SZAKDOLGOZAT

Tőzsér Tamás

Debrecen

2010

Debreceni Egyetem
Informatikai Kar

MI algoritmusok nemzetközi programozói versenyek feladatainak megoldásában

Témavezető:
Kósa Márk
egyetemi tanársegéd

Készítette:
Tőzsér Tamás
programtervező informatikus
hallgató

Debrecen
2010

Tartalomjegyzék

1 Bevezetés

A szakdolgozat célja, hogy a programozói versenyeken előforduló mesterséges intelligencia algoritmusokkal megoldható feladatok felismerésében és megoldásában segítséget nyújtson a versenyekre való felkészülés során.

Számos nemzetközi programozó verseny létezik, az egyes versenyek szabályai és így a stílusuk is nagy mértékben eltér egymástól. A szakdolgozaton belül főleg az ACM[1] feladatain fogom bemutatni a megoldás alapelveit, de ezek természetesen más versenyeken is használhatóak. Az ACM feladatok legkedveltebb nyelve a C++ itt is ezt fogom használni a példakódoknál.

A programozói versenyekről általánosan elmondható, hogy számos különböző feladattípusból állítják össze a feladatsorokat. Az egyes típusokhoz léteznek általános megoldási módszerek, ezeknek a követése nagymértékben meggyorsíthatja a feladatok megoldását. Sokszor a feladatoknak a megfelelő típushoz rendelése önmagában is egy nehéz feladat, így a sikeres feladatmegoldásnak elengedhetetlen része a feladatok gyors kategóriákba sorolása. Az első lépés tehát az, hogy megkeressük a feladatoknak azon jellemzőit, melyek alapján feltételezhetjük, hogy azokat mesterséges intelligencia algoritmusokkal megoldhatjuk. A szakdolgozat első részében erről lesz szó.

Ezt követően részletesen meg kell ismerkednünk a leggyakrabban használt MI algoritmusokkal, mivel ezek pontos ismerete elengedhetetlen a gyors feladatmegoldáshoz. A versenyeken általában szűk időkeretek állnak rendelkezésre és nincs idő az alapvető algoritmusokban elkövetett hibák keresésére.

Azzal, hogy felismertük egy feladatról, hogy az egy mesterséges intelligencia algoritmussal megoldható feladat még nem jelenti azt, hogy azzal a módszerrel is érdemes azt megoldani. Ugyanis az MI algoritmusok általánosan véve lassúak és erőforrás igényesek, csak akkor célszerű őket használni, ha nem rendelkezünk olyan háttérismerettel, amit felhasználva egyszerűsíthetnénk a problémát. A szakdolgozatban ezekről a háttérismeretekről is lesz szó illetve, hogy ezeket

1 http://cm.baylor.edu/welcome.icpc

felhasználva hogyan gyorsíthatjuk fel az algoritmusokat.

Az egyes feladattípusok tanulása során mindig az első feladatok megoldása a legnehezebb, ezért bemutatom néhány feladat teljes megoldását az elejétől a végéig, hogy könnyebbek legyenek az első lépések.

2 Az MI feladatok felismerése

Az mesterséges intelligencia feladatok megoldásának nehézsége, hogy nehéz felismerni a feladatok szövege mögött megbújó feladattípust. A feladatok kategorizálása során hasznos lehet a feladatok átfogalmazása. Ha például a feladatot át lehet úgy fogalmazni, hogy abban gráfok vagy fák jelenjenek meg akkor jó eséllyel MI feladatot találtunk. Általános jel az is, ha a feladatban valamilyen útvonal vagy optimális állapot meghatározása a cél.

Vegyünk egy példát, az Edit Step Ladders[2] feladatban a cél, hogy meghatározzuk egy szótárban a leghosszabb szósor hosszát, amelyben az egyik szóból egy másikba csak bizonyos szabályoknak megfelelően lehet átmenni. Első ránézésre ezt a problémát egy dinamikus programozási feladatnak is tekinthetnénk. Azonban felfigyelhetünk arra, hogy a feladatot átfogalmazhatjuk egy gráf problémává. A szavak egy gráf csúcsai, az élek azt jelképezik két csúcs között, hogy a szabályoknak megfelelően áttranszformálhatjuk-e az elsőt a másodikba. A kérdés pedig az, hogy mekkora a legnagyobb távolság két csúcs között. Ezt pedig meghatározhatjuk például útkeresések segítségével. Azonban ez a megoldás nagyon lassú, a későbbi fejezetekben majd látni fogjuk, hogy a gráfok felépítését gyakran teljesen elhagyhatjuk a feladat megoldása során.

2 http://online-judge.uva.es/p/v100/10029.html

3 A legfontosabb algoritmusok

3.1 Alapvető gráf algoritmusok

3.1.1 A gráfok ábrázolása

Számos különböző módja van a gráfok ábrázolásának:

- Szomszédsági mátrix: n csúcs esetén egy nxn-es mátrixban egyes érték szerepel az (i, j) helyen, ha i-ből tart él j-be, és nulla ha nem. Ez egy nagyon egyszerű ábrázolási módszer, de csak kis számú csúcs esetén alkalmazható.
- Szomszédsági lista listában: Minden csúcshoz fenntartunk egy láncolt listát, amelyben szerepelnek azok a csúcsok, amelyekbe él mutat belőle. Ez a módszer tártakarékos, de nehézzé teszi a „van-e él az i és j csúcs között?” típusú kérdések megválaszolását. Azonban ezeket a kérdéseket gyakran elkerülhetjük.
- Szomszédsági lista mátrixban: A mátrix minden sora egy-egy csúcshoz tartozik. A sorokban pedig azok a csúcsok szerepelnek, amelyekhez él megy a sorhoz tartozó csúcsból. Ez a módszer ránézésre ötvözi az előző két módszer hátrányait. De ha előre tudjuk, hogy egy csúcshoz hány él tartozik, akkor hatékonyan foglalhatunk számára helyet, és elkerülhetjük a mutatók használatát.
- Élek táblája: Egy tömbben vagy listában két végpontjukkal tartjuk nyilván az éleket. Ez nagyon nehézzé teszi a „melyek az i csomópont szomszédai?” típusú kérdések megválaszolását. De bizonyos algoritmusokban ez a módszer is jól használható.

A megfelelő ábrázolásmód tehát a konkrét feladattól függ. Ezért érdemes mindegyik típust begyakorolni.

Szomszédsági lista mátrixban egy megvalósítása:[3]

```
#define MAXV          100   /* a csúcsok maximális száma */
#define MAXDEGREE     50    /* maximális szomszédok száma */

typedef struct {
    int edges[MAXV+1][MAXDEGREE];
                /* szomszédsági információk */
    int degree[MAXV+1];
                /* a szomszédok száma minden csúcshoz */
    int nvertices;          /* a csúcsok száma a gráfban */
    int nedges;             /* az élek száma a gráfban */
} graph;
```

3 Programming Challenges, 192. oldal.

3.1.2 Szélességi bejárás

Két alapvető bejárási algoritmus létezik a szélességi és a mélységi bejárás. A két bejárás csak a meglátogatott elemek sorrendjében tér el egymástól. Ha a legrövidebb utat keressük egy súlyozatlan gráfban, akkor a szélességi bejárást kell használunk.

Az algoritmus célja, hogy a gráf minden egyes csúcsát pontosan egyszer járja be. Ehhez két logikai értékű tömböt használunk fel. Az egyik tömbben azt tartjuk nyilván, hogy már mely csúcsokat dolgoztuk fel. A másik tömbben pedig azokat, amelyeket már felfedeztük.

