obyčejným tříděním. Vezmeme si pole, jehož prvky jsou uspořádané dvojice čísel: $(0, 0), (P[1], 1), (P[2], 2), \ldots, (P[n], n)$. Toto pole uspořádáme – primárně podle první souřadnice, tedy odpovídajícího prefixového součtu, a sekundárně podle indexu, který mu odpovídá. V uspořádaném poli budou všechny indexy, kterým odpovídá stejná hodnota prefixového součtu, tvořit vždy souvislý úsek. Nejvzdálenější dvojici

indexů, kterým odpovídá stejný prefixový součet, dokážeme pomocí tohoto uspořádaného pole snadno určit v lineárním čase.

P-II-2 Body v rovině

Základní pomalé řešení

Začněme s pomalejším, ale zjevně funkčním řešením. Jednoduchou možností je zvolit nějakou konečnou množinu kandidátů na vhodné dělící přímky. Potom stačí vždy pro každý černý bod vyzkoušet všechny kandidáty a zjistit, zda některý z nich vyhovuje. Vhodnými kandidáty mohou být například přímky, které procházejí dvěma bílými body. Algoritmus potom vypadá tak, že pro každou dvojici bílých bodů vyzkoušíme, zda černý bod leží na opačné straně přímky než všechny ostatní bílé body. Černý bod na přímce ležet nesmí, u bílých bodů to nemusíme ani testovat, neboť podle zadání žádné tři bílé body neleží na přímce.

Testování, zda nějaký bod X leží na přímce YZ, napravo, nebo nalevo od ní, můžeme provést například pomocí znaménka vektorového součinu vektorů YZ a YX. Takovýto algoritmus by pro každou z řádově n^2 dvojic bílých bodů testoval řádově n bodů, a to všechno by prováděl opakovaně pro každý černý bod, tedy q-krát. Jeho výsledná časová složitost by proto byla $\mathcal{O}(qn^3)$.

Součástí správného řešení by ovšem měl být také důkaz, že naše množina kandidátů skutečně postačuje. Přesněji, potřebujeme dokázat, že pokud existuje (nějaká libovolná) rozdělující přímka, potom také jedna z námi zvolených přímek bude vyhovovat. Vezměme si tedy libovolnou vyhovující přímku. Můžeme ji „posouvat“ směrem od černého bodu, dokud přímka neprotne některý bílý bod. (V tomto okamžiku jsme tedy již dokázali, že pokaždé, když existuje rozdělující přímka, existuje také rozdělující přímka procházející některým bílým bodem.)

Pokud jsme měli štěstí, právě nalezená přímka prochází dvěma bílými body. Jestliže tomu tak není, začneme přímku otáčet kolem toho bílého bodu, kterým prochází. Nejprve zvolíme kladný směr otáčení. Mohou nastat dvě možnosti. Buď otáčená přímka nejprve "narazí" na druhý bílý bod a našli jsme vyhovující přímku, nebo nejprve narazí na černý bod, případně zároveň na bílý a černý bod. V takovém případě ji budeme otáčet opačným směrem. A protože otáčející se přímka pokryje celou rovinu dříve, než opět narazí na černý bod, musí tentokrát nejprve narazit na bílý bod. Tím je důkaz ukončen.

První předvýpočet

Právě uvedené řešení provádí mnoho zbytečné práce. Například je zjevné, že přímka s bílými body po obou svých stranách nebude nikdy vyhovovat. Takové přímky tedy nemá vůbec smysl opakovaně testovat pro každý černý bod.

Nejlepší bude zbavit se těchto špatných přímek hned na začátku. Na začátku výpočtu tedy provedeme předvýpočet v čase $\mathcal{O}(n^3)$ a pro každou přímku určenou dvěma bílými body zjistíme, zda všech $n-2$ zbývajících bílých bodů leží na jedné její straně, nebo ne. Tím nám zůstane jen $k \leq n^2$ přímek, které mají všechny bílé body na jedné straně. Pro každý černý bod nyní stačí zkontrolovat pouze jeho polohu vůči každé z k přímek. Takto dostaneme řešení s celkovou časovou složitostí $\mathcal{O}(n^3 + qk)$.

