Korešpondenčný seminár z programovania
XXVIII. ročník, 2010/11
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

KSP finančne podporujú: MICROSTEP-MIS spol. s r.o.
Agentúra na podporu výskumu a vývoja

Vzorové riešenia 1. kola letnej časti

1. Žiarovky

vzorák písal Mio
(max. 10 bodov)

Nebudeme sa s tým babrať a prejdeme rovno k veci. Úloha bola ľahká, len si ju bolo treba správne preložiť. Tak poďme prekladať: Úlohou bolo zistiť, či daný obvod svieti. Keďže je obvod korektný, treba len zistiť, či je zapnutý vypínač. Na zapnutom vypínači nie je nič podozrivé, ale keď si lepšie pozrieme vypnutý vypínač, tak si všimneme, že obsahuje znak „/“ alebo „\“. Takýto znak však neobsahuje žiadna iná súčiastka. Teda preložená úloha znie: Zisti, či sa v schéme nachádza znak „/“ alebo „\“. Toto už neznie tak zložito.

Pomocou dvoch vnorených cyklov sa pozrieme na každý znak v schéme (prvým prechádzame riadky, druhým jednotlivé znaky v nich) a skontrolujeme, či to náhodou nie je jeden zo znakov „/“ alebo „\“. Ak áno, tak obvod nesvieti. Ak sme prešli všetky znaky a nenašli sme „/“ ani „\“, tak obvod svieti. Odpoveď vypíšeme a je to.

Časová zložitosť je lineárna od veľkosti vstupu: $O(r \cdot s)$. Rýchlejšie to nejde, pretože musíme skontrolovať každý znak.

Pamäťová zložitosť je konštantná ($O(1)$), pretože si pamätáme len konštantne veľa informácií.

Listing programu:

```
program Ziarovky;
var r, s, i, j: integer;
    z: char;
    ok: boolean;

begin
  ReadLn(r, s);
  ok := true;
  for i := 1 to r do
    for j := 1 to s do begin
      Read(z);
      if (z = '/') or (z = '\') then
        ok := false;
    end;
  if ok then WriteLn('svieti')
  else WriteLn('nesvieti');
end.
```


Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

2. Z ostrova niet úniku

opravoval Hermi
(max. 10 bodov)

Hneď na začiatku mnohí z vás odhalili veľkú fintu. Vďaka tomu, že sa príklad odohráva na mriežke a súper sa môže pohybovať všetkými ôsmimi smermi, v našom rade galeónov budeme používať len pravé uhly.

Najkratšou spojnicou medzi bodmi A a B je každá cesta, ktorá nerobí žiaden krok zlým smerom. Napríklad ak je A vľavo hore od B, potom akákoľvek postupnosť posunov dolu a doprava, ktorá začne v A a skončí v B, je rovnako dlhá a zároveň je to aj najkratšia spojnica. Na príklade to je jasne vidieť:

```
...........      ...........
..A*.......      ..A******..
...**......      ........*..
....**.....      ........*..
.....**....      ........*..
......**...      ........*..
.......*B..      ........B..
...........      ...........
```

Po krátkom zamyslení si uvedomíme, že stačí uzavrieť ostrov do obdĺžnika. Nájdeme najsevernejší, najjužnejší, najvýchodnejší a najzápadnejší bod ostrova. Potom každý z nich posunieme o jedno políčko smerom k jeho strane (severný na sever, atď.). Dostaneme štyri body, ktoré nám jednoznačne určujú obdĺžnik. Sláva, hurá! V $O(RS)$ čase a pamäti načítame mapu, nájdeme tieto body, doplníme obdĺžnik a vypíšeme.

Nikto ale 10 bodov nezískal. Ako je to možné? Dve slová: priveľa pamäte. Teraz prestaňte čítať a zamyslite sa nad riešením, ak vám prezradím, že beží v čase $O(RS)$ a pamäti $O(S)$. Pozrime sa naň:

Ako si môžeme dovoliť spracovávať riadok s najviac konštantným počtom nasledujúcich riadkov? Finta je jednoduchá – treba zabudnúť na našu fintu. Totiž ak by sme chceli vypísať obdĺžnik, ozaj to s menšou pamäťou spraviť nevieme.

Predpokladajme, že vypĺňame nejaký riadok mapy. Ak je ešte nad ostrovom, jednoducho ho necháme prázdny a teda vypíšeme nezmenený. Ak sa už ale v nasledujúcom riadku nejaká pevnina nachádza, musíme tu začať formovať galeóny. Nevieme, ako širokú radu by sme potrebovali na obdĺžnik, ale vieme, že teraz nám stačí, aby sme obkľúčili pevninu v nasledujúcom riadku. Preto formáciu roztiahneme po ľavú a pravú hranicu pevniny.

Teraz budeme spracúvať niekoľko riadkov, ktoré budú tvoriť ostrov. Tam opäť stačí pozrieť na nasledujúci riadok. Vieme, pokiaľ nám musí siahať nový ľavý a pravý kraj formácie, a navyše vieme, kde nám končí ľavá a pravá strana už vytvorenej časti, preto ju v prípade potreby rozšírime.[1] Ak už na ďalšom riadku pevnina nie je, ostrov skončil a uzavrieme formáciu. Takéto riešenie si v každom momente pamätá iba dva riadky mapy.

Ako poučenie z tohto príkladu by si mal každý zobrať, že aj keď nájdeme peknú fintu, ktorá nám riešenie zjednoduší, treba sa zamyslieť nad tým, či sú úlohy naozaj ekvivalentné a či sa pôvodná naozaj nedá vyriešiť efektívnejšie.

Listing programu:

```
program Priklad2;
var
  teraz, predtym: string;
  R, S, i, j: integer;
  lavy_koniec, pravy_koniec, min, max: integer;
  ostrov, pevnina: boolean;
```

[1] Ale nezužujeme, dokiaľ ostrov neskončí. Inak by sa mohlo stať, že by sme nezískali správne riešenie.

Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

```
begin
  ostrov := false;
  ReadLn(R, S);
  ReadLn(predtym);
  for i := 2 to R do begin
    ReadLn(teraz);
    min := S; max := 1;
    for j := 1 to S do
      if teraz[j] = '#' then begin
        if j < min then min := j;
        if j > max then max := j;
      end;
    pevnina := not ((min = S) and (max = 1));
    if ostrov then begin {ostrov prave bezi}
      if min > lavy_koniec + 1 then min := lavy_koniec + 1;
      if max < pravy_koniec - 1 then max := pravy_koniec - 1;
      for j := min - 1 to lavy_koniec do predtym[j] := 'G';
      for j := pravy_koniec to max + 1 do predtym[j] := 'G';
      if lavy_koniec > min - 1 then lavy_koniec := min - 1;
      if pravy_koniec < max + 1 then pravy_koniec := max + 1;
    end;
    if (ostrov) and (not pevnina) then begin {sme na konci ostrova}
      for j := lavy_koniec to pravy_koniec do teraz[j] := 'G';
      ostrov := false;
    end;
    if (not ostrov) and (pevnina) then begin {nasli sme zaciatok ostrova}
      lavy_koniec := min - 1;
      pravy_koniec := max + 1;
      ostrov := true;
      for j := lavy_koniec to pravy_koniec do predtym[j] := 'G';
    end;
    WriteLn(predtym);
    predtym := teraz;
  end;
  WriteLn(predtym);
end.
```

vzorák písal Ivan
(max. 10 bodov)

3. Zlovoľný populista

Na vyriešenie tejto úlohy stačilo vymyslieť relatívne jednoduchý greedy[2] algoritmus a napísať funkčný kód.

Každá zastávka pokrýva územie dĺžky $2d$ a našou úlohou je pokryť všetky uvedené body (domy) na priamke. Keď si zvolíme správny prístup k problému, ľahko vymyslíme jednoduchý spôsob rozostavovania zastávok, ktorý je optimálny (t. j. menej zastávok nestačí).