Az eljárás mindig egy kezdő csúcsból indul ki. Ezt a csúcsot felvesszük a feldolgozandó csúcsok sorába. A továbbiakban mindig a sor első elemét dolgozzuk fel, míg ki nem ürül a sor. A feldolgozás első lépése, hogy a csúcsot megjelöljük feldolgozottként. Ezután minden belőle kiinduló él menti csúcsot, ha az még nem volt felfedezve, felfedezetté teszünk és beszúrjuk őket a feldolgozandó csúcsok sorának végre. Ha kiürült a sor akkor bejártuk a gráfot, legalábbis annak a kiinduló csúccsal egybefüggő részét. Fontos, hogy az algoritmus futása előtt mindkét tömb hamis értékekkel legyen feltöltve.

```
void bfs(graph *g, int start, bool processed[],
          bool discovered[], int parent[]) {
    std::queue<int> q; /* sor a feldolgozandó csúcsokkal */
    int v; /* a jelenlegi csúcs */
    int i; /* számláló */

    q.push(start);
    discovered[start] = true;

    while (q.empty() == false) {
        v = q.front();
        q.pop();
        process_vertex(v);
        processed[v] = true;
        for (i = 0; i < g->degree[v]; i++)
            if (valid_edge(g->edges[v][i]) == true) {
                if (discovered[g->edges[v][i]] == false) {
                    q.push(g->edges[v][i]);
                    discovered[g->edges[v][i]] = true;
                    parent[g->edges[v][i]] = v;
                }
                if (processed[g->edges[v][i]] == false)
                    process_edge(v, g->edges[v][i]);
            }
    }
}
```

A graph adatszerkezet a korábban leírt szomszédsági lista mátrixban reprezentációt használja. Ennek lecseréléséhez mindössze a szomszédos csúcsok bejárásának algoritmusát kell átírnunk az új reprezentációnak megfelelőre.

Ennek az algoritmusnak megvan az az előnye, hogy nagyon rugalmas. A valid_edge, process_edge, process_vertex metódusoktól függ, hogy mit teszünk a bejárt élekkel és csúcsokkal, sőt akár ki is hagyhatunk bizonyos éleket.

A parent tömb bár nem feltétlen szükséges a bejáráshoz, mégis gyakran hasznos lehet. Segítségével megmondhatjuk a feldolgozás után, hogy az egyes csúcsokhoz mely csúcsokon keresztül jutottunk el. Ezt a tömböt felhasználva megadhatjuk a legrövidebb utat egy költségmentes gráfban.

3.1.3 Dijkstra-algoritmus[4]

A Dijkstra-algoritmus célja a legrövidebb utak megtalálása egy súlyozott gráfban. Azonban megvan az a megkötése, hogy a gráfban nem lehetnek negatív súlyú élek. Az algoritmus végrehajtása során egy kiinduló s csúcsból megkeresi az összes többi csúcsba vezető legrövidebb utat, így a kívánt csúcsba vezetőt is.

Tegyük fel, hogy egy G gráfban a legrövidebb út s és t csúcs között átmegy egy x csúcson. Ebben az esetben ennek az útvonalnak nyilvánvalóan tartalmaznia kell az s és x közötti legrövidebb utat, hisz ha nem így lenne rövidíthetnénk az s és t közötti utat is. Tehát meg kell határoznunk a legrövidebb utat s és x között, hogy megkaphassuk a legrövidebb utat s és t között.

Az algoritmus során egymás után határozzuk meg a legrövidebb utakat s és a többi csúcs között. Egy csúcsból önmagába jutás költsége nulla. Így mindig van egy csúcs, amelynek ismerjük az odajutás minimális költségét, ez pedig a kezdeti csúcs. Ezután a k kiválasztott csúcsnak – egy olyan csúcs, amelyet még nem vizsgáltunk meg, de már ismerjük az odavezető legrövidebb utat – végignézzük az összes kimenő élét. Ha egy él a c csúcsba mutat, akkor megvizsgáljuk, hogy a kezdő csúcsból a k csúcsba jutás költsége plusz a k csúcsból a c-be mutató él költsége kisebb-e, mint a kezdő csúcsból a c csúcsba jutás költsége? Ha igen akkor találtunk egy jobb utat, ezt fel is jegyezzük. Miután az összes kimenő élt végignéztük, a következő csúcs kiválasztása úgy történik, hogy végignézzük az összes olyan csúcsot, amelyet még nem választottunk ki, és kiválasztjuk közülük azt, amelybe a legolcsóbban eljuthatunk a kezdő csúcsból. Az ebbe a csúcsba jutás költsége ugyanis már optimális lesz.

A következő példakódban a 3.1.1 fejezetben látható adatszerkezet súllyal kiegészített változatát használom. A metódus paraméterül várja a gráfot, a csúcsot, amelyből a távolságokra vagyunk kíváncsiak, illetve a distance és a parent tömböket, amelyeket feltölt a minimális költségekkel és az oda vezető úttal. Az utat visszafele megkaphatjuk a parent tömbből egy egyszerű algoritmus segítségével.

4 The Algorithm Design Manual, 6.3.1 fejezet.

```
void dijkstra(graph *g, int start, int distance[],int parent[]){
    bool processed[g->nvertices];
    int v, w;
    for (int i = 0; i < g->nvertices; ++i) {
        processed[i] = false;
        distance[i] = (unsigned) (-1) >> 1;
        parent[i] = -1;
    }
    distance[start] = 0;
    v = start;
    while (!processed[v]) {
        processed[v] = true;
        for (int p = 0, weight; p < g->degree[v]; ++p) {
            w = g->edges[v][p];
            weight = g->weight[v][p];
            if (distance[w] > (distance[v] + weight)) {
                distance[w] = distance[v] + weight;
                parent[w] = v;
            }
        }
        v = 0;
        for (int i = 0, dist = (unsigned) (-1) >> 1;
                i < g->nvertices; ++i)
            if (!processed[i] && dist > distance[i]) {
                dist = distance[i];
                v = i;
            }
    }
}
```

Az (unsigned) (-1) >> 1 egy kellően nagy érték megadásához kell. Ha ennél nagyobb értékek is előfordulhatnak a gráfban, akkor nagyobb szám típusokat kell használnunk.

3.2 Szerkesztési távolság meghatározása

Érdekes problémát jelentenek azok a feladatok, amelyekben néhány egyszerű szabály felhasználásával, egy szóból egy másikba kell minimális költséggel eljutnunk. Ezek a szabályok pedig:

- Beszúrhatunk egy betűt a szóba.
- Törölhetünk egy betűt a szóból.
- Kicserélhetünk egy betűt egy másik betűre.

A problémára könnyen találhatunk egy visszalépéses algoritmust, melyet például rekurzióval egyszerűen megvalósíthatunk. Azonban az ilyen megoldások rettenetesen teljesítenek, hisz végrehajtási idejük a szavak hosszával exponenciálisan nő.

A rekurzív megoldások tanulmányozásával azonban rájöhetünk arra, hogy azért futnak ezek a megoldások olyan lassan, mert ugyanazt az állapotot újra és újra kiértékelik. Ezt felismerve egy sokkal jobb megoldáshoz juthatunk. Ennek a technikának a neve dinamikus programozás. A lényege az, hogy a problémákat részproblémákra bontjuk, az egyes részproblémák megoldását eltároljuk, és az újabb problémák megoldását a korábban megoldottakra vezetjük vissza.

Mit jelent ez az esetünkben? A részproblémákat a szavak prefixei jelentik. Az ezek közötti átmenetek költségét kell meghatároznunk. A részproblémákat a továbbiakban az első és második részszó hosszával fogom jelölni. Ha az első szó i hosszú prefixéből a második szó j hosszú prefixébe jutásának a költségét akarjuk megkapni (i, j), akkor három korábbi állapotra lesz szükségünk. Az (i-1, j) ebből törléssel, az (i-1, j-1) ebből betűcserével és az (i, j-1), amiből beszúrással kaphatjuk meg a kívánt állapotot. Ekkor csak ki kell számítanunk, hogy a három művelet költsége plusz a kiinduló állapotokba jutás költsége közül melyik a legolcsóbb, és ezt az értéket el kell tárolni. Ezt addig folytatjuk, amíg a szavak végére nem érünk. Fontos, hogy a szavakat ne az egy hosszú, hanem a nulla hosszúságú prefixeiktől kezdve vizsgáljuk. A szó eleji beszúrásokat csak így tudjuk kezelni.

```
int getDistance(char a[], char b[]) {
    int lenA = strlen(a), lenB = strlen(b), minCost;
    // itt tárolom a részszavak költségét
    int costs[lenA + 1][lenB + 1];
    // a nulla hosszú részszavak költségé egyszerű
    for (int i = 0; i ≤ lenA; ++i)
        costs[i][0] = i;    // törlés
    for (int i = 0; i ≤ lenB; ++i)
        costs[0][i] = i;    // beszúrás
    for (int i = 1; i ≤ lenA; ++i)
        for (int j = 1; j ≤ lenB; ++j)
            if (a[i - 1] == b[j - 1])
                // nincs szükség beavatkozásra
                costs[i][j] = costs[i - 1][j - 1];
            else {
                minCost = costs[i - 1][j - 1];
                minCost = MIN(minCost, costs[i][j - 1]);
                minCost = MIN(minCost, costs[i - 1][j]);
                costs[i][j] = minCost + 1;
            }
    return costs[lenA][lenB];
}
```

A feladatot bonyolítani lehet úgy, ha az egyes műveleteknek akár a módosítandó betűtől függően is más költségük van. De ez nem befolyásolja az algoritmus alapját, csak az egyes lépések költségszámítását teszik bonyolultabbá.