Celý předvýpočet ale dokážeme provést ještě mnohem efektivněji, a navíc bude hned jasné, že hodnota k bude vždy malá. Přímky, které má smysl testovat, budou totiž přesně odpovídat jednotlivým stranám konvexního obalu všech bílých bodů (takže jich bude nejvýše n).

Nám bude dokonce stačit jedna implikace: Jestliže všechny ostatní bílé body leží na stejné straně od přímky vedoucí body A a B, pak úsečka AB tvoří stranu jejich konvexního obalu. Toto zjevně platí, a proto každá přímka v naší množině kandidátů je skutečně prodloužením některé strany konvexního obalu.

Platí ale i opačná implikace, a proto každá strana konvexního obalu odpovídá nějaké možné rozdělující přímce. Jestliže totiž úsečka AB tvoří stranu konvexního obalu, musí všechny ostatní bílé body ležet na stejné straně od ní. (Dokážeme sporem: kdyby body C a D ležely na opačných stranách přímky AB, potom celý čtyřúhelník $CADB$ je součástí konvexního obalu, což je spor s tím, že AB je jeho strana.)

V domácím kole jsme se naučili, jak sestrojit konvexní obal v čase $\mathcal{O}(n \log n)$. Této znalosti můžeme nyní využít k výraznému zlepšení časové složitosti našeho algoritmu. Ten se bude skládat ze dvou fází:

1. Vezmeme n bílých bodů a v čase $\mathcal{O}(n \log n)$ sestrojíme jejich konvexní obal. Tím jsme získali množinu nejvýše n přímek, které stačí dále testovat.
2. Pro každý černý bod postupně vyzkoušíme každou z přímek nalezených v prvním kroku: vezmeme vždy libovolný třetí bílý bod a podíváme se, zda leží na opačné straně přímky. Každý černý bod proto zpracujeme v čase $\mathcal{O}(n)$.

Celková časová složitost tohoto algoritmu je tedy $\mathcal{O}(n \log n + qn)$.

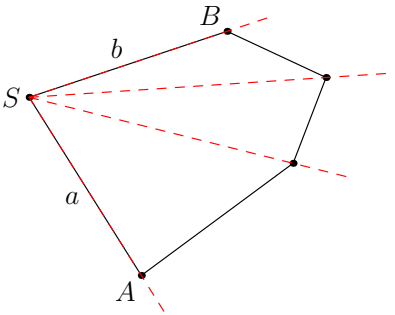
Alternativní implementace druhé části algoritmu: Můžeme využít toho, že algoritmus pro sestrojení konvexního obalu nám jeho strany najde uspořádané po obvodě. Když půjdeme po konvexním obalu například proti směru hodinových ručiček, víme, že bílé body máme stále nalevo od testované přímky. Stačí proto testovat, jestli černý bod leží napravo od ní.

Vzorové řešení

Z pozorování o konvexním obalu dokonce vyplývá, že rozdělující přímka existuje právě tehdy, když se černý bod nachází vně konvexního obalu. Jakmile totiž zkoumaný černý bod leží vně obalu, můžeme použít větu o separující přímce z řešení domácího kola. Samotný černý bod je také konvexní množinou. Opačná implikace je zjevná.

Jak zjistit lépe než v lineárním čase, zda leží černý bod v konvexním útvaru? Dobré způsoby jsou opět založeny na vhodném předvýpočtu. Je třeba rozřezat si daný útvar na vhodné kousky nějak systematicky rozložené vedle sebe a následně pro každý černý bod binárně vyhledat ten správný kousek, v němž by mohl ležet. Existuje více konkrétních implementací, můžeme například rozřezat náš útvar svislými řezy vedoucími přes všechny jeho vrcholy.