Uvažujme o ľubovoľnom pokrytí a skúmajme, ako môže vyzerať. Aby boli všetky domy pokryté, musí byť pokrytý aj ten úplne vľavo. Existuje teda (aspoň) jedna zastávka, ktorá ho pokrýva. Keďže vľavo od toho domu nie je nič, čo by bolo potrebné pokryť, môžeme túto zastávku posunúť čo najviac doprava tak, aby ho ešte pokrývala. Táto poloha zastávky je určite jedna z optimálnych (z pohľadu počtu zastávok), lebo najľavejší dom musí byť pokrytý

[2] Po slovensky pažravý. Takto označujeme algoritmy, ktoré si vyberajú z viacerých možností tú, ktorá je **momentálne** najlepšia, teda „nemyslia na budúcnosť“.

Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

a takto zastávka pokrýva všetko, čo môže (jej posúvaním doľava pokryjeme len menej a menej domov).

Keď sme si už určili pozíciu jednej zastávky, vieme zistiť, ktoré domy sú ňou pokryté a na tie nepokryté môžeme použiť rovnakú úvahu, až kým nepokryjeme všetky. Týmto spôsobom vyrobíme jedno možné pokrytie s najmenším možným počtom zastávok.

Podstatné je uvedomiť si, že keď sme vyberali miesto, kam dať prvú zastávku, museli sme ukázať, že lepšie sa to naozaj nedá. Bez ohľadu na to, ako sú umiestnené ostatné domy a kam budeme dávať ostatné zastávky, nejaká zastávka musí pokryť najľavejší dom. My poznáme najlepšie miesto pre ňu a môžeme ju tam teda dať – greedy algoritmus bude fungovať. Ak by sme začínali niekde v strede, nevieme len tak povedať, či nebude lepšie postaviť zastávku trochu doľava alebo doprava.

Naše riešenie prechádza zoznamom domov zľava doprava, pričom si pamätá polohu zatiaľ poslednej postavenej zastávky. Vždy, keď príde k domu (označme jeho pozíciu a_i), ktorý ešte nie je pokrytý, pridá novú zastávku čo najviac doprava (teda na pozíciu $a_i + d$).

Vďaka tomu, že zastávky sú už na vstupe usporiadané zľava doprava, časová zložitosť nášho riešenia je $O(n)$. Pamäťová zložitosť je $O(n)$, pretože kvôli formátu výstupu nemôžeme postupne vypisovať postavené zastávky, ale musíme si ich pamätať a vypísať až nakoniec.

Listing programu:

```
var n, d, m, p, i, a: longint;
    Z: array [1..1000000] of longint;

begin
  ReadLn(n, d);
  m := 0;  { počet zastávok }
  p := -d;  { poloha poslednej zastávky }

  for i := 1 to n do begin
    Read(a);
    if a - p > d then begin  { posledná zastávka nepokrýva a_i }
      p := a + d;
      Inc(m);
      Z[m] := p;
    end;
  end;

  WriteLn(m);
  for i := 1 to m - 1 do
    Write(Z[i], ' ');
  WriteLn(Z[m]);
end.
```

4. Zmiešané tímy

opravoval Bruno
(max. 15 bodov)

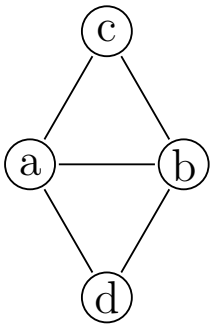
Táto úloha sa dala riešiť veľmi jednoduchým spôsobom – vždy, keď nájdeme nejakú trojicu ľudí, ktorí sú navzájom kamarátmi, vyberieme ich do tímu. Podstatnou časťou úlohy bol teda dôkaz správnosti – nemôže sa stať, že nahradením niektorých trojíc inými zvýšime množstvo tímov?

Dôkaz

Požičiame si terminológiu z teórie grafov – študentov budeme označovať ako vrcholy, priateľstvá medzi nimi ako hrany a tímy ako trojuholníky.

Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

Uvažujme trojicu študentov a, b, c, ktorí sú navzájom kamarátmi. Kedy by sa mohlo stať, že sa nám neoplatí vybrať ich ako tím? Znamenalo by to, že je lepšie vybrať nejaký iný trojuholník, ktorého vrcholom je aspoň jeden z týchto troch študentov. Pretože im zostáva už len najviac jedna voľná hrana, musí byť medzi vrcholmi tohto trojuholníka ešte jeden ďalší študent z trojice a, b, c. Bez ujmy na všeobecnosti, nech sú to študenti a a b, zostávajúci vrchol označme d.



Rozoberme dve možnosti: ak sú vrcholy c a d spojené hranou, potom títo štyria študenti tvoria samostatný komponent a nezáleží na tom, ktorý trojuholník z neho si vyberieme – vždy zostane jeden vrchol nevyužitý.

V opačnom prípade sa vrcholy c a d už určite nenachádzajú v ďalšom trojuholníku. Preto znova nezáleží na tom, či si vyberieme trojuholník a, b, c, alebo a, b, d – zvyšok grafu to neovplyvní.

Z toho nám vyplýva, že vybraním ľubovoľnej trojice voľných študentov, ktorí sú navzájom kamarátmi, si nepokazíme riešenie.

Implementácia

Pre každého študenta s_1 sa teda pokúsime nájsť tím, do ktorého by mohol patriť. Postupne prechádzame všetkých jeho kamarátov s_2; pre každého z nich zase prechádzame všetkých jeho kamarátov s_3. Takto máme vybranú trojicu študentov, stačí nám už len overiť, či sa s_1 priatelí s s_3. Ak áno, našli sme trojuholník a zaradených študentov si označíme, aby sme ich už nepriradili do ďalšieho tímu.

Toto celé trvá konštantný čas, keďže každý študent má najviac troch priateľov. Tento postup opakujeme pre každého študenta, časová zložitosť hľadania tímov bude preto $O(N)$. Načítanie tímov trvá $O(M)$. M je najviac $3N/2$, takže $M = O(N)$ a celková zložitosť programu bude $O(M + N) = O(N)$.

Pamätáme si dve polia, každé s veľkosťou N. Priateľstvá si pamätáme pre daného študenta ako zoznam priateľov, takže pamäťová zložitosť je $O(N)$.

Bodovanie

Za algoritmus bez dôkazu sa dalo získať najviac 6 bodov. Dobrý dôkaz vedel získať ďalších 9 bodov.

Listing programu:

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

int main() {
    int n, m, pocet = 0;
    scanf("%d %d", &n, &m);
    vector<vector<int> > G(n);  // graf priateľstiev
    vector<bool> U(n, false);  // je už študent v nejakom tíme?
    for (int i = 0; i < m; ++i) {
        int a, b;
        scanf("%d %d", &a, &b);
        --a, --b;
        G[a].push_back(b);
        G[b].push_back(a);
    }

    for (int i = 0; i < n; i++) {
        int s1 = i;
```


Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

```
        for (int j = 0; j < (int) G[s1].size(); j++) {
            int s2 = G[s1][j];
            for (int k = 0; k < (int) G[s2].size(); k++) {
                int s3 = G[s2][k];
                for (int el = 0; el < (int) G[s3].size(); el++) {
                    if (G[s3][el] == s1 && !U[s1] && !U[s2] && !U[s3]) {
                        // našli sme trojuholník
                        pocet++;
                        U[s1] = U[s2] = U[s3] = true;
                    }
                }
            }
        }
    }