Ha az optimális megoldáshoz szükséges lépéssorozatra is szükségünk van, akkor fel kell vennünk még egy tömböt a költségek mellé, ahova az oda jutás műveletét tároljuk el. Így visszafelé haladva egy egyszerű rekurzív programmal megkaphatjuk a lépéssorozatot.

Ennek az algoritmusnak további optimalizálási potenciálja van. Például a memória igényt csökkenthetjük, mivel nincs szükségünk a költségek mátrixára, mindig csak egy sor értékeivel számolunk. Azonban a feladatok többségében ilyen optimalizálásokra nincs szükség.

3.3 Leghosszabb közös részsorozatok meghatározása

A feladat az, hogy találjuk meg két szó között a leghosszabb közös részsorozatot. Ez a probléma rokonságban áll az előbbi fejezeteben található szerkesztési távolság megtalálásával. Gondoljuk ugyanis végig, egy közös részszót úgy kaphatunk meg, ha az egyik szóból kitöröljük a másik szóban nem megtalálható betűket, és beszúrjuk azokat, amelyek a másik szóban megtalálhatóak, de az egyikben nem. Ekkor ugyanis azok a betűk, amelyeket nem töröltünk ki, vagy nem szúrtunk be, pontosan a kívánt leghosszabb közös részsorozatot fogják alkotni. Tehát az algoritmusokban annyi a különbség, hogy ebben az esetben megtiltjuk a betűcserét és csak a karakterek beszúrását és törlését engedjük meg. Persze a költségek tömbjéből máshogy történik a megoldás kinyerése is.

A költségek tömbjének jobb alsó sarkában ugyanis a minimális beszúrások és törlések számát találjuk meg. De nekünk nem erre van szükségünk, hanem éppen a meg nem változtatott betűk számára. Az A és B közötti leghosszabb közös részszavaknak (továbbiakban LKR) megvan az a tulajdonságuk, hogy az A és LKR közötti távolság plusz az LKR és B közötti távolság megegyezik az A és B közötti távolsággal, ahol az A és LKR közötti távolság megegyezik a törlések számával, és az LKR és B közötti távolság pedig a beszúrások számával.

Tehát A és B távolsága = |A| - |LKR| (ennyi törlés van benne ugyanis) + |B| - |LKR| (ennyi beszúrás van benne). Ebből kifejezhetjük az LKR hosszát. |LKR| = (|A| + |B| - A és B távolsága) / 2

```
#include <string.h>
#define KEEP   0
#define DELETE 1
#define INSERT 2

int getLCS(char a[], char b[], char LCS[]) {
    int lenA = strlen(a); // az első szó hossza
    int lenB = strlen(b); // a második szó hossza
    int lenLCS;           // a leghosszabb közös részszó hossza
    int costs[lenA + 1][lenB + 1]; // a részszavak költsége
    char op[lenA + 1][lenB + 1];   // az optimális műveletek

    for (int i = 0; i ≤ lenA; ++i) {
        costs[i][0] = i;
        op[i][0] = DELETE;
    }
    for (int i = 0; i ≤ lenB; ++i) {
        costs[0][i] = i;
        op[0][i] = INSERT;
    }
    for (int i = 1; i ≤ lenA; ++i)
        for (int j = 1; j ≤ lenB; ++j)
            if (a[i - 1] == b[j - 1]) {
                costs[i][j] = costs[i - 1][j - 1];
                op[i][j] = KEEP;
            } else if (costs[i][j - 1] < costs[i - 1][j]) {
                costs[i][j] = costs[i][j - 1] + 1;
                op[i][j] = INSERT;
            } else {
                costs[i][j] = costs[i - 1][j] + 1;
                op[i][j] = DELETE;
            }

    lenLCS = (lenA + lenB - costs[lenA][lenB]) / 2;
```

Ez a függvény a paraméterül kapott a és b szó leghosszabb közös részszavát határozza meg, és helyezi el a szintén paraméterül kapott LCS tömbbe, illetve visszaadja a részszó hosszát. Az op tömbben tárolom el az optimális műveleteket. Erre azért van szükség, hogy könnyen felépíthessem a közös részt. Ha csak a hosszára vagyok kíváncsi akkor erre nincs szükség.

```
    LCS[lenLCS] = '\0';
    for (int i = lenA, j = lenB, len = lenLCS - 1; len >= 0;) {
        if (op[i][j] == KEEP) {
            LCS[len--] = a[i - 1];
            --i;
            --j;
        } else if (op[i][j] == DELETE) {
            --i;
        } else
            --j;
    }
    return lenLCS;
}
```

Ez itt magának a részszónak a meghatározása. Visszafelé haladok a szavak eleje felé a műveleteknek megfelelő irányban.

4 Hatékonyság növelés

A versenyeken nem elég csupán egy a helyes eredményt meghatározó algoritmust megtalálni. A versenyeken mindig vannak időbeli és memóriabeli korlátok, ezek mértéke pedig versenyenként és feladatonként változik. Ezért fontos megtanulni megbecsülni az egyes algoritmusok várható futási idejét és memória igényét, amennyiben pedig szükséges optimalizálni a kódot. A feladatok szövegében gyakran megtalálható, hogy mekkora adatmennyiségre számíthatunk. Teljesen más megközelítést igényelhet ugyanaz a feladat, ha maximum ötven elemen kell számításokat végeznünk, mintha százezren.

Fontos azonban megjegyezni, hogy a kód optimalizálását több okból se szabad túlzásba vinni. Egyrészt ez a versenyeken értékes időt visz el. Másrészt elbonyolíthatja a kódot, ez pedig elkerülhetetlenül hibákhoz vezet. Nem érdemes egy algoritmus kódjában alacsony szintű optimalizálásokat végezni. Ezek ugyanis csak nagyon ritkán adnak annyi plusz sebességet, amennyivel az időlimit alá kerülhetünk. Lényeges gyorsulást csak az algoritmus teljes lecserélésével érhetünk el. Ezek gyakran csak nagy adatmennyiség esetén válnak kedvezőbbé, viszont akkor sokkal gyorsabbá válnak.

Ezekből is láthatjuk, hogy ha már egy kész megoldás beadása esetén szembesülünk az időtúllépés hibával, akkor azzal rengeteg időt vesztünk, hisz valószínűleg teljesen újra kell írnunk a programot. Ezért már a program tervezése során nagy gondot kell fordítanunk a hatékonyságra.

A hatékonyság növelése és a programozási idő csökkentésében az egyik legfontosabb dolog, hogy lehetőleg minél kevesebb algoritmust kelljen nekünk megírni. Válasszunk olyan programozási nyelvet például C++ vagy Java, ahol a nyelv standard könyvtáraiban számos algoritmus már készen megtalálható. Ilyen szempontból a C nyelv nem túl szerencsés. Ezeknek az algoritmusoknak az a két nagy előnyük, hogy nem kell őket tesztelnünk, illetve valószínűleg úgyis gyorsabbak lesznek, mint bármilyen általunk írt algoritmus.

Gyorsabb a programokat alkothatunk azzal is, ha a feladatok kívánt kimenetét

gondosan ellenőrizzük. Gyakran a megoldásban kevesebb információt kell visszaadnunk, mint az a feladatból következne. Erre gyakori példát adnak azok a feladatok, ahol valamilyen útvonal meghatározása után nem az egész útvonalat kell visszaadnunk, hanem csak az út idejét vagy hosszát. Például a Robot[5] feladatban. Ilyenkor mindig gyanakodhatunk arra, hogy van olyan algoritmus, amely csupán a kért információt adja vissza, de magáról az útról nem mond többet. Az ilyen algoritmusok pedig általában jóval gyorsabbak.