My si ukážeme jednu z implementačně nejjednodušších možností. Jeden vrchol našeho konvexního k-úhelníka si označíme jako speciální a označíme ho S. Z něj vycházejí dvě strany: strana a do bodu, který označíme A, a strana b do bodu B. Představme si nyní $k-1$ polopřímek, které vedou z bodu S přes každý z ostatních vrcholů (včetně vrcholů A a B).



Tyto polopřímky nám rozdělí rovinu na $k-1$ úhlů: jeden vypuklý, který neobsahuje vnitřek našeho k-úhelníka, a $k-2$ zbývajících. Vypuklý úhel si můžeme představit jako sjednocení dvou polorovin: jedné určené stranou a a druhé určené stranou b. V každém ze zbývajících úhlů ta část, která patří do našeho k-úhelníka, tvoří trojúhelník. Dvě strany trojúhelníka leží na našich polopřímkách, třetí tvoří vždy jedna ze stran k-úhelníka.

Jak nyní zpracujeme konkrétní černý bod? Začneme tím, že ověříme, zda leží v jedné z prázdných polorovin určených stranami a a b. Jestliže ano, našli jsme rozdělující přímku a končíme. Jestliže ne, černý bod se nachází někde v úhlu ASB. Tento úhel máme rozdělen na $k-2$ menších. Pomocí $\mathcal{O}(\log k)$ otázek tvaru „leží bod nalevo od této polopřímky?“ umíme binárním vyhledáváním určit ten úhel, v němž černý bod leží. Následně už jenom stačí vzít stranu našeho konvexního obalu, která dotyčnému úhlu odpovídá, a podívat se, zda bod leží na její správné straně. (Když se náhodou stane, že černý bod leží přesně na jedné z našich polopřímek, můžeme ho zařadit do libovolného z úhlů určených dotyčnou polopřímkou.)

Sestavili jsme tedy algoritmus pracující v čase $\mathcal{O}(n \log n + q \log k)$. A protože $k \leq n$, můžeme jeho časovou složitost dále shora odhadnout jako $\mathcal{O}((n+q) \log n)$.

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

// Bod
struct point {
    long long x, y;
    point(long long x=0, long long y=0) : x(x), y(y) {}
};
```

```
// Operátor < na uspořádání bodů
bool operator< (const point &A, const point &B)
{ return A.x < B.x || ( A.x == B.x && A.y < B.y ); }

// Vektorový součin: vrátí >0 / 0 / <0 podle toho, zda OAB zatáčí proti směru
// ručiček / jde rovně / zatáčí po směru
long long cross(const point &O, const point &A, const point &B)
{ return (A.x-O.x)*(B.y-O.y)-(A.y-O.y)*(B.x-O.x); }

// Skalární součin: pro O, A, B na přímce vrátí >0 jestliže jsou A a B
// na stejné straně O, <0 jestliže jsou na opačných stranách
long long dot(const point &O, const point &A, const point &B)
{ return (A.x-O.x)*(B.x-O.x)+(A.y-O.y)*(B.y-O.y); }

// Konvexní obal pomocí Grahamova algoritmu;
// předpokládá, že P je uspořádané zleva doprava
vector<point> convex_hull(vector<point> P) {
    int n = P.size(), k = 0;
    vector<point> H(2*n);
    for (int i = 0; i < n; i++) {
        while (k >= 2 && cross(H[k-2], H[k-1], P[i]) <= 0) k--;
        H[k++] = P[i];
    }
    for (int i = n-2, t = k+1; i >= 0; i--) {
        while (k >= t && cross(H[k-2], H[k-1], P[i]) <= 0) k--;
        H[k++] = P[i];
    }
    H.resize(k-1);
    return H;
}