    printf("%d\n", pocet);
}
```

opravoval Mic
(max. 15 bodov)

5. O dominách

Príklad nebol veľmi zložitý, napriek tomu prišlo len 25 riešení a nie všetky boli správne. Ale preto je tu vzorák, aby ste nabudúce dostali plný počet bodov.

Bodovanie

Za lineárnu časovú a pamäťovú zložitosť som dával najviac 14 bodov. Najviac 15 bodov bolo za konštantnú pamäťovú zložitosť a lineárny čas. Sľúbených 9 bodov dostali tí, čo riešili existenčnú úlohu. Nefunkčné riešenia dostali najviac 3 body. Body som strhával ako obyčajne za nedostatočný popis, drobné chyby v algoritme, bugy v kóde a chýbajúce odhady časovej a pamäťovej zložitosti.

Prevod na inú úlohu

Našou úlohou je z celej danej sady domín, ktoré obsahujú čísla 0 až 6, postaviť rad domín, ktoré na seba nadväzujú. My si túto úlohu preformulujeme do reči teórie grafov.[3] Vrcholy nám budú tvoriť čísla (od 0 po 6) a každé domino nám bude reprezentovať hranu. Keďže môžeme mať viac domín rovnakého tvaru, tak aj medzi dvoma vrcholmi môže byť viac hrán. V grafe môžu byť aj slučky (kvôli dominám tvaru (x, x)).

Ak si zoberieme nejaký korektný rad domín, napríklad $(1, 2), (2, 3), (3, 1), (1, 1)$, tak ten zodpovedá nejakému „rovnakému“ ťahu v grafe (jediný rozdiel je, že v rade sú domináa v ťahu sú hrany[4]). Rovnako to funguje aj opačným smerom. Ak zoberieme ťah v grafe, tak dostaneme korektný rad domín. Preto ak v grafe nájdeme ťah, ktorý prechádza cez všetky hrany, tak z neho vieme triviálne vyrobiť požadovaný rad domín. Ak taký ťah neexistuje, tak jeden rad zo všetkých domín spraviť nevieme. Takýto ťah v grafe sa volá eulerovský ťah a jeho hľadanie je známy a nie veľmi obtiažny problém. Odteraz zabudneme na domináa budeme sa rozprávať výhradne o grafoch.

Čo je to graf?

Každý graf sa skladá z množiny **vrcholov** V a množiny **hrán** E. Každá hrana je dvojica vrcholov z V. Hrany obyčajne zapisujeme ako usporiadané dvojice, čiže hranu z vrcholu u do vrcholu v zapíšeme ako (u, v). Hovoríme, že vrcholy u aj v sú **incidentné** s hranou

[3] Tí, ktorí nevedia, čo je to graf, nech si prečítajú ďalší odsek a potom pokračujú v čítaní tohto odseku.

[4] Alebo hrany sú v ťahu?

Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

(u, v). Vrcholy môžu reprezentovať napríklad križovatky a hrany ulice medzi jednotlivými križovatkami. Alebo vrcholy budú čísla a hrany budú domináa.

Hrany v našom prípade sú neorientované (to zodpovedá cestnej sieti bez jednosmeriek, pri dominách to znamená, že ich môžeme otáčať). Hrana môže byť aj tvaru (u, u), čo zodpovedá dominu s rovnakými číslami alebo „okružnej ulici“. Takýmto hranám budeme hovoriť **slučky**. Ak medzi vrcholmi u a v existuje hrana (u, v), tak hovoríme, že vrcholy u a v sú **susedné** a že vrchol u je **susedom** vrcholu v (a naopak). V grafe môžu byť hrany viackrát. Vtedy hovoríme, že daná hrana je **viacnásobná**.

Cesta v grafe je postupnosť vrcholov $u_1, u_2, \ldots, u_k$, pričom vrcholy u_i a u_{i+1} sú susedné. V ceste sa nemôžu vrcholy opakovať. Naopak **ťah** je „cesta“, v ktorej sa vrcholy môžu opakovať, ale hrany nie. V prípade viacnásobných hrán, ťah môže obsahovať hranu najviac toľkokrát, koľkokrát je daná hrana v grafe. Rovnako ako môžeme cesty a ťahy charakterizovať postupnosťou vrcholov, vieme tak spraviť aj postupnosťou hrán ktoré ležia v ťahu, čo budeme častejšie používať.

Graf je **súvislý**, ak existuje cesta medzi ľubovoľnou dvojicou vrcholov (inými slovami, medzi každými dvoma križovatkami sa dá dostať tak, že budeme chodiť len po existujúcich uliciach).

Príklad: $V = \{1, 2, 3\}$, $E = \{(2, 1), (1, 2), (2, 2), (2, 3)\}$. Tento graf obsahuje 3 vrcholy a 4 hrany. Graf obsahuje slučku $(2, 2)$ a viacnásobnú hranu $(1, 2)$ (keďže hrany sú neorientované, tak hrana $(1, 2)$ je vlastne taká istá ako hrana $(2, 1)$). Hrany v poradí, v akom sú vymenované, tvoria ťah.

Existencia eulerovského ťahu

Budeme predpokladať, že graf neobsahuje slučky. Ak obsahuje, tak ich vymažeme (a zapamätáme si to) a na konci slučky jednoducho vložíme do ťahu. Napríklad ak sme vymazali slučku $(2, 2)$ a našli sme ťah $(2, 1), (1, 2), (2, 3)$, tak slučku vložíme do ťahu napríklad na začiatok alebo na predposledné miesto.

Pre potreby tohto vzoráku si zavedieme pojem **párny** vrchol a **nepárny** vrchol. Párny vrchol je taký vrchol, ktorý má párny počet incidentných hrán; nepárny vrchol má nepárny počet incidentných hrán. Keď budeme hovoriť o parite vrcholu, budeme mať na mysli, či je párny alebo nepárny.

Graf musí spĺňať niekoľko vecí na to, aby v ňom existoval eulerovský ťah. V prvom rade musí byť súvislý. Ak nie je súvislý, tak jeden ťah nemôže pokryť všetky hrany. Pozrime sa teraz na eulerovský ťah. Zoberme si ľubovoľný vrchol u. Aká je jeho parita? Predpokladajme na chvíľku, že u neleží ani na jednom konci ťahu. Za každý výskyt vrcholu u v ťahu si zarátame dve hrany, s ktorými je u incidentný.

Ak sa u vnútri ťahu nachádza k-krát, potom je incidentný s $2k$ hranami a je párny. Ak u leží aj na okraji ťahu, tak je incidentný s hranou navyše. Ak leží iba na jednom konci, má $2k + 1$ hrán a je nepárny. Ak leží na oboch koncoch, tak má $2k + 2$ hrán (a je párny). Ak u leží iba na jednom kraji, potom vrchol na druhom kraji je tiež nepárny. To znamená, že 0 alebo 2 vrcholy sú nepárne a ostatné sú párne.

Ak graf porušuje túto podmienku, tak nemôže obsahovať eulerovský cyklus. Ako vidíme, táto podmienka je nutná. Je aj postačujúca? Áno, je. To znamená, že stačí skontrolovať súvislosť grafu a paritu pre každý vrchol grafu. V ďalšom odseku sa dozviete, prečo je to aj postačujúca podmienka.

Hľadanie ťahu

Ukážeme, ako v súvislom grafe, ktorý má najviac[5] dva nepárne vrcholy, nájsť eulerovský ťah. Pre začiatok budeme predpokladať, že všetky vrcholy sú párne. Začneme tak, že si vyberieme ľubovoľný vrchol (napríklad u) a budeme sa hýbať po grafe tak, že začneme vo

[5] Kvízová otázka: môže mať iba jeden nepárny vrchol?

Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

vrchole u a vždy sa posunieme do ľubovoľného susedného vrcholu po hrane, ktorou sme ešte nešli. Takto vyrobíme nejaký ťah, ktorý ale ešte nemusí byť eulerovský. V ktorom vrchole tento ťah končí?

Poďme počítať paritu vrcholov. Na začiatku je každý vrchol párny. Keď sa posunieme z vrcholu u do susedného vrcholu v a „vymažeme“ príslušnú hranu, u aj v sa stanú nepárnymi. Z toho, že v je nepárny, vyplýva, že je incidentný s aspoň jednou nepoužitou hranou. Preto z v vieme predĺžiť ťah do ďalšieho vrcholu (a „vymažeme“ príslušnú hranu). Týmto úkonom zmeníme paritu vrcholu v na párnu a tiež zmeníme paritu ďalšieho vrcholu. Ak to nie je vrchol u, tak to znamená, že vrchol je nepárny a preto z neho vedie aspoň jedna nepoužitá hrana. Ak je to vrchol u, tak potom jeho parita je párna a potenciálne už nemusí byť žiadna hrana voľná.

Ako vidíme, nech sa hýbeme po grafe akokoľvek, zastať môžeme len vo vrchole u (lebo ináč vrchol, v ktorom sme, je nepárny a teda je incidentný s nepoužitou hranou). Preto keď skončíme, postupnosť vrcholov, ktoré sme navštívili, bude tvoriť cyklus. Graf, ktorý ostal, už nemusí byť súvislý, ale všetky vrcholy v ňom sú párne.

Zoberme si teraz nejaký vrchol, ktorý leží na cykle, ale ostávajú mu ešte nejaké hrany. Ak z neho začneme vytvárať nový ťah (samozrejme bez hrán, ktoré sú v predchádzajúcom cykle), tak vytvoríme ďalší cyklus. Momentálne máme dva cykly, ktoré sú spojené (aspoň) jedným vrcholom. V tomto vrchole môžeme oba cykly rozpojiť a spojiť ich do jedného veľkého cyklu. Ak tento cyklus obsahuje všetky hrany, tak sme vyhrali; v opačnom prípade si opäť vyberieme vrchol, ktorý leží v cykle a má ešte nepoužité hrany, a ...

Týmto spôsobom budeme postupne zväčšovať cyklus, až nám na koniec neostanú žiadne hrany (graf je súvislý). Vtedy tento cyklus rozpojíme v nejakom vrchole, čím dostaneme hľadaný ťah. Čo ak graf na začiatku obsahuje dva nepárne vrcholy? Vtedy prvé prehľadávanie začneme v jednom z týchto vrcholov a ono nutne skončí v druhom nepárnom vrchole (argumenty, prečo je to tak, sú rovnaké ako vyššie). Po tomto prehľadaní nám ostane graf s párnym počtom vrcholov, čo už je situácia, ktorú vieme riešiť.

Ako túto vec implementovať? Priamočiaro by to asi nebolo príliš efektívne: nájdeme cyklus, nájdeme vrchol, ktorému ostávajú hrany, nájdeme ďalší cyklus, ... Namiesto toho môžeme efektívne využiť rekurziu. Spravíme si rekurzívnu funkciu, ktorá bude prehľadávať graf do hĺbky. Bude vyzerať nejako takto (G[vrchol] je zoznam hrán, s ktorými vrchol susedí):