Lehetőleg mindig kerüljük a visszalépéses algoritmusokat is. Ezek ugyanis általában rettenetesen lassan futnak. Amikor csak lehetőségünk van eliminájuk az azonos állapotok többszöri kiértékelését. Erre nyújtanak egy módszert a dinamikus programozási technikák. Tanulmányozzuk ezeket, és használjuk őket bátran.

5 http://uva.onlinejudge.org/external/3/314.html

5 Feladatmegoldás lépésről lépésre

Az első feladat legyen egy egyszerű kétszemélyes játék.

5.1 Bachet játéka[6]

A Bachet-féle játékot valószínűleg mindenki ismeri, legfeljebb más néven. Kezdetben *n* kő van az asztalon. Két játékos játszik (Stan és Ollie), akik felváltva lépnek. Mindig Stan kezd. Egy lépés abból áll, hogy elveszünk legalább egy, legfeljebb *k* követ az asztalról. Az nyer, aki az utolsó követ veszi el.

Most ennek a játéknak egy speciális változatát tekintjük. Az egy lépésben elvehető kövek száma csak egy *m* számból álló halmaznak egy eleme lehet. Az *m* szám között mindig ott van az 1, így a játék soha nem áll meg, amíg az utolsó követ is el nem vettük.

Input

A bemenet számos sorból áll. Minden sor egy játékot ír le, pozitív számok sorozatával. Az első szám *n* ($n \leq 1000000$), a táblán lévő kövek száma; a második szám *m* ($m \leq 10$), amely megadja a további számok darabszámát; a sor utolsó *m* darab száma az egy lépésben elvehető kövek számát határozza meg.

Output

A bemenet minden egyes sorára egy sort kell a kimenetre írni, amely vagy a „Stan wins”, vagy az „Ollie wins” szöveget tartalmazza, feltételezve, hogy mindketten tökéletesen játszanak.

6 https://it.inf.unideb.hu/honlap/acm/10404
http://acm.uva.es/p/v104/10404.html

Példa input

```
20 3 1 3 8
21 3 1 3 8
22 3 1 3 8
23 3 1 3 8
1000000 10 1 23 38 11 7 5 4 8 3 13
999996 10 1 23 38 11 7 5 4 8 3 13
```

Példa output

```
Stan wins
Stan wins
Ollie wins
Stan wins
Stan wins
Ollie wins
```

Az első lépés a feladat típusának felismerése. Ez ebben az esetben egyszerű feladat. Egy kétszemélyes játék győztesét kell megtalálnunk.

A második lépés során ellenőrizzük a kezelendő adatmennyiséget: $n \leq 1000000$, $m \leq 10$. Ezek alapján nincs szükség speciális algoritmusra, a teljes állapotteret a memóriában tarthatjuk.

A megoldás algoritmusa: A játék aktuális állapotát az írja le, hogy még hány kő van az asztalon és, hogy ki következik. Mivel a lehetséges állapotok száma csekély, ezért lehetőségünk van rá, hogy minden állapot jóságát a memóriában tarthassuk. Ráadásul mindössze egyetlen végállapot van. Ilyen esetben hatékony megközelítés lehet, ha visszafele indulunk el.

Ha már nincs kő az asztalon, és mi következünk, az azt jelenti, hogy elvesztettük a játékot. Tehát ez egy vesztes állapot. Minden olyan állapot, amelyből léphetünk vesztes állapotba az nyertes állapot, mivel ezekben az esetekben a másik félnek már nem lesz lehetősége nyertes lépésre. Azok az állapotok, amelyekből pedig csak és kizárólag nyertes állapotokba léphetünk vesztes állapotok, mivel ezekből nincs olyan lehetséges lépés, amellyel az ellenfelet vesztes állapotba kényszeríthetnénk.

Tehát nullától kiindulva meghatározhatjuk minden állapot jóságát. Majd megvizsgáljuk, hogy az n köves állapot nyertes vagy vesztes állapot.

Ennek megfelelően pedig kiírhatjuk, hogy Stan vagy Ollie nyert.

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

#define MAXM 10
#define MAXN 1000000

int main(int argc, char** argv) {
    int n, m, s[MAXM];
    char *r = (char*) malloc(sizeof (char) * (MAXN + 1));
    while (scanf("%d %d", &n, &m) != EOF) {
        for (int i = 0; i < m; ++i)
            scanf("%d", &s[i]);
        memset(r, 0, sizeof (char) * (n + 1));
        for (int i = 0; i < n; ++i) {
            if (r[i] == 0) {
                for (int o = 0; o < m; ++o) {
                    if (s[o] + i ≤ n) {
                        r[s[o] + i] = 1;
                    }
                }
            }
        }
        printf(r[n] == 1 ? "Stan wins\n" : "Ollie wins\n");
    }
    free(r);
    return (EXIT_SUCCESS);
}
```

Megjegyzések a feladathoz:

- Az r tömbben tárolom, hogy az i darab kő az asztalon az épp következő játékos számára kedvező (egyes érték), vagy kedvezőtlen (nullás érték).
- Dinamikusan foglalom a memóriát r számára, mivel még char típus esetén is majdnem egy megabájt memóriára van szüksége az egymillió elemnek. Ha csak szimplán egy lokális tömbként deklarálnám a változót, akkor az a stackben tárolódna. A versenyeken nem szoktak ekkora helyet lehetővé tenni a stackben, így hibával leállna a program.
- Az egyszerűség kedvéért nyolc bitet használok fel az állapotok tárolására. Tulajdonképpen egy bit is elég lenne rá, de teljesítmény tekintetében szinte semmit se változtatna.

5.2 Euklidesz játéka[7]

Két játékos játszik: Stan és Ollie. Két természetes számmal kezdenek. Stan, a kezdőjátékos a két szám közül a nagyobbikból kivonja a kisebbik bármely pozitív többszörösét, feltéve, hogy az eredmény nemnegatív. Ezután Ollie, a másik játékos ugyanezt teszi az eredményül kapott két számmal, majd ismét Stan, és így tovább felváltva mindaddig, amíg az egyik játékos a nagyobb számból kivonva a kisebbiknek egy többszörösét nullát nem kap, és ezáltal nyer. A játékosok kezdhetnek például a (25,7) számpárral:

25 7

11 7

4 7

4 3

1 3

1 0

A győztes Stan.

Input

A bemenet számos sorból áll. Minden sor két pozitív egész számot tartalmaz, amelyek a játék kiinduló számpárját adják meg. Mindig Stan kezd. A bemenet utolsó sora két nullából áll, és nem kell feldolgozni.

Output

A bemenet minden sorára egy sort kell a kimenetre írni: ha Stan nyer, akkor a `Stan wins`, ha Ollie, akkor az `Ollie wins` szöveget, feltételezve, hogy mindketten tökéletesen játszanak.

7 https://it.inf.unideb.hu/honlap/acm/10368
http://acm.uva.es/p/v103/10368.html

Példa input