// Binární vyhledávání: vrátí pořadové číslo úsečky, která určuje přímku obsahující
// černý bod (řečený též sysel), resp. 0, pokud je černý bod vlevo, 10^8, jestliže
// je vpravo
long long binsrch(point& sysel, vector<point>& hull, unsigned long long size) {
    if (cross(hull [0], hull [1], sysel) < 0) return 0;
    if (cross(hull [0], hull [size-1], sysel) > 0) return 1e8;
    long long lower = 1, upper = size - 1;
    while (upper - lower > 1) {
        if (cross(hull [0], hull [(lower+upper) / 2], sysel) > 0)
            lower = (lower + upper) / 2;
        else
            upper = (lower + upper) / 2;
    }
    return lower;
}

int main() {
    int N; cin >> N;
    vector<point> P(N);
    for (int n=0; n<N; ++n)
        cin >> P[n].x >> P[n].y;
    sort(P.begin(), P.end());
    vector <point> hull = convex_hull(P);
    int Q; cin >> Q;
    point sysel;
    long long answer;
```

```
    for (int i = 0; i < Q; ++i) {
        cin >> sysel.x >> sysel.y;
        answer = binsrch(sysel, hull, hull.size());
        if (answer == 0) {
            cout << hull [0].x << ' ' << hull [0].y << ' ';
            cout << hull [1].x << ' ' << hull [1].y << endl;
        } else if (answer == 1e8) {
            cout << hull [0].x << ' ' << hull [0].y << ' ';
            cout << hull.back().x << ' ' << hull.back().y << endl;
        } else if (cross(hull [answer], hull [answer + 1], sysel) < 0) {
            cout << hull [answer].x << ' ' << hull [answer].y << ' ';
            cout << hull [answer + 1].x << ' ' << hull [answer + 1].y << endl;
        } else {
            cout << "Primka neexistuje" << endl;
        }
    }
    return 0;
}
```

P-II-3 Počítačová síť

Máme souvislý graf, v němž je stejný počet vrcholů a hran. Chceme vybrat co nejvíce jeho hran tak, aby s žádným vrcholem nesousedilo více vybraných hran. Jinými slovy, hledáme maximální párování v grafu.

Protože vrcholů a hran je stejný počet, v grafu je jistě obsažen právě jeden cyklus. Libovolná kostra grafu má totiž $n-1$ hran a nejsou v ní žádné cykly. Když k ní přidáme chybějící n-tou hranu, vznikne zmíněný jediný cyklus. Někde v našem grafu je tedy několik vrcholů spojených do cyklu a na každý z nich může být připojeno několik stromů, v nichž už žádné cykly nejsou.

Vyřešíme nejprve jednodušší úlohu: představme si, že máme pouze strom, tzn. souvislý graf s n vrcholy a $n-1$ hranami. V něm hledáme maximální párování. Na takto zjednodušenou úlohu se dá použít dynamické programování. Nejprve strom zakořeníme – libovolný jeho vrchol prohlásíme za kořen stromu. Potom budeme postupně od listů stromu směrem ke kořeni procházet všechny jeho vrcholy. V každém vrcholu v nás bude zajímat, jaké nejlepší párování lze nalézt v podstromu pod vrcholem v. Spočítáme si v něm vždy dvě čísla: $A(v)$ udává velikost nejlepšího párování, které může použít i samotný vrchol v, zatímco $B(v)$ určuje velikost takového nejlepšího párování, v němž vrchol v použít nesmíme (neboť chceme do párování zařadit hranu, která spojuje vrchol v s jeho otcem).

Jak spočítáme číslo $B(v)$? Když bude vrchol v spojen se svým otcem, určitě nebude spojen s žádným ze svých synů. Stačí tedy sečíst hodnoty $A(c)$ všech synů vrcholu v.

Jak spočítáme číslo $A(v)$? Tentokrát musíme vzít maximum z více možností, jak může párování vypadat. První možností je, že v nespojíme s žádným z jeho synů, takže velikost párování bude opět rovna součtu hodnot $A(c)$ všech jeho synů. Druhou možností je, že hranu mezi vrcholem v a některým z jeho synů u zařadíme do párování. Velikost takového párování už nebude rovna součtu všech $A(c)$, ale ze součtu odpadne hodnota $A(u)$ a naopak místo ní přibude $1 + B(u)$.