```
euler (vrchol):
    for hrana in G[vrchol]:
       if not navstivena[hrana]:
            navstivena[hrana]:=true;
            euler(druhy_koniec_hrany(hrana,vrchol));
            break;
    print vrchol
```

V tomto programe nájdeme prvú hranu, ktorá nie je použitá, označíme ju za použitú a rekurzívne sa na ňu zavoláme. Keď sa vrátime z rekurzie, skončíme cyklus `for` a vypíšeme vrchol. Všimnite si, že takto vypíšeme cyklus, ktorý sme našli, v opačnom poradí.

Čo by sa stalo, keby sme odstránili príkaz `break`, ktorý ukončí cyklus? Po návrate z rekurzie sa zavoláme aj na ďalšie nenavštívené hrany z vrcholu v (ak ešte také sú). Keďže ostávajúci graf má iba párne vrcholy, toto prehľadávanie skončí vo vrchole v a pri vracaní sa naspäť vypíše cyklus z vrcholu v do vrcholu v, čím sme efektívne spojili oba cykly.

Redukcia pamäťovej zložitosti

Za predchádzajúce riešenie by ste dostali 14 bodov. Prečo? Lebo množstvo pamäte, ktoré predchádzajúce riešenie používa, lineárne závisí od počtu hrán v grafe. Aj keby sme si nepamätali každú hranu zvlášť, ale pre každý typ hrany len počet. Prečo? Hore uvedený

Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

algoritmus používa rekurziu.[6] Vždy, keď sa rekurzívne zavoláme, program si uloží na stack hodnoty parametrov funkcie a jej lokálne premenné (aby ich vedel obnoviť po návrate z funkcie). Preto pamäťová zložitosť algoritmu závisí lineárne od hĺbky rekurzie.[7]

Ako redukovať pamäťovú zložitosť? Zoberme si najskôr hrany/domináa tvaru (x, x). Tieto môžeme z grafu odstrániť a na konci ich jednoducho pridať do postupnosti. Zoberme si teraz ostatné hrany, ktoré sú v grafe veľakrát. Napríklad nech hrana $(1, 2)$ sa nachádza v grafe 47 krát. To, či existuje eulerovský ťah, závisí len od parity vrcholov. Preto ak vymažeme párny počet hrán $(1, 2)$, tak sa nám parita vrcholov 1 a 2 nezmení. Preto môžeme z grafu vymazať 46 hrán tvaru $(1, 2)$ a na konci pridať do postupnosti na správne miesto 23 „slučiek" tvaru $(1, 2), (2, 1)$. Pri mazaní si ale treba dať pozor. Ak by sme mali 74 hrán tvaru $(1, 2)$, tak vymazaním 74 z nich síce paritu vrcholov nezmeníme, ale môžeme rozpojiť graf. Preto v takom prípade vymažeme iba 72 hrán.

Ak toto aplikujeme na každú hranu, dostaneme graf, ktorý má konštantnú veľkosť (7 vrcholov, najviac 42 hrán). Potrebujeme si poznačiť aj vymazané hrany (pripomíname, že na začiatku sme z grafu vymazali slučky). Ak si ich nebudeme pamätať po jednom, ale pre každý typ vymazanej hrany si zapamätáme len ich počet, budeme na to potrebovať 28 čísiel. Pamäťová zložitosť ostáva konštantná, graf má konštantnú veľkosť, takže aj hĺbka rekurzie bude konštantná.

Časová a pamäťová zložitosť

Keďže púšťame algoritmus na graf konštantnej veľkosti, časová aj pamäťová zložitosť je konštantná. Avšak vstup musíme načítať a spracovať, čo nám trvá lineárny čas. Preto celková časová zložitosť algoritmu je $O(N)$ a pamäťová zložitosť je $O(1)$.

Poznámka

Časová zložitosť všeobecného algoritmu na hľadanie eulerovskej cesty závisí od reprezentácie grafu (predpokladáme, že hrany nie sú viacnásobné a graf neobsahuje slučky). Nech N je počet vrcholov grafu a M je počet jeho hrán. Ak si pre každý vrchol pamätáme zoznam jeho susedov, tak časová aj pamäťová zložitosť tohto algoritmu je $O(N + M)$. Ak si pamätáme graf ako maticu susednosti, tak časová aj pamäťová zložitosť algoritmu je $O(N^2)$.

Bonus

Všimnite si, že pri redukcii pamäťovej zložitosti sme spúšťali algoritmus na hľadanie eulerovského ťahu na graf, ktorý má konštantnú veľkosť (máme 7 vrcholov, medzi každou dvojicou vrcholov sú najviac 2 hrany). Preto môžeme použiť ľubovoľný funkčný algoritmus na hľadanie eulerovského ťahu, kľudne aj taký s exponenciálnou časovou zložitosťou (napríklad skúšanie všetkých možných ťahov) a stále si zachováme konštantnú pamäťovú zložitosť a lineárny čas výpočtu.

Implementácia

Vzorové programy nájdete dnes dva. Obidva sú takmer úplne totožné, akurát jeden je napísaný v C++ a druhý v Pascale. Graf si reprezentujeme ako maticu susednosti G. To je dvojrozmerné pole veľkosti 7 krát 7, kde si na pozícii $G[i][j]$ pamätáme, koľko hrán z i do j máme (každú hranu máme poznačenú dvakrát, raz v $G[i][j]$ a raz v $G[j][i]$). V poli H, ktoré má rovnaké rozmery ako G, si pamätáme, koľko hrán sme na začiatku z grafu odstránili (pamätáme si tam aj slučky). V poli W si pamätáme, koľko hrán sme už použili. Takže ak $W[i][j] < G[i][j]$, tak to znamená, že z vrcholu i existuje nepoužitá hrana do vrcholu j.

Vzorák v C++:

[6] Ak to implementujeme bez rekurzie, budeme mať rovnaký problém.

[7] Kvízová otázka: Akú hĺbku „rekurzie" mal film Inception (slovenský názov je Počiatok)?

Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

Listing programu:

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

vector<vector<int> > G, H, W;
vector<int> stupne;
vector<int> tah;

void euler(int vrchol) {
    for (unsigned int i = 0; i < G[vrchol].size(); i++) {
        if (W[vrchol][i] < G[vrchol][i]) {
            W[vrchol][i]++;
            W[i][vrchol]++;
            euler(i);
            tah.push_back(vrchol);
        }
    }
}

void vypis_slucku(int vrchol) {
    while (H[vrchol][vrchol] > 0) {
        cout << vrchol << vrchol << " ";
        H[vrchol][vrchol]--;
    }
    for (unsigned int i = 0; i < G[vrchol].size(); i++) {
        while (H[vrchol][i] > 0) {
            H[vrchol][i] -= 2;
            H[i][vrchol] -= 2;
            cout << vrchol << i << " " << i << vrchol << " ";
        }
    }
}

int main() {
    int N;
    int start = 0;
    int Hran = 0;
    cin >> N;
    if (N == 0) {
        cout << "Da sa." << endl;
        return 0;
    }
    G.resize(7, vector<int>(7, 0) );
    W = H = G;
    stupne.resize(7, 0);
    //Nacitame maticu susednosti
    for (int i = 0; i < N; i++) {
        int a, b;
        cin >> a >> b;
        G[a][b]++;
        G[b][a]++;
        start = a;
    }
    //Zmensime graf
    for (unsigned int i = 0; i < G.size(); i++) {
        for (unsigned int j = 0; j < G[i].size(); j++) {
            if (G[i][j] > 2 && i != j) {
                if (G[i][j] % 2 == 1) {
```


Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

```
                H[i][j] = G[i][j] - 1;
            } else {
                H[i][j] = G[i][j] - 2;
            }
            G[i][j] -= H[i][j];
        }
        if (i == j) {
            H[i][j] = G[i][j] / 2;
            G[i][j] = 0;
        }
        Hran += G[i][j];
    }
}
//Skontrolujeme, ci nemame velky pocet neparnych vrcholov
int nepar = 0;
for (int i = 0; i < (int) stupne.size(); i++)
    if (stupne[i] % 2 == 1) {
        nepar++;
        start = i;
    }
if (nepar > 2) {
    cout << "Neda sa." << endl;
    return 0;
}
euler(start);
//Kontrola, ci graf nie je suvisly -- Ak sme nepouzili vsetky hrany
if ((int) tah.size() != Hran) {
    cout << "Neda sa." << endl;
    return 0;
}
cout << "Da sa." << endl;
for (int i = 0; i < (int) tah.size() - 1; i++) {
    vypis_slucku(tah[i]);
    cout << tah[i] << tah[i + 1] << " ";
    vypis_slucku(tah[i + 1]);
}
if (tah.size() > 0) cout << tah[tah.size() - 1] << start << " ";
vypis_slucku(start);
cout << endl;
}
```

Vzorák v Pascale:

Listing programu:

```
const maxN = 100000;
var N, i, j, dlzkaTahu, a, b, start, nepar, Hran: integer;
    G, H, W: array [0..6, 0..6] of integer;
    stupne: array[0..6] of integer;
    tah: array [0..maxN] of integer;


procedure euler(vrchol: integer);
var i: integer;
begin
  for i := 0 to 6 do begin
    if W[vrchol][i] < G[vrchol][i] then begin
      Inc(W[vrchol][i]);
      Inc(W[i][vrchol]);
```


Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

```
      euler(i);
      tah[dlzkaTahu] := i;
      Inc(dlzkaTahu);
    end;
  end;
end;

procedure vypis_slucku(vrchol: integer);
var i: integer;
begin
  while H[vrchol][vrchol] > 0 do begin
    Write(vrchol, vrchol, ' ');
    Dec(H[vrchol][vrchol]);
  end;
  for i := 0 to 6 do begin
    while H[vrchol][i] > 0 do begin
      Write(vrchol, i, ' ', i, vrchol, ' ');
      Dec(H[vrchol][i], 2);
      Dec(H[i][vrchol], 2);
    end;
  end;
end;

begin
  ReadLn(N);
  if N = 0 then begin
    WriteLn('Da sa');
    Exit;
  end;
  for i := 0 to 6 do stupne[i] := 0;
  for i := 0 to 6 do for j := 0 to 6 do begin
    G[i][j] := 0;
    H[i][j] := 0;
    W[i][j] := 0;
  end;
  {Nacitame maticu susednosti}
  for i := 0 to N - 1 do begin
    Read(a, b);
    Inc(G[a][b]);
    Inc(G[b][a]);
    start := a;
  end;
  {Zmensime graf}
  Hran := 0;
  for i := 0 to 6 do for j := 0 to 6 do begin
    if (G[i][j] > 2) and (i <> j) then begin
        if G[i][j] mod 2 = 1 then begin
          H[i][j] := G[i][j] - 1;
        end else begin
          H[i][j] := G[i][j] - 2;
        end;
        Dec(G[i][j], H[i][j]);
    end;
    if i = j then begin
      H[i][j] := G[i][j] div 2;
      G[i][j] := 0;
    end;
    Inc(Hran, G[i][j]);
  end;
  {Skontrolujeme, ci nemame velky pocet neparnych vrcholov}
```


Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

```
  nepar := 0;
  for i := 0 to 6 do begin
    if stupne[i] mod 2 = 1 then begin
      Inc(nepar);
      start := i;
    end;
  end;
  if nepar > 2 then begin
    WriteLn('Neda sa.');
    Exit;
  end;
  euler(start);
  {Kontrola, ci graf nie je suvisly -- Ak sme nepouzili vsetky hrany}
  if dlzkaTahu <> Hran then begin
    WriteLn('Neda sa.');
    Exit;
  end;
  WriteLn('Da sa.');
  for i := 0 to dlzkaTahu - 2 do begin
    vypis_slucku(tah[i]);
    Write(tah[i], tah[i + 1], ' ');
    vypis_slucku(tah[i + 1]);
  end;
  if dlzkaTahu > 0 then WriteLn(tah[dlzkaTahu - 1], start);
  vypis_slucku(start);
  WriteLn;
end.
```

6. Oleja je dosť, nenaolejujeme aj lampu?

vzorák písal Bob
(max. 20 bodov)

Všimnime si, že jazykolamy sú vlastne dobre uzátvorkované postupnosti dvoch druhov zátvoriek: *pr* a *st*. Našou úlohou je nájsť dobre uzátvorkovanú postupnosť rovnakej dĺžky, ktorá v abecednom poradí nasleduje.

Základná myšlienka riešenie bude podobná, ako keď hľadáme nasledujúcu permutáciu: chceme čo najdlhší prefix jazykolamu zachovať bez zmeny, ďalší znak čo najmenej zväčšiť a zvyšok doplniť na čo najmenší (v abecednom poradí) jazykolam.

Ako príklad si vezmime vzorový vstup *pprrst*. Najdlhším prefixom, ktorý môžeme zachovať, je *ppr* (ak by sme nechali bez zmeny *pprr*, mohli by sme to doplniť iba s *pr* alebo *st*, tým by ale nevznikol väčší jazykolam). Nasledujúci znak *r* zväčšíme na *s*, zatiaľ máme *pprs*. Zvyšok sa dá doplniť jediným spôsobom (keďže potrebujeme zatvoriť zátvorky, ktoré otvorili *p* a *s*), vznikne nám tak jazykolam *pprstr*.

Budeme teda postupne skúšať prefixy zadaného jazykolamu od najdlhšieho po najkratší. Aby sme vedeli určiť, čím sa dá aktuálny prefix doplniť na jazykolam, v zásobníku si budeme pamätať zoznam zátvoriek, ktorými ešte potrebujeme uzatvoriť otvorené zátvorky v prefixe. Napríklad pre prefix *prpssts* budeme mať na zásobníku zhora nadol znaky *t*, *t* a *r*.

Ako upraviť obsah zásobníka, keď prechádzame z prefixu wc na kratší prefix w (c je nejaký zo znakov *p*, *r*, *s*, *t*)? Ak je c uzatvárajúcou zátvorkou, vložíme ju na vrch zásobníka, keďže má niekde v prefixe w svoj otvárajúci náprotivok. V opačnom prípade odstránime prvok z vrchu zásobníka – tento znak mal uzatvoriť zátvorku, ktorú otvára c, to už ale nie je potrebné.

Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

Keď sme si už určili konkrétny prefix a upravili zásobník, vyskúšame postupne všetky možnosti pre nasledujúce písmeno (vyberieme si najmenšiu vyhovujúcu). Nech pôvodne nasledovalo písmeno c, my chceme použiť písmeno d ($c < d$). V prípade, že je d uzatvárajúcou zátvorkou, musíme ešte skontrolovať, či je najbližšia neuzavretá zátvorka v prefixe rovnakého druhu. Na to využijeme hodnotu na vrchu zásobníka.