```
34 12
15 24
0 0
```

Példa output

```
Stan wins
Ollie wins
```

A feladat szövegét elolvasva feltűnő lehet, hogy nem szerepel benne semmilyen megkötés az inputként megkapott két számra. Emiatt feltételezhetjük, hogy az olyan megoldások melyek a teljes állapotteret megpróbálják letárolni sikertelenek lesznek. De a problémát szemlélve észrevehetjük, hogy erre nincs is szükség.

A feladat neve beszédes, mégpedig az euklideszi algoritmusra utal, amellyel két szám legnagyobb közös osztóját határozhatjuk meg. A feladat során is ezt fogja megállapítani a két játékos, tehát a végső két szám független a játék menetétől.

Az egyes állapotokat két csoportra oszthatjuk. Azokra az állapotokra, amelyekből csak egyetlen lépés lehetséges, és azokra, amelyekből több. A győztes kilétét az dönti el, hogy páros vagy páratlan lépésszámban érjük el a végállapotot. Amíg olyan állapotok jönnek egymás után, amelyekből csak egy lépés lehetséges, addig figyelnünk kell ezt a páros-páratlan szabályt. Ezt addig kell folytatnunk, amíg nem találunk egy olyan állapotot, amelyből több lehetséges továbblépési lehetőség is van. Ekkor az a játékos, amelyik soron következik megnyeri a játékot. Ugyanis az az állapot, amelyikhez úgy jutunk el, hogy elvesszük az összes lehetséges követ az vagy nyertes vagy vesztes állapot. Ha nyertes állapot, akkor a lehetséges maximális mennyiségtől egyel kisebb mennyiséget kell elvennünk. Ha vesztes akkor pedig a lehető legtöbbet kell elvennünk és meg is nyertük a játékot. Mivel nem kell megadnunk a nyeréshez szükséges lépéssorozatot ezért nem kell megadnunk a fenti két lépés közül melyik a helyes, illetve nem is kell tovább vizsgálnunk a játék menetét.

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

int main(int argc, char** argv) {
    int a, b, winner;
    scanf("%d %d", &a, &b);
    while (a != 0) {
        if (a > b)
            std::swap(a, b);
        winner = 1;
        while (true) {
            if (a == b || b / a > 1)
                break;
            winner = -winner;
            b = b % a;
            std::swap(a, b);
        }
        printf(winner == 1 ? "Stan wins\n" : "Ollie wins\n");
        scanf("%d %d", &a, &b);
    }
    return (EXIT_SUCCESS);
}
```

Megjegyzés a megoldáshoz:

A megoldás során feltettem, hogy az inputként kapott számok el fognak férni az int típusban, de használhattam volna nagyobb értékeket megengedő típust is.

5.3 Számlabirintus[8]

Tekintsünk egy számlabirintust, amelyet egy 0 és 9 közé eső számokat tartalmazó kétdimenziós tömbbel reprezentálunk, ahogy az alábbi példán látható. A labirintusban bármelyik ortogonális (azaz északi, déli, keleti és nyugati) irányban mozoghatunk. Ha a mátrix elemeit költségként értelmezzük, izgalmas feladat lehet megtalálni a labirintus adott belépési ponttól adott kilépési pontig történő bejárásának minimális költségét.

0	3	1	2	9
7	3	4	9	9
1	7	5	5	3
2	3	4	2	5

Feladat

A feladatod, hogy megtaláld egy adott N×M-es labirintus bal felső sarkából a jobb alsó sarkába történő eljutás minimális költségét, ahol 1 ≤ N,M ≤ 999. A fenti példában ez a költség 24.

Input

A bemenet számos labirintust tartalmaz. Az első sor egy pozitív egész számból áll, amely a labirintusok számát adja meg. Az egyes labirintusok a következőképpen vannak megadva: az első sor a labirintus sorainak számát (N), a második sor a labirintus oszlopainak számát (M), további N sor pedig a labirintus egyes soraiban álló értékeket adja meg egy-egy szóközzel elválasztva.

Output

Minden labirintus esetén egy sort kell a kimenetre írni, amely a kívánt minimális költséget tartalmazza.

8 http://uva.onlinejudge.org/external/9/929.html

Példa input

```
2
4
5
0 3 1 2 9
7 3 4 9 9
1 7 5 5 3
2 3 4 2 5
1
6
0 1 2 3 4 5
```

Példa output

```
24
15
```

Ebben a feladatban egy minimális költségű utat kell megkeresnünk. Az ilyen feladatoknál a térképet egy súlyozott élekkel rendelkező gráfnak tekintjük. Az optimális út megkeresése során pedig a szélességi kereséshez hasonló algoritmust használunk, csak az ilyen esetekben nem egy egyszerű sort, hanem egy prioritásos sort használunk. Az ilyen sorokban az elemek rendezetten helyezkednek el. Az egyes csúcsok prioritásának pedig az odajutás költségét tekintjük.

A prioritásos soroknak azonban van egy hátrányuk, az új elemek beszúrása során mindig rendeznünk kell a sort. Ez sok időt vehet igénybe a feladat megoldása során. Ebben az esetben viszont kihagyhatjuk a rendezéseket, ha megfelelően tartjuk nyilván a meglátogatandó csúcsokat. Erre azért van lehetőség mivel tudjuk, hogy a költségek csak nulla és kilenc között változhatnak.

A módszer pedig a következő: az egyes csúcsokhoz tartozó költségeknek csak a tízes maradékosztályát tároljuk. Az algoritmus során végig megyünk sorban a maradékosztályokon nullától kilencig és minden oda tartozó csúcsot feldolgozunk. Ilyen feldolgozás mellett minden elemet a költsége szerinti sorrendben dolgozunk fel. A megoldáshoz elég azt számon tartani, hogy hányszor fordultunk körbe a maradékosztályokon. Így a megoldás a körbefordulások számának tízszerese plusz az aktuális maradék értéke.

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

#define MAXRC 1000  // a maximális sor és oszlop szám
// a labirintus a költségekkel
char palya[MAXRC + 2][MAXRC + 2];
std::vector<int> sorok[10];         // a maradékosztályok sorai

// a lépések x és y koordinátáit egyetlen számban tárolom
// ez a metódus ebből a kódból visszaadja az x és y koordinátát
void getXY(int code, int &x, int &y) {
    x = code / MAXRC;
    y = code % MAXRC;
}

// a lépések x és y koordinátáit egyetlen számban tárolom
// ez a függvény az x és y koordinátákból meghatározza a kódot
int getCode(int x, int y) {
    return x * MAXRC + y;
}
```

Ez a módszer nem működne, ha a költség lehetne tíznél nagyobb is. Ekkor nem a megfelelő sorrendben dolgozná fel a csúcsokat az algoritmus.

```
// ha még nem jártam be az adott mezőt eltárolom a a bejárandó
// mezők sorába
void storeIfNeeded(int i, int x, int y) {
    int mod;
    if (palya[y][x] >= 0) {
        mod = (i + palya[y][x]) % 10;
        sorok[mod].push_back(getCode(x, y));
        palya[y][x] = -2;
    }
}

// a labirintust bejárhatatlan mezőkkel veszem körbe, hogy
// ne kelljen attól tartani, hogy kifutok a pályáról
void makeFrame(int n, int m) {
    memset(palya[0], -1, m + 2);
    memset(palya[n + 1], -1, m + 2);
    for (int i = 1; i < n + 1; ++i) {
        palya[i][0] = -1;
        palya[i][m + 1] = -1;
    }
}
```

A már bejárt mezőket úgy jelöltem a labirintusban, hogy az adott mező költségét mínusz egyre változtattam, a már felfedezettekét pedig mínusz kettőre. A pályát úgy

vettem fel, hogy az minden szélén egy mezővel szélesebb legyen a szükségesnél. A makeFrame metódusnak az a feladata, hogy a labirintus ezen széleit már meglátogatottá tegye. Ennek az az értelme, hogy így a feldolgozás közben nem kell vizsgálnom, hogy az adott lépéssel még a pályán maradok-e, hisz a meglátogatott mezőkből nem léphetek tovább.

```
int main(int argc, char** argv) {
    int n, N, M, megoldas, x, y, mod;
    char *c, row[2000];

    for (int i = 0; i < 10; ++i)
        sorok[i].reserve(MAXRC * MAXRC);

    scanf("%d\n", &n);
    for (int q = 0; q < n; ++q) {
        // feladatonkénti inicializáció
        scanf("%d\n%d\n", &N, &M);
        makeFrame(N, M);
        for (int i = 1; i < N + 1; ++i) {
            gets(row);
            c = row;
            for (int j = 1; j < M + 1; ++j, c += 2)
                palya[i][j] = *c - '0';
        }

        for (int i = 0; i < 10; ++i)
            sorok[i].clear();
        megoldas = 0;

        mod = palya[1][1] % 10;
        sorok[mod].push_back(getCode(1, 1));
```

Ez a program inicializációs része itt olvasom be az inputot és készítem el a labirintus reprezentációját.