Jestliže budeme procházet strom zdola nahoru a pro každý vrchol v si zapamatujeme čísla $A(v)$ a $B(v)$, celý výpočet provedeme v lineárním čase. Výsledkem potom bude číslo $A(r)$ v kořeni celého stromu r.

Vraťme se nyní k původní úloze. Zkoumaný graf není stromem, ale je v něm navíc jeden cyklus. Vyhledáme tento cyklus a zvolíme v něm jednu libovolnou hranu. Tato hrana buď v párování bude, nebo nebude. Jestliže tam nebude, můžeme ji z grafu hned odstranit a zůstane nám strom, pro který již známe řešení. Jestliže tam bude, její koncové dva vrcholy určitě nebudou spárované s žádným jiným vrcholem, takže můžeme odstranit z grafu všechny ostatní hrany, které z těchto dvou vrcholů vedou. Tím se graf rozpadne na několik stromů, které již opět umíme vyřešit. Stačí tedy vyzkoušet uvedené dva případy a vybrat z nich maximum. Uvažované případy jsou jen dva, a proto celé řešení má nadále lineární časovou i paměťovou složitost.

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

int N;
vector<vector<int> > sousedi;
vector<bool> visited;
vector<int> A, B;
int cyklus_a, cyklus_b;

// Prohledávání do hloubky pro nalezení cyklu v původním grafu
void dfs_najdi_cyklus(int v, int parent) {
  if (visited[v]) {
    cyklus_a = v;
    cyklus_b = parent;
    return;
  }

  visited[v] = 1;

  for (unsigned i = 0; i < sousedi[v].size(); i++) {
    int c = sousedi[v][i];
    if (c == parent) continue;
    dfs_najdi_cyklus(c, v);
    if (cyklus_a != -1 && cyklus_b != -1) return;
  }
}

// Vypočítá pro vrchol v hodnoty A[v] a B[v]
void vypocitej_vrchol(int v, int parent) {
  B[v] = 0;
  for (unsigned i = 0; i < sousedi[v].size(); i++) {
    int c = sousedi[v][i];
    if (c == parent) continue;
    vypocitej_vrchol(c, v);
    B[v] += A[c];
  }

  A[v] = B[v];
  for (unsigned i = 0; i < sousedi[v].size(); i++) {
    int u = sousedi[v][i];
```

```
    if (u == parent) continue;
    A[v] = max(A[v], B[v] - A[u] + (1 + B[u]));
  }
}

// Zjistí velikost maximálního párování za předpokladu, že graf neobsahuje cykly
int maximum_na_stromech() {
  A.assign(N, -1);
  B.assign(N, -1);

  int vysledek = 0;
  for (int v = 0; v < N; v++) {
    if (A[v] == -1 && B[v] == -1) {
      vypocitej_vrchol(v, -1);
      vysledek += A[v];
    }
  }

  return vysledek;
}

void vyrad_z_vectoru(vector<int>& v, int hodnota) {
  vector<int>::iterator pozice = find(v.begin(), v.end(), hodnota);
  if (pozice != v.end()) v.erase(pozice);
}

int main() {
  cin >> N;
  sousedi.resize(N);
  visited.resize(N, 0);
  for (int i = 0; i < N; i++) {
    int a, b; cin >> a >> b; a--; b--;
    sousedi[a].push_back(b);
    sousedi[b].push_back(a);
  }

  // Najdeme nějaké dva vrcholy, které jsou v cyklu a sousedí
  cyklus_a = cyklus_b = -1;
  dfs_najdi_cyklus(0, -1);

  // Možnost 1: hranu (cyklus_a, cyklus_b) nevybereme do párování
  // (takže ji můžeme z grafu hned odstranit a zůstane jen strom)
  vyrad_z_vectoru(sousedi[cyklus_a], cyklus_b);
  vyrad_z_vectoru(sousedi[cyklus_b], cyklus_a);
  int moznost1 = maximum_na_stromech();

  // Možnost 2: hranu (cyklus_a, cyklus_b) vybereme do párování
  // (takže ostatní s nimi ani nemusí sousedit a zůstane jen les)
  for (int i = 0; i < N; i++) {
    vyrad_z_vectoru(sousedi[i], cyklus_a);
    vyrad_z_vectoru(sousedi[i], cyklus_b);
  }
  sousedi[cyklus_a].clear();
  sousedi[cyklus_a].push_back(cyklus_b);
  sousedi[cyklus_b].clear();
  sousedi[cyklus_b].push_back(cyklus_a);
  int moznost2 = maximum_na_stromech();
  cerr << cyklus_a << "," << cyklus_b << " " << moznost1 << "," << moznost2 << endl;
```

```
  int vysledek = max(moznost1, moznost2);
  cout << vysledek << endl;
}
```