Nakoniec skontrolujeme, či sa dá náš prefix nasledovaný písmenom d doplniť na jazykolam. Určite potrebujeme aspoň uzavrieť všetky neuzavreté zátvorky. Či sa to stíha, zistíme porovnaním počtu znakov zostávajúcich do konca jazykolamu s veľkosťou zásobníka. Ak nám ešte zostala rezerva, voľné miesto vyplníme reťazcom *ppp...pr...rrr*.

Pre každý nepoužiteľný prefix jazykolamu vykonáme $O(1)$ operácií (okrem toho, pri ktorom nakoniec zmeníme koniec postupnosti, tam vykonáme $O(N)$ operácii, ale vtedy aj skončíme). Celková časová zložitosť tohto riešenia je preto lineárna od dĺžky vstupu.

Keďže táto úloha bola praktická, záležalo aj na bezchybnej implementácii vášho riešenia. Žiaľ, dosť veľa súťažiacich dostalo 0 bodov. Ak si neviete nájsť chybu, skúste si napísať nejaké pomalšie (aj exponenciálne) riešenie a porovnávajte s ním výsledky vášho programu. Často sa problém prejaví už na malých vstupoch.

Listing programu:

```
#include <iostream>
#include <string>
#include <stack>
using namespace std;

char Z[4] = {'p', 'r', 's', 't'};

inline bool zatvara(char c) {
    return c == 'r' || c == 't';
}

int main() {
    string J;
    cin >> J;
    int n = J.size();
    stack<char> S;

    for (int i = n - 1; i >= 0; --i) {
        if (zatvara(J[i]))  // upravíme zásobník
            S.push(J[i]);
        else
            S.pop();

        for (int j = 0; j < 4; ++j)
            if (Z[j] > J[i]){  // chceme väčší znak
                if (zatvara(Z[j])) {
                    if (S.empty() || S.top() != Z[j])
                        continue;  // uzatvárajúca zátvorka nepasuje
                    S.pop();
                } else
                    S.push(Z[j + 1]);  // uzatvárajúca k Z[j]

                int k = n - 1 - i - S.size();  // veľkosť rezervy
                if (k >= 0) {
                    cout << J.substr(0, i) << Z[j];  // vypíšeme prefix, zväčšený znak,
                    cout << string(k / 2, 'p') << string(k / 2, 'r'); // ppp...pr...rrr,
                    while (!S.empty()) {
                        cout << S.top();  // a nakoniec obsah zásobníka
```


Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

```
                    S.pop();
                }
                cout << endl;
                return 0;
            }

            if (zatvara(Z[j]))
                S.push(Z[j]);
            else
                S.pop();
        }
    }
}
```

opravoval Mišof
(max. 20 bodov)

7. Opozícia opäť zasahuje

Táto úloha nie je práve ťažká, jej riešenie ale vyžaduje postupne niekoľko nezávislých krokov. Prehrýzť sa cez všetky dá zabrať, čo sa tragicky odrazilo na počte riešiteľov. V tomto vzorovom riešení preto pôjdeme na vec pekne pomaly, krok za krokom.

Zbavme sa priemeru

V prvom rade si môžeme uvedomiť, že optimalizovať *priemernú vzdialenosť* dom-zastávka je to isté ako optimalizovať *súčet vzdialeností* dom-zastávka. Priemer totiž nie je nič iné ako súčet vydelený číslom n, teda počtom domov. A počet domov je stále ten istý.

My teda budeme vo zvyšku riešenia uvažovať o úlohe, kde optimalizujeme súčet vzdialeností domov od najbližších zastávok. Výhodou bude, že si v riešení vystačíme s celými číslami, a teda všetky výpočty budú presné.

Jedna zastávka s nekonečným dosahom

Na úvod sa pozrieme na jednoduchšiu úlohu. Zabudneme na to, že máme nejaký maximálny dosah zastávky d. A tiež na to, že môžeme použiť viac ako jednu zastávku. V našej zjednodušenej úlohe teda máme dedinu, ktorú treba celú pokryť jednou zastávkou. Kam je optimálne ju umiestniť?

Prvým tipom by mohol byť *priemer* súradníc domov, no ľahko sa ukáže, že tento tip nemusí vždy fungovať. (Pre dedinu s domami 0, 40 a 200 je optimálnym riešením 40, nie 80.)

Keď máme postupnosť tvorenú nepárnym počtom prvkov, jej *medián* je prvok, ktorý by bol presne uprostred, keby sme prvky postupnosti usporiadali podľa veľkosti. Pre postupnosť s párnym počtom prvkov existuje viacero rôznych definícií mediánu, najčastejšie sa však používa priemer hodnôt dvoch prvkov, ktoré sú prostredné podľa veľkosti.

A práve medián je najlepším miestom pre zastávku. Prečo je to tak? Predstavme si, že zastávku posúvame smerom zľava doprava a sledujeme súčet vzdialeností od domov. Kým je vľavo od zastávky menej domov ako vpravo, jej posunutím doprava sa súčet zmenší. Keď už sa dostaneme za prostredný prvok, bude viac domov, od ktorých sa budeme vzďaľovať, ako tých, ku ktorým sa budeme blížiť. Preto bude súčet zase stúpať. Najmenší súčet je teda v mediáne všetkých súradníc. (Ak je n párne, bude všade medzi prostrednými dvoma domami súčet rovnaký, môžeme teda použiť ľubovoľný z týchto bodov.)

Jedna zastávka s obmedzeným dosahom

Čo sa teraz stane, ak pridáme premennú d? Tá nám obmedzuje množinu bodov, kam môžeme zastávku umiestniť: môže byť najviac d pred posledným domom, a zároveň najviac d za prvým. Množina miest, kam môžeme zastávku umiestniť, je teda vždy nejaký konečný interval. (A ak je medzi prvým a posledným domom vzdialenosť väčšia ako $2d$, je tento interval prázdny.)

Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

Ak tento interval obsahuje medián, je všetko v poriadku, optimálne umiestnime zastávku a vyhrali sme. V opačnom prípade je zjavne optimálne, keď umiestnime zastávku najbližšie k mediánu, ako sa dá – teda do toho konca intervalu, ktorý je k mediánu bližšie. (Totiž keby sme ju posúvali smerom k opačnému koncu intervalu, tak ju posúvame ďalej od mediánu, a teda celková vzdialenosť domov od nej narastá.)

Napríklad pre $d = 100$ uvažujme dedinu s domami na súradniciach 0, 10, 20, 30, 40, 100, 170. Optimálne by bolo umiestniť zastávku pred dom na súradnici 30. Ak ale máme pokryť všetky tieto domy, musí zastávka stáť niekde v intervale $[70, 100]$. Preto je optimálnym riešením postaviť zastávku na súradnici 70 – najbližšie k mediánu ako smieme.

Predpočítanie v kubickom čase

Predpokladajme, že domy majú čísla od 0 po $n-1$. Pre domy s číslami $\{i, \ldots, j\}$ označíme $P[i, j]$ optimálnu polohu zastávky a $S[i, j]$ zodpovedajúci súčet vzdialeností týchto domov od nej.

Keďže domy už máme usporiadané, pre ľubovoľné i a j vieme v konštantnom čase určiť medián súradníc domov a z neho hodnotu $P[i, j]$. Následne vieme v čase $O(j - i)$ spočítať hodnotu $S[i, j]$. Ak pre nejaké i a j nevieme dotyčný úsek pokryť jednou zastávkou, bude $S[i, j] = \infty$ a $P[i, j]$ ľubovoľné.

Takéto riešenie teda vypočíta polia P a S dokopy v čase $O(n^3)$. Neskôr ukážeme efektívnejší spôsob výpočtu poľa S. Ale najskôr sa pozrieme, ako polia P a S použiť.

Optimálny počet zastávok

Určite nič nepokazíme, keď si rovnako ako v úlohe 3 vopred zistíme, koľko že to zastávok môžeme použiť. Tento optimálny počet, označme ho z, vieme zistiť pažravo v čase $O(n)$.

Dynamické programovanie

Potrebujeme teraz zodpovedať otázku: aký najmenší súčet vzdialeností vieme dosiahnuť, ak celú dedinu, teda všetkých n domov, korektne pokryjeme z zastávkami. Označme si odpoveď na túto otázku $B[n, z]$.