```
        // megoldás
        while (true) {
            // a sorok végigjárása
            for (int i = 0; i < 10; ++i) {
                // a sor feldolgozása
                while (!sorok[i].empty()) {
                    getXY(sorok[i].back(), x, y);
                    sorok[i].pop_back();
                    if (x == M && y == N) {
                        megoldas = megoldas * 10 + i;
              // megvan a megoldás, három ciklusból való kiugrás
                        goto kesz;
                    }
                    palya[y][x] = -1;
                    storeIfNeeded(i, x - 1, y); // balra
                    storeIfNeeded(i, x + 1, y); // jobbra
                    storeIfNeeded(i, x, y - 1); // fel
                    storeIfNeeded(i, x, y + 1); // le
                }
            }
            ++megoldas;
        }
kesz:
        printf("%d\n", megoldas);
    }
    return (EXIT_SUCCESS);
}
```

Ez a megoldás érdemi része: számon tartom, hogy hányszor néztem végig a maradékosztályokat, és feldolgozom a felfedezett mezőket. Az itt szereplő goto nem túl szerencsés programozói eszköz, de mivel három ciklusból kell kilépni, ezért egyszerűbbé teszi a megoldást.

Megjegyzések a megoldáshoz:

- Mivel az std vektorának nem használjuk ki minden funkcióját még gyorsabb megoldást érhetünk el, ha saját tömbön alapuló vermet készítünk. Ezt könnyen megtehetjük, mivel előre ismerjük a maximális elemszámot. De ez tovább bonyolítja a programot, így ez nem ajánlott.

5.4 Szerencsejáték[9]

A szerencsejáték mindig is nagyon népszerű volt Kínában. Hiába volt tiltott az idők nagy részében, az emberek ennek ellenére játszottak Mah Jongot, Pai Gowot, Fan-Tant, Sic Bot és más játékokat titokban. Sanghaj az 1930-as években rengeteg illegális szerencsebarlangnak adott otthont, melyeket befolyásos bűnbandák irányították. Legtöbbjüket 1949-ben bezárták a kommunisták, biztonságosabbá téve a várost.

Ebben a feladatban egy kevésbé ismert játékkal az Ah Ce Emm-mel foglalkozunk. Ebben a játékban véletlen számú kavicsot kapsz. A célod pedig az, hogy elveszítsd az összeset. Ha sikerül megkapod a nyereményt. Többféle lehetőséged is van a kavicsok számának változtatására, de mindegyikért fizetned kell egy bizonyos összeget.

- A tűz: ha van legalább 11 kavicsod, akkor eldobhatsz pontosan 11 kavicsot, és x1-et kell fizetned.
- A sárkány: ha a köveid száma hárommal osztható, akkor harmadukat eldobhatod mindegyikért egyet fizetve. Tehát ha van tizenkét kavicsod, akkor ezzel a lépéssel nyolcra csökkentheted a számukat, és négyet kell fizetned.
- A sas: kérhetsz pontosan 7 új kavicsot x2 összegért.
- A bátorság: megduplázod a kavicsaid számát és kapsz még egyet, minden új kőért egyet kell fizetned. Tehát ha van három kavicsod, akkor ezzel a lépéssel hétre növelheted a számukat négy pénzért cserébe.

Az x1 és x2 értéke játékról játékra változik. Nem lehet több kavicsod, mint eredetileg volt: ha egy lépés az eredeti mennyiségük fölé emelné a kövek számát, akkor nem választhatod azt a lépést. A feladatod, hogy írj egy programot, ami a megadott kövek száma és az x1, x2 értékek alapján meghatározza az összes kavics elvesztésének minimális költségét.

9 http://acmicpc-live-archive.uva.es/nuevoportal/data/problem.php?p=3182

Input

Minden egyes sor három egész számot tartalmaz, és más tesztesethez tartozik. Az első szám $1 \leq n \leq 200000$ a kövek kezdeti száma. A második és a harmadik szám $1 \leq x1, x2 \leq 500000$ a tűz és sas költsége.

A bementet egy három nullát tartalmazó sor zárja.

Output

Minden egyes tesztesethez ki kell írnod egy egész számot új sorba, az összes kő elvesztésének minimális költségét. Ha lehetetlen minden követ elveszteni, akkor írd ki, hogy `Impossible' (idézőjelek nélkül).

Példa input

```
33 122 200
1000 100 200
2 10 10
0 0 0
```

Példa output

```
255
1953
Impossible
```

Ezt a feladatot egyszerűen megoldhatnánk a szabályok leprogramozásával egy visszalépéses rekurzív algoritmussal. Azonban az input tanulmányozásával kiderül, hogy ez nem járható lehetőség, mivel a megoldáshoz szükséges lépések elvi száma túlságosan nagy. Át kell tehát fogalmaznunk a feladatot.

A kövek aktuális száma meghatároz egy állapotot. Az állapotok között akkor vezet él, ha a játék szabályai szerint az egyik állapotból a másikba léphetünk. Minden egyes élnek költsége van, ennek a mértékét is a szabályok határozzák meg. A feladatunk tehát, hogy megtaláljuk az n követ tartalmazó csúcsból a legkisebb költségű utat a nulla követ tartalmazó csúcsba. Ennek az útnak a költségére lesz szükségünk.

A feladatot megoldhatjuk úgy, ha visszafelé játsszuk a játékot. Vagyis a nulla követ tartalmazó állapotból indulunk ki és a szabályokat visszafele alkalmazva

megállapítjuk, hogy az egyes állapotokból mekkora költséggel érhetünk vissza az üres állapotba.

Az így megfordított szabályok:

- A tűz: ha az aktuális kavicsszám + 11 ≤ n, akkor felvehetsz pontosan 11 kavicsot x1-et fizetve.
- A sárkány: ha a köveid száma kettővel osztható, és az aktuális köveid száma nem nagyobb mint az n kétharmada, akkor másfélszer annyi köved lehet, a köveid árának feléért. Tehát ha van tizenkét kavicsod, akkor ezzel a lépéssel tizennyolcra növelheted a számukat hatot fizetve.
- A sas: ha van legalább 7 kavicsod, akkor eldobhatsz pontosan 7 kavicsot x2 összegért.
- A bátorság: a kavicsaid számából egyet elvesz, majd a maradékot elfelezi. Csak páratlan számú kő esetén alkalmazható. Legalább három kő kell a használatához. Minden eldobott kőért egyet kell fizetned. Tehát ha van három kavicsod, akkor ezzel a lépéssel egyre csökkentheted a számukat kettőt fizetve.

Ezekbe a szabályokba már beleépítettem azt a szabályt is, hogy a kövek száma nem haladhatja meg a kezdeti értéket.

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

#define MAXN 200000
#define MAXX 500000

int main(int argc, char** argv) {
    int n, x1, x2, costs[MAXN + 1], cost, pebbles;
    std::priority_queue<std::pair<int, int> > moves;
    scanf("%d %d %d", &n, &x1, &x2);
    while (n > 0) {
        memset(costs, 0, (n + 1) * sizeof (int));
        moves = std::priority_queue<std::pair<int, int> >();
        // tűz
        if (n >= 11)
            moves.push(std::make_pair(-x1, 11));
```

Ez a program inicializációs része. A megoldás egyetlen lépéssel kezdődhet a tűzzel, mivel csak ezzel lehet minden követ eldobni.

```
        while (!moves.empty()) {
            cost = -moves.top().first;
            pebbles = moves.top().second;
            moves.pop();
            if (costs[pebbles] != 0)
                continue;
            costs[pebbles] = cost;
            if (pebbles == n) {
                break;
            }
            // sas
            if (pebbles >= 7)
                moves.push(std::make_pair(
                    -(cost + x2),
                    pebbles - 7)
                );
            // tűz
            if (pebbles + 11 ≤ n)
                moves.push(std::make_pair(-(cost + x1),
                        pebbles + 11));
            if (pebbles % 2 == 0) {
                // sárkány
                if (pebbles + pebbles / 2 ≤ n)
                    moves.push(std::make_pair(
                        -(cost + pebbles / 2),
                        pebbles + pebbles / 2)
                    );
            } else
                // bátorság
                if (pebbles > 1)
                    moves.push(std::make_pair(
                        -(cost + (pebbles - 1) / 2 + 1),
                        (pebbles - 1) / 2)
                    );
        }
```

Megjegyzések a megoldáshoz:

- Ez a megoldás érdemi része. Az optimális útkereséshez std::priority_queue-t használok, melyben az elemek csökkenő sorrendben rendezettek. Az std::pair használatának az az előnye, hogy összehasonlíthatóak (először az első elemeket hasonlítja össze, majd ha ezek egyenlőek akkor a másodikat). Így ezt sem nekem kell definiálnom a rendezéshez.
- Mivel a std::priority_queue alapértelmezetten csökkenően rendez, ezért a

költségeket negatív előjellel kell tárolnom, erre figyelni kell.