Jiné korektní řešení je založeno na hladové myšlence: Dokud máme v grafu nějaký vrchol u stupně 1, můžeme jedinou hranu uv vedoucí z vrcholu u zařadit do vytvářeného párování a následně odstranit z grafu vrcholy u, v a všechny ostatní hrany vedoucí z v, které už do řešení zařadit nemůžeme.

Důkaz: Uvažujme libovolné optimální párování. Je-li v něm použita hrana uv, jsme hotoví. Pokud ne, musí v něm být použita nějaká jiná hrana vw, jinak bychom hranu uv mohli do párování přidat a dostali bychom lepší párování. Potom ale můžeme hranu vw z párování vyřadit a místo ní tam přidat právě hranu uv. Tím jsme zjevně opět dostali korektní párování a navíc stejné velikosti, tedy také optimální.

Výše uvedený postup budeme stále opakovat, dokud nedostaneme graf, který již žádné vrcholy stupně 1 nemá. Obecně mohou nastat dva případy: Jestliže nám už nic z grafu nezbylo, máme nalezeno maximální párování grafu. Mohl nám ale ještě zůstat graf, který má všechny vrcholy stupně aspoň 2. Pro náš původní graf ze zadání úlohy nastává tento druhý případ: zůstane nám nevyřešený právě náš jediný cyklus v grafu (v němž mají všechny vrcholy stupeň přesně 2). Cyklus ale vyřešíme snadno: má-li k vrcholů, pak jeho nejlepší párování má zjevně velikost $\lfloor k/2 \rfloor$.

Při implementaci stačí pamatovat si stupně jednotlivých vrcholů a jakmile stupeň některého vrcholu klesne na 1, zařadíme tento vrchol do fronty na zpracování.

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

int N;
vector< vector<int> > sousedi;
vector<int> stupen;
vector<bool> pouzil;

int main() {
    cin >> N;
    sousedi.resize(N);
    stupen.resize(N, 0);
    pouzil.resize(N, false);
    for (int i=0; i<N; ++i) {
        int a, b; cin >> a >> b; a--; b--;
        sousedi[a].push_back(b); ++stupen[a];
        sousedi[b].push_back(a); ++stupen[b];
    }

    queue<int> Q;
    for (int i=0; i<N; ++i) if (stupen[i]==1) Q.push(i);

    int reseni = 0;
    while (!Q.empty()) {
        int kde = Q.front(); Q.pop();
        if (pouzil[kde] || stupen[kde] == 0)
            continue;   // Tento vrchol jsme použili nebo zablokovali
```