Ako túto hodnotu vypočítať? Pozrime sa na poslednú zastávku. Tá „obslúži“ niekoľko posledných domov. V tejto chvíli môžeme mať na výber viacero možností, koľko týchto domov bude – vyskúšame teda všetky a vyberieme najlepšiu z nich. Predpokladajme teda, že už sme sa rozhodli, že posledná zastávka obsluhuje domy m až $n-1$. V poliach P a S už máme spočítané, kam túto zastávku postaviť a aký bude súčet vzdialeností obslúžených domov od nej. A čo nám ostalo? Prvých m domov, ktoré treba optimálne pokryť $z-1$ zastávkami. A optimálnu cenu toho vieme v našom značení zapísať ako $B[m, z-1]$.

Platí teda nasledovný vzťah:

$$B[n, z] = \min_{0 \le m < n} (S[m, n-1] + B[m, z-1])$$

(A nesmieme zabudnúť na okrajové podmienky: Ak nám už nezostali žiadne zastávky, teda ak $z = 0$, tak nám už nesmeli zostať ani nepokryté domy. Preto $B[0, 0] = 0$ a $B[n, 0] = \infty$ pre $n > 0$.)

Všetky hodnoty od $B[0, 0]$ po $B[n, z]$ vieme vypočítať v čase $O(n^2 z)$, čo môžeme zhora odhadnúť ako $O(n^3)$. Tým dostávame celé riešenie fungujúce s touto časovou zložitosťou.

Listing programu:

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

#define MAX 1024
#define INF (1LL << 60)
int N, D;     // pocet domov a max. vzdialenost
```


Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

```
int Z;     // min pocet zastavok na pokrytie dediny
int X[MAX];     // suradnice domov
int P[MAX][MAX];     // optimalna poloha zastavky pre usek
long long S[MAX][MAX];     // optimalny sucet vzdialenosti pre usek
long long B[MAX][MAX];     // optimalne riesenie pre cast dediny

void getP(int i, int j) {
    P[i][j] = X[(i + j) / 2];
    if (P[i][j] < X[j] - D) P[i][j] = X[j] - D;
    if (P[i][j] > X[i] + D) P[i][j] = X[i] + D;
}

void getS(int i, int j) {
    S[i][j] = 0;
    for (int k = i; k <= j; ++k) S[i][j] += abs(P[i][j] - X[k]);
    if (X[j] - X[i] > 2 * D) S[i][j] = INF;
}

int getZ() {
    int answer = 1, start = X[0];
    for (int n = 1; n < N; ++n)
        if (X[n] > start + 2 * D) { start = X[n]; ++answer; }
    return answer;
}