- Nem használok processed tömböt, hanem az állapot költségét figyelem, ha már nem nulla akkor az azt jelenti, hogy abban az állapotban már jártam.

A megoldás kiíratása.

```
            if (costs[n] == 0)
                printf("Impossible\n");
            else
                printf("%d\n", costs[n]);
            scanf("%d %d %d", &n, &x1, &x2);
        }
        return (EXIT_SUCCESS);
    }
```

5.5 Szóláncok[10]

Hány lépésben lehet egy macskát kutyává változtatni? Négy lépésben megoldható:

cat – hat – hot – hog – dog

Egy szólánc szavak olyan sorozata, melyben csak kis különbségek vannak a szomszédos szavak között. Pontosabban:

- A két szó csak egyetlen betűben különbözik.
- Az egyik szóból előállítható a másik, ha két szomszédos betűt felcserélünk (pl. center – centre).

A feladatod, hogy írj egy programot, mely meghatározza a legolcsóbb szóláncot két megadott szó között. Minden műveletnek rögzített ára van: az input meghatározza a 't' és 'c' kicserélésének költségét, az 'e' és 'r' felcserélésének a költségét, ha ebben a sorrendben állnak, és így tovább. Egy szólánc költsége a benne található műveletek költségének az összege.

Input

Az input számos tesztesetet tartalmaz. Minden teszteset első sora két egész értéket tartalmaz: a szavak számát $2 \leq n \leq 4000$ és az ábécé betűinek számát $1 \leq m \leq 26$. A következő n sor egy-egy szót tartalmaz. Ez az n darab szó azonos hosszú – ez a hossz 1 és 10 között változhat – és csak az angol ábécé első m karakterét tartalmazza.

Az ezt követő m sor leírja a karakterek más karakterekkel való lecserélésének a költségét. Mind az m sor m darab egészet tartalmaz 1 és 1000 között. A j-edik egész az i-edik sorban az i-edik karakter lecserélésének a költsége a j-edik karakterrel. (Például a második szám a negyedik sorban a 'd' 'b'-re cserélésének költsége.) Az átlóban lévő számok mind nullák.

A következő m sor a szomszédos betűk felcserélésének költségét tartalmazza. Mind az m sor m darab egészet tartalmaz 1 és 1000 között. A j-edik egész az i-edik

10 http://www.inf.unideb.hu/~mkosa/acm2005/acm.pdf

sorban az i-edik karakter felcserélésének a költsége a j-edik karakterrel a jobb oldalon. (Például a center centre-ré alakításának a költségét az ötödik sor tizennyolcadik egésze adja meg, míg a centre center-ré alakításának a költségét a tizennyolcadik sor ötödik egésze. Ez a két szám nem feltétlen egyezik meg.) Az átlóban lévő értékek mind nullák.

Az inputot egy két nullát tartalmazó sor zárja.

Output

Minden tesztesethez egy egész értéket kell kiírni egy külön sorba: a minimális költségét annak a láncnak, amely az első szóval kezdődik és az n-edikkel ér véget. A lánc minden szavának az input n szavából kell kikerülnie, de nem kell minden szót felhasználni. Ha nem létezik ilyen lánc akkor írd ki, hogy `Impossible' (idézőjelek nélkül).

Példa input

```
6 4
aba
baa
aab
bad
cab
cad
0 8 2 2
9 0 3 2
9 9 0 9
9 9 9 0
0 1 1 1
1 0 1 1
1 1 0 1
1 1 1 0
0 0
```

Példa output

```
5
```

Ezt a feladatot két részfeladatra lehet bontani. Először is meg kell tudni határozni az egyes szavak közti távolságot. Majd ezeket a távolságokat felhasználva meg kell keresnünk a legrövidebb utat az első és utolsó szó között.

```
 #include <stdlib.h>
 #include <stdio.h>
 #include <string.h>
 #define MAXN 4000   // a szavak maximális száma
 #define MAXM 26     // a betűk maximális száma
 #define MAXWL 10    // a maximális szóhossz
 #define IMPOSSIBLE 10000000
 #define MIN(a,b) ((a)>(b)?(b):(c))

 int changeCosts[MAXM][MAXM], swapCosts[MAXM][MAXM],
 dists[MAXN][MAXN];

 int getDist(char a[], char b[]) {
     bool changed = false;
     int cost = 0;
     for (char *c1 = a, *c2 = b; *c1 != '\0'; ++c1, ++c2) {
         if (*c1 != *c2) {
             if (!changed) {
                 changed = true;
                 if (*c1 == *(c2 + 1) && *(c1 + 1) == *c2) {
                     cost = swapCosts[*c1 - 'a'][*(c1 + 1) -
'a'];
                     ++c1;
                     ++c2;
                 } else
                     cost = changeCosts[*c1 - 'a'][*c2 - 'a'];
             } else
                 return IMPOSSIBLE;
         }
     }
     return cost;
 }
```

A getDist függvény meghatározza két szó közti távolságot. Ha egy művelettel nem lehet egyik szóból a másikba jutni, akkor pedig az IMPOSSIBLE értékkel tér vissza, amely nagyobb, mint a legdrágább szólánc elméleti költsége.

```
int main(int argc, char** argv) {
    char dictionary[MAXN][MAXWL + 1], visited[MAXN];
    int n, m, node, cost;
    scanf("%d %d\n", &n, &m);
    while (n != 0) {
        memset(visited, 0, sizeof (visited));
        for (int i = 0; i < n; ++i)
            gets(dictionary[i]);
        for (int i = 0; i < m; ++i)
            for (int j = 0; j < m; ++j)
                scanf("%d", &changeCosts[i][j]);
        for (int i = 0; i < m; ++i)
            for (int j = 0; j < m; ++j)
                scanf("%d", &swapCosts[i][j]);
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
               dists[i][j]=getDist(dictionary[i],dictionary[j]);
        node = 0;
```

Itt található a program inicializációs része. Itt határozom meg az egyes szavak távolságát egymástól.

```
        while (!visited[node]) {
            visited[node] = true;
            for (int i = 0; i < n; ++i) {
                cost = dists[0][node] + dists[node][i];
                if (dists[0][i] > cost)
                    dists[0][i] = cost;
            }
            node = 0;
            cost = IMPOSSIBLE;
            for (int i = 0; i < n; ++i) {
                if (!visited[i] && cost > dists[0][i]) {
                    cost = dists[0][i];
                    node = i;
                }
            }
            if (node == n - 1)
                break;
        }
        if (dists[0][n - 1] == IMPOSSIBLE)
            printf("Impossible.\n");
        else
            printf("%d\n", dists[0][n - 1]);
        scanf("%d %d\n", &n, &m);
    }
    return (EXIT_SUCCESS);
}
```

A legrövidebb út meghatározására egy átalakított Dijkstra-algoritmust használok.

5.6 Zipper[11]

Adott három szó. A feladatod annak a megállapítsa, hogy a harmadik szó létrehozható-e az első két szó betűinek kombinációjával. Az első két szót szabadon keverheted, de mindkettőnek az eredeti sorrendjében kell maradnia.

Például létrehozható-e a „tcraete" szó a „cat" és „tree" szavakból?

String A: cat

String B: tree

String C: tcraete

Mint azt láthatod, létrehozhatjuk a harmadik szót az első két szó betűinek felhasználásával. Második példának vegyük a „catrtee" létrehozását a „cat" és „tree" szavakból.

String A: cat

String B: tree

String C: catrtee

Végül vegyük észre, hogy lehetetlen a „cttaree" szót létrehozni a „cat" és „tree" szavakból.