```
        int sous = -1;
        for (int a : sousedi[kde]) if (!pouzil[a]) sous=a;

        pouzil[kde] = pouzil[sous] = true;
        ++reseni;
        for (int a : sousedi[kde])
            if (!pouzil[a]) { --stupen[a]; if (stupen[a]==1) Q.push(a); }
        for (int a : sousedi[sous])
            if (!pouzil[a]) { --stupen[a]; if (stupen[a]==1) Q.push(a); }
    }

    int na_cyklu = 0;
    for (int i=0; i<N; ++i)
        if (!pouzil[i] && stupen[i]>1) ++na_cyklu;
    reseni += (na_cyklu/2);

    cout << reseni << endl;
}
```

P-II-4 Mimozemské počítače

Podúloha A (3 body)

Mimozemšťané nám dodali sálový KSP, jehož funkce `housenka(n,E)` rozhoduje problém existence housenkové kostry v daném neorientovaném grafu. Pomocí této funkce chceme rozhodnout, zda graf G obsahuje nějakou hamiltonovskou cestu.

Vlastnost „graf obsahuje housenkovou kostru“ si můžeme ekvivalentně zformulovat takto: „graf obsahuje takovou cestu, že každý vrchol grafu, který na této cestě neleží, s ní sousedí“. To už se začíná podobat hamiltonovské cestě, ale s jedním rozdílem – cesta nemusí procházet všemi vrcholy, stačí jít kolem některých z nich.

Abychom vyřešili zadanou úlohu, potřebujeme vymyslet vhodnou úpravu zadaného grafu G, pro kterou platí: Jestliže původní graf G obsahoval hamiltonovskou cestu, bude upravený graf G' obsahovat housenkovou kostru. A naopak, jestliže G žádnou hamiltonovskou cestu neobsahoval, ani upravený graf G' housenkovou kostru obsahovat nesmí.

Existuje velmi jednoduchá transformace s právě uvedenou vlastností: ke každému vrcholu v v grafu G přidáme nový vrchol v' a hranu vv'.

Pokud v původním grafu existovala hamiltonovská cesta, pak v upraveném grafu zjevně existuje housenková kostra: stačí odstranit všechny hrany kromě jedné hamiltonovské cesty v původním grafu a těch hran, které jsme do grafu G přidali.

Předpokládejme naopak, že upravený graf G' obsahuje housenku. Jak může vypadat její „hlavní“ cesta? Vrcholy, které jsme přidali do G, mají všechny stupeň 1, proto se takový vrchol může nacházet jedině na začátku, nebo na konci cesty. Zbytek cesty tedy nutně tvoří *některé* vrcholy původního grafu G. No a co by se stalo, kdyby mezi nimi některý vrchol v chyběl? Potom s ním sousedící vrchol v' neleží ani na naší cestě, ani s ní nemůže sousedit, což je spor. Proto jestliže G' obsahuje housenku, její „hlavní“ cesta nutně odpovídá hamiltonovské cestě v původním grafu G.

```
def cesta(n, E):
    for i in range(n): E.append( (i, n+i) )
    housenka(2*n, E)
```

Podúloha B (3 body)

Mimozemšťané nám dodali sálový KSP, jehož funkce `housenka(n,E)` rozhoduje problém existence housenkové kostry v daném neorientovaném grafu. Pomocí této funkce chceme rozhodnout, zda graf G obsahuje nějakou hamiltonovskou kružnici.

V podúloze A jsme si ukázali, jak lze funkci `cesta(n,E)` naprogramovat pomocí funkce `housenka(n,E)`. Z domácího kola víme, jak lze funkci `kruznice(n,E)` implementovat pomocí funkce `cesta(n,E)`. Když tyto dva postupy složíme dohromady, máme podúlohu B vyřešenou.

Detailnější popis začneme připomenutím, jak fungovala transformace z domácího kola. Z grafu G vytvoříme nový graf G' tak, že přidáme vrchol n, který bude kopií vrcholu 0; dále přidáme vrcholy $n+1$ a $n+2$ a hrany mezi 0 a $n+1$ a mezi n a $n+2$. Takto sestrojený graf G' obsahuje hamiltonovskou cestu právě tehdy, když G obsahoval hamiltonovskou kružnici.

Jestliže nemáme k dispozici funkci `cesta(n,E)`, ale máme funkci `housenka(n,E)`, musíme následně na graf G' použít konstrukci z řešení podúlohy A: každý vrchol G' dostane nového kamaráda, který s ním bude spojen hranou. Pro takto sestrojený graf G'' platí: G'' obsahuje housenkovou kostru právě tehdy, když graf G' obsahoval hamiltonovskou cestu.

Celé řešení bude tedy vypadat následovně: vezmeme vstupní graf G, z něho vytvoříme G', z něho dále získáme graf G'', a ten zadáme do našeho sálového KSP. Pokud KSP rozsvítí zelené světlo, znamená to, že G'' obsahuje housenkovou kostru, a tedy G obsahuje hamiltonovskou kružnici. Naopak červené světlo znamená, že G hamiltonovskou kružnici neobsahuje. Přesně toho jsme chtěli dosáhnout.