int main() {
    // nacitame vstup
    cin >> N >> D;
    for (int n = 0; n < N; ++n) cin >> X[n];
    // predpocitame P a S
    for (int z = 0; z < N; ++z)
        for (int k = z; k < N; ++k) {
            getP(z, k); getS(z, k);
        }
    // pazravo zistime Z
    Z = getZ();
    // postupne pocitame B
    for (int z = 0; z <= Z; ++z) B[0][z] = 0;
    for (int n = 1; n <= N; ++n) B[n][0] = INF;
    for (int z = 1; z <= Z; ++z)
        for (int n = 1; n <= N; ++n) {
            B[n][z] = INF;
            for (int m = 0; m < n; ++m)
                B[n][z] = min(B[n][z], B[m][z - 1] + S[m][n - 1]);
        }
    cout << "optimalny sucet vzdialenosti: " << B[N][Z] << endl;
    // iduc od konca, vypiseme jedno riesenie
    while (Z > 0) {
        for (int m = 0; m < N; ++m)
            if (B[N][Z] == B[m][Z - 1] + S[m][N - 1]) {
                cout << P[m][N - 1] << endl;
                --Z;
                N = m;
                break;
            }
    }
}
```


Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

Lepšie dynamické programovanie

Aby sme predchádzajúce riešenie zrýchlili, treba si uvedomiť nasledovnú skutočnosť: rovnako ako v úlohe 3, aj teraz zostrojujeme jedno možné pokrytie dediny *najmenším* počtom zastávok – len tentokrát spomedzi všetkých možných pokrytí musíme vybrať jedno špeciálne.

Predstavme si, že zostrojujeme hľadané optimálne riešenie. Ideme od konca dediny, niekoľko domov sme už pokryli niekoľkými zastávkami. V predchádzajúcom riešení sme situáciu, v ktorej sa nachádzame, popísali dvoma parametrami: počet zostávajúcich domov a počet zostávajúcich zastávok. Lenže toho druhého parametra sa vieme zbaviť – vieme totiž, že zostávajúce domy *musíme* pokryť *najmenším možným* počtom zastávok.

Presnejšie to bude vyzerať nasledovne. Označme $Z[x]$ minimálny počet zastávok potrebný na pokrytie prvých x domov. Celé pole Z vieme zjavne vyplniť v lineárnom čase počas toho, ako zisťujeme minimálny celkový počet zastávok.

Nové otázky, ktoré si teraz budeme klásť, vyzerajú nasledovne: Aký najmenší súčet vzdialeností vieme dosiahnuť, ak pokryjeme prvých x domov optimálnym počtom zastávok? Odpoveď na túto otázku označíme $C[x]$.

Pre hodnoty C teraz platí podobný vzťah ako pre hodnoty B: $C[0] = 0$ a

$$C[x] = \min_{y:Z[y]+1=Z[x]} (S[y, x-1] + C[y])$$

Slovne: Hľadám optimálne riešenie pre posledných x domov. Niekoľko z nich pokryjem poslednou zastávkou. Ostane mi y domov. Toto y musí byť také, aby som tých y domov vedel optimálne pokryť na $Z[x] - 1$ zastávok. Vyskúšam všetky také y a vyberiem to, ktoré mi dá najlepšie riešenie.

Dynamickým programovaním vieme hodnoty v poli C spočítať v čase $O(n^2)$.

Lepšie predpočítanie

Jediné, čo ešte stojí medzi nami a kvadratickým riešením, je predpočítanie hodnôt v poli S. Pre každý úsek $\{i, \ldots, j\}$, ktorý sa dá pokryť jednou zastávkou, potrebujeme nájsť optimálny súčet vzdialeností pri takomto pokrytí.

Zvoľme si pevne i. Budeme postupne zväčšovať j, posúvať zastávku a pamätať si aktuálny súčet vzdialeností už spracovaných domov od nej.

Spracovať prvý dom (teda $j = i$) je ľahké: zastávka stojí pri ňom a súčet vzdialeností je nula. Všimnime si, že už teraz je určený koniec intervalu, v ktorom môže ležať naša zastávka: ten je na súradnici $X[i]+d$, kde $X[i]$ je súradnica i-teho domu. Tento koniec zostane nemenný, zatiaľ čo začiatok povoleného intervalu sa bude posúvať doprava – po spracovaní domu j bude na súradnici $X[j] - d$.

Ako postupne zvyšujeme j, rastie aj medián súradníc domov v spracúvanom úseku. Optimálna poloha zastávky sa teda posúva doprava, nikdy nie doľava. (Ak sa ľavá hranica úseku, kde smieme mať zastávku, posúva rýchlejšie ako medián, môže sa nám stať, že nám aj tá zastávku „potlačí“ doprava.)

Niekedy sa nám môže stať, že zastávku potrebujeme posunúť doprava výrazne, okolo niekoľkých domov. Spočítať, ako sa zmení súčet vzdialeností počas takéhoto posunu, môže byť problém. Preto každý takýto posun rozbijeme na viacero posunov „o dom ďalej“. No a posun o jeden dom ľahko odsimulujeme v konštantnom čase – vieme aj vzdialenosť, o ktorú zastávku posúvame, aj to, od koľkých domov ide ďalej a ku koľkým bližšie.

Budeme teda mať dva typy krokov, ktoré budeme vhodne striedať: „zväčši j o 1“ a „posuň zastávku k nasledujúcemu domu (alebo o menej ako dom, na hranicu povoleného intervalu)“. Každý krok vieme odsimulovať v konštantnom čase. Aj zväčšení j, aj posunov zastávky bude dokopy $O(n)$. Toto spravíme zvlášť pre každé i, čím dostávame predpočítanie, a teda aj celý algoritmus, v časovej zložitosti $O(n^2)$.

Na záver už len jemne provokačná otázka: A nešlo by to ešte lepšie? :-)

Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

8. Opozícia na úteku

opravoval Usamec
(max. 25 bodov)

Úvodom pár slov pre riešiteľov. Prišli mi v podstate tri druhy riešení. Vzorové, nefunkčné a nedokončené. Ja ale verím tomu, že mnoho z vás bolo schopných vymyslieť funkčné, hoci trochu pomalšie riešenie. Nebojte sa takéto riešenia posielať. Pokiaľ fungujú, určite za ne dostanete nie malé množstvo bodov.

Budeme sa baviť v grafovej terminológii. Cesty budú hrany grafu a križovatky vrcholy. Číslom N označujeme počet vrcholov, číslom M počet hrán.

Riešenie tejto úlohy bude pozostávať z dvoch krokov. Najprv nájdeme všetky vrcholy a hrany, cez ktoré vôbec môže viesť najkratšia cesta, a následne budeme hľadať vrchol, ktorý keď odstránime, tak zrušíme všetky najkratšie cesty (uvedomte si, že toto je to isté, ako hľadať vrchol, cez ktorý idú všetky najkratšie cesty).

Hľadanie najkratšej cesty – Dijkstrov algoritmus

Popis tohoto štandardného algoritmu môžete nájsť skoro v každej lepšej knihe o algoritmoch, na Wikipédii a v Liahni. Tento algoritmus nájde pre každý vrchol najkratšiu vzdialenosť a cestu od daného začiatočného vrcholu. Zhrňme si jeho fungovanie: Pre každý vrchol si pamätáme zatiaľ najlepšiu nájdenú vzdialenosť od začiatku. Na začiatku má začiatočný vrchol vzdialenosť 0, ostatné nekonečno (v praxi buď dostatočne veľké číslo alebo -1). Na začiatku sú všetky vrcholy v množine otvorených vrcholov. V každom kroku potom vyberieme otvorený vrchol s najmenšou nájdenou vzdialenosťou, vylepšíme vzdialenosti jeho susedov a tento vrchol prehlásime za uzatvorený.

Takto nájdeme pre každý vrchol jeho najkratšiu vzdialenosť od začiatku. Zároveň vďaka tomu, že všetky hrany majú nezápornú dĺžku, si môžeme byť istí, že keď prehlasujeme nejaký vrchol za uzavretý, tak sa doňho už kratšou cestou dostať nevieme.

Pre naše potreby si algoritmus potrebujeme mierne modifikovať. Chceme nájsť všetky hrany, cez ktoré vedie nejaká najkratšia cesta. Všimnite si, že síce pôvodné hrany boli neorientované, ale tieto nové hrany už budú orientované, lebo pri najkratšej ceste sa nám po hrane oplatí hýbať iba jedným smerom. Označme $s(x)$ ako vzdialenosť vrcholu x od začiatku, $k(x)$ vzdialenosť vrcholu x od konca a $c(x, y)$ dĺžku hrany medzi vrcholmi x, y. Hodnoty $s(x)$ a $k(x)$ vieme získať dvakrát pusteným Dijkstrovým algoritmom. Ak je súčet $s(x) + c(x, y) + k(y)$ rovný dĺžke najkratšej cesty zo začiatočného vrcholu do koncového, tak hrana z x do y leží na nejakej najkratšej ceste. Takto vieme ľahko o každej hrane rozhodnúť, či leží na nejakej najkratšej ceste.

Iná možnosť: Keď vylepšujeme nájdenú vzdialenosť do vrcholu, tak si zapamätáme, odkiaľ sme tam prišli. Keď je nájdená vzdialenosť rovnaká, tak si zapamätáme ďalšiu hranu. Takto máme potom pre každý vrchol zoznam hrán, odkiaľ sa doňho dá prísť po najkratšej ceste. Následne pustíme prehľadávanie z koncového vrcholu, aby sme vybrali iba tie vrcholy a hrany, ktoré vedú ku koncovému vrcholu.

Implementačné poznámky. Pokiaľ si budeme vrcholy udržiavať v poli, tak nájdenie najbližšieho vrcholu nám trvá čas $O(N)$, čiže náš celkový čas bude $O(N^2 + M)$. Ale môžeme si vrcholy pamätať v halde. Potom vieme vybrať najbližší vrchol a upraviť jeho vzdialenosť v čase $O(\log N)$.[8] Takúto haldu si môžeme napísať sami, alebo použiť v C++ `set`.

Existuje ale ešte lenivejšia metóda. Daný vrchol vložíme do haldy vždy, keď nejak upravíme jeho vzdialenosť (čiže teoreticky táto halda bude mať veľkosť $O(M)$). A keď vrchol vyberáme z haldy, tak skontrolujeme, či sme ho už náhodou predtým neprehlásili za uzavretý. Takto nám stačí vedieť prvky do haldy vkladať a vyberať, nepotrebujeme meniť hodnotu prvku, čiže nám stačí v C++ `priority_queue`.

[8] Fibonacciho halda vie upraviť vzdialenosť v čase $O(1)$, potom máme celkový čas algoritmu $O(N \log N + M)$, ale v praxi je táto halda pomalšia ako normálna.

Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

Každopádne, nech použijeme hociktorý prístup, dosiahli sme časovú zložitosť $O((N + M) \log N)$ a pamäťovú $O(N + M)$. A výstupom je graf, v ktorom je každá cesta zo začiatočného do koncového vrcholu najkratšia.

Triviálne riešenia

Keď už vieme hľadať najkratšiu cestu, môžeme napísať dve pomerne triviálne riešenia. Prvé využíva pôvodný (neupravený) Dijkstrov algoritmus: najprv nájde dĺžku najkratšej cesty medzi začiatkom a koncom. Následne skúsi odobrať každý vrchol a pre každé odobratie opäť zistí dĺžku najkratšej cesty medzi začiatkom a koncom. Ak sa táto dĺžka zmení, tak tento vrchol zrušil všetky najkratšie cesty v pôvodnom grafe. Takto máme časovú zložitosť $O(N(N + M) \log N)$.

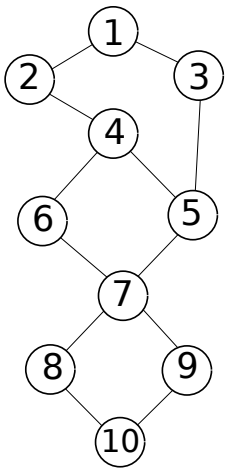
Trochu lepším riešením je graf osekať len na graf obsahujúci najkratšie cesty a až z neho skúšať odoberať vrcholy. Potom vieme jedným prehľadávaním (BFS či DFS) zistiť, či tento osekaný graf je stále súvislý. Takto dosiahneme zložitosť $O(N(N + M) + (N + M) \log N) = O(N(N + M))$.

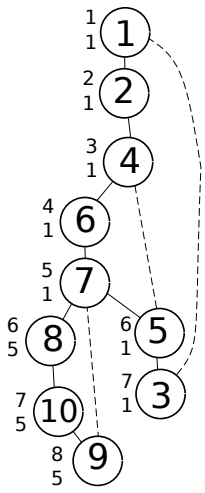
Hľadanie artikulácií

Artikuláciou voláme vrchol, ktorý keď odstránime z grafu, tak sa graf stane nesúvislým (predpokladáme, že predtým bol súvislý). Vieme ju hľadať v čase $O(N + M)$. Toto nám celkom stačí. Ak v osekanom grafe nájdeme artikuláciu, tak určite vieme pomocou nej zrušiť všetky najkratšie cesty. (Rozmyslite si, že to platí aj naopak: každý vrchol, ktorým vieme prerušiť všetky najkratšie cesty v pôvodnom grafe, je artikuláciou v osekanom grafe.)

Artikulácie bežne hľadáme v neorientovaných grafoch. V našom prípade ale platí, že vrchol je artikuláciou v neorientovanej verzii grafu práve vtedy, keď je ňou v orientovanej verzii (dôkaz jedným smerom je triviálny; opačným smerom využijeme acyklickosť nášho grafu a fakt, že každý vrchol leží na nejakej ceste zo začiatočného do koncového vrcholu).

Odteraz preto zabudnime na orientáciu hrán. Algoritmus je nasledovný: Pustíme prehľadávanie do hĺbky z nejakého vrcholu. Takto nám vznikne DFS strom (pozri obrázok grafu a jeho DFS stromu pri spustení z vrcholu 1) obsahujúci