Input

A bemenet első sora egy pozitív egész számot tartalmaz 1 és 1000 között. Ez határozza meg a következő adathalmazok számát. Minden adathalmaz feldolgozása azonos. Az adathalmazok a következő sorokban szerepelnek, egy adathalmaz soronként.

Minden egyes adathalmazhoz egy sor tartozik benne három szóval, melyeket egy szóköz választ el. Minden szó csak kis és nagy betűkből áll. A harmadik szó hossza mindig az első lét szó hosszának az összege. Az első két szónak 1 és 200 között lesz a hossza, a határokat is beleértve.

Output

Minden egyes adathalmazhoz írd ki:

11 http://acmicpc-live-archive.uva.es/nuevoportal/data/problem.php?p=3195

`Data set n: yes`

Ha a harmadik szó létrehozható az első két szóból, vagy

`Data set n: no`

Ha nem. Természetesen az n-et ki kell cserélni az adathalmaz sorszámával, ahogy azt a lenti példa outputban láthatod.

Példa input

```
3
cat tree tcraete
cat tree catrtee
cat tree cttaree
```

Példa output

```
Data set 1: yes
Data set 2: yes
Data set 3: no
```

A feladat megoldása során nyilvánvalóan nem próbálhatjuk végig az első két szó minden összefésülését, hisz ebben az esetben a megoldás ideje ezen szavak hosszával exponenciálisan nőne. A megoldás kulcsa az, hogy a feladatot kisebb feladatokra bontsuk, és ezeket a kisebb feladatokat oldjuk meg. Egy lehetséges felbontása a problémának:

Vizsgáljuk meg, hogy az első két szó helyett azoknak csak a prefixeit felhasználva a harmadik szó milyen hosszú prefixét tudjuk előállítani?

Nulla hosszúságú előtagokból csak a kívánt szó nulla hosszúságú előtagját tudjuk előállítani.

Az (i, j) hosszúságú prefixek felhasználásával előállítható szórész hosszát visszavezethetjük az (i - 1, j) és (i, j - 1) hosszúságú prefixekből előállíthatóakra. Meg kell határoznunk, hogy melyik esetből továbblépve kapunk nagyobb értéket. Az (i - 1, j) állapot értéke legyen K. Ebből az állapotból K + 1 értékkel léphetük tovább, ha az első szó i-edik karaktere megegyezik a kívánt szó K + 1-edik karakterével. Ugyanis ekkor a harmadik szó felépítéséhez felhasználhatjuk ezt a betűt. Ha ez nem teljesül

akkor pedig K-val léphetünk tovább. Az (i, j - 1) állapot értéke legyen L. Ebből az állapotból L + 1 értékkel léphetük tovább, ha a második szó j-edik karaktere megegyezik a kívánt szó L + 1-edik karakterével. Ugyanis ekkor a harmadik szó felépítéséhez felhasználhatjuk ezt a betűt. Ha ez nem teljesül akkor pedig L-lel léphetünk tovább.

Ezeket a vizsgálatokat addig kell folytatnunk, míg el nem érjük a szavak végét. Ha a teljes szavak felhasználásával előállítható prefix hossza megegyezik a harmadik szó hosszával, akkor lehetséges az első két szóból előállítani a harmadikat. Ha kevesebb, akkor pedig lehetetlen.

```
 int main(int argc, char** argv) {
     int n, lenA, lenB, down, right;
     char a[201], b[201], c[401];
     int row[202];
     scanf("%d", &n);
     for (int q = 1; q <= n; ++q) {
         scanf("%s %s %s", a, b, c);
         lenA = strlen(a);
         lenB = strlen(b);
         row[0] = 0;
         for (int j = 1; j < lenB + 1; ++j)
             row[j] = row[j - 1] + (b[j - 1] == c[row[j - 1]]);
         for (int i = 0; i < lenA; ++i)
             for (int j = 0; j < lenB + 1; ++j) {
                 down = row[j] + (a[i] == c[row[j]]);
                 if (j > 0)
                     right = row[j-1] + (b[j-1] == c[row[j-1]]);
                 else
                     right = 0;
                 row[j] = MAX(down, right);
             }
         printf("Data set %d: %s\n", q, (row[lenB] == strlen(c) ?
"yes" : "no"));
     }
     return (EXIT_SUCCESS);
 }
```

Az algoritmus folyamán fontos, hogy a korábbi állapotok értékét megőrizzük, és ne számítsuk ki őket újra és újra. Ez ugyanis nagyon megnövelné a megoldás idejét.

Az állapotok eltárolása történhetne például egy mátrixban. Észrevehetjük azonban, hogy nincs szükség az összes állapot értékének megőrizésére. Mindig csak az aktuális állapot balra és fölfelé lévő szomszédjaira vagyunk kíváncsiak. Ezt

felhasználva elég csak egy sornyit tárolni a mátrixból és mindig felülírni az értékeket.

Az értékek meghatározásánál kihasználtam a C++ nyelv azon tulajdonságát, hogy egy logikai vizsgálat egyes értékkel tér vissza, ha igaz volt a feltétel, és nullával ha nem.

6 Összefoglalás

Mint azt láthattuk ennek a feladattípusnak a megoldása nem lehetetlen feladat. A problémát a hatékony algoritmusok megtalálása adja. Az MI algoritmusokat jól lehet alkalmazni bizonyos feladatok megoldása során, főleg ha minden egyes állapotot be kell járnunk a megoldásban. De a hagyományos mesterséges intelligencia algoritmusok – melyekkel az általános optimalizációs problémákat megoldhatjuk – a versenyeken tapasztalható feladatok többségében, az ottani adatmennyiségek és időkorlátok mellett nem alkalmazhatóak.

A sikeres feladatmegoldásnak tehát az a kulcsa, hogy felismerjük, hogy az adott feladat milyen szintű optimalizációt igényel, és ne is próbáljunk meg olyan algoritmusokat lekódolni, amelyek bár helyes eredménnyel járnának, de a verseny keretei között nem érnének véget. A visszalépéses algoritmusokat gyakorlatilag sohasem lehet eredményesen alkalmazni. Más megoldás típusokat kell keresni, mint például a dinamikus programozás. Ennek a szemléletnek a megtanulása a fontos.

Mint az élet oly sok más területén, itt is a gyakorlással lehet legkönnyebben sikereket elérni. Ha már magabiztosan ismerjük fel a feladat típusokat, akkor sokkal nagyobb eséllyel fogjuk tudni meghatározni a megfelelő megoldási módszert. Ehhez persze már elméleti ismeretekre is szükség van. Hisz hiába tudjuk, hogy a hagyományos algoritmus a probléma megoldásához elégtelen, ha nem tudunk alkalmas megoldást találni. Bár már ezzel is értékes kódolási és hibás megoldásért járó büntető időtől szabadulhatunk meg.

A legfontosabb dolog a hatékony algoritmusokhoz vezető módszereknek és gondolatmeneteknek a megismerése. Ugyanis lehetetlen minden egyes speciális esethez szükséges algoritmus változatot megtanulni. A cél csupán az, hogy a leggyakrabban alkalmazott algoritmusokat hibátlanul ismerjük, és ezekre építve a feladatban megjelenő sajátosságokhoz tudjuk alakítani a megoldást. Ezt csak sok gyakorlással lehet elérni.

7 Irodalomjegyzék

- Steven S. Skieana, Miguel Revilla
  Programming Challenges 9. és 10. fejezet.
- Jon Kleinberg & Éva Tardos
  Algorithm Design 3. fejezet.
- Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani
  Algorithms
- Steven S. Skiena
  The Algorithm Design Manual 5. fejezet.
- http://uva.onlinejudge.org/
  Uva Online Judge
- http://www.topcoder.com/tc
  Programming Contests, Software Development, and Employment Services
- https://it.inf.unideb.hu/honlap/acm
  ACM versenyfeladatok programozóknak

8 Köszönetnyilvánítás

Külön köszönetet szeretnék mondani témavezetőmnek, Kósa Márknak a szakdolgozat elkészítésében és a versenyzésben nyújtott segítségéért.

Köszönettel tartozom Pánovics Jánosnak az általa nyújtott segítségekért és, hogy lehetővé tette a versenyeken való indulást.

Meg szeretném köszönni minden csapattársaimnak a közös versenyzést. Sokat tanultam tőlük.