```
def kruznice(n, E):
    # Sestrojíme si seznam sousedů vrcholu 0
    sousede0 = []
    for x, y in E:
        if x==0: sousede0.append(y)
        if y==0: sousede0.append(x)

    # Do grafu přidáme nový vrchol n, který je kopií vrcholu 0
    E += [ (n, x) for x in sousede0 ]

    # a ještě dva nové vrcholy, které slouží jako konce cesty
    E += [ (0, n+1), (n, n+2) ]

    # Nakonec zavoláme funkci cesta(), která rozsvítí správné světlo
    # (zde volaná funkce cesta() je naše funkce z řešení podúlohy A)
    cesta(n+3, E)
```

Podúloha C (4 body)

Mimozemšťané nám tentokrát dodali kufříkový KSP, jehož funkce `existuje_pokryti(k,n,m,Z)` rozhoduje problém existence pokrytí množinami, které má velikost nejvýše k. Pomocí tohoto kufříkového KSP chceme nalézt nejmenší možný počet žáků, kteří jsou předsedou aspoň jednoho zájmového kroužku.

Řešení úlohy je založeno na jednoduchém pozorování: KSP, který máme k dispozici, umí téměř přímo řešit naši úlohu, jenom si ji musíme vhodně zformulovat.

Na vstupu jsme dostali ke každému kroužku seznam žáků, kteří ho navštěvují. Tuto informaci si ale dokážeme reprezentovat i opačně: pro každého žáka z sestrojíme množinu kroužků $K(z)$, do nichž chodí.

Představme si nyní, že postupně vybíráme žáky, kteří budou patřit do rady předsedů. Vždy, když nějakého vybereme, uděláme z něj předsedu všech kroužků, do nichž chodí a které ještě předsedu nemají. Jakmile tedy vybereme takovou skupinu žáků, že každý kroužek v ní má zastoupení, budeme mít platnou radu předsedů.

Otázku ze zadání tedy můžeme položit následovně: „Kolik nejméně z množin $K(z)$ musíme vybrat, aby jejich sjednocení obsahovalo všechny kroužky?“

A toto je přesně otázka na velikost pokrytí množinami. Pomocí kufříkového KSP, který máme k dispozici, dokážeme správnou odpověď nalézt na $\mathcal{O}(\log z)$ otázek pomocí binárního vyhledávání.

```
# Načteme údaje o kroužcích, převedeme je na údaje o žácích

Z = int(input())        # počet žáků
K = int(input())        # počet kroužků

# Pro každého žáka seznam jeho kroužků
krouzky = [ [] for z in range(Z) ]
for k in range(K):
    krouzek = [ int(x) for x in input().split() ]
    for x in krouzek: krouzky[x].append(k)   # žák x navštěvuje kroužek k

# Budeme binárně vyhledávat minimální počet žáků-předsedů
# na začátku víme, že 0 předsedů je málo a že Z určitě stačí

malo, dost = 0, Z
while dost > malo+1:
    stred = (malo + dost) // 2
    if existuje_pokryti( stred, K, Z, krouzky ): dost = stred
    else: malo = stred

print(dost)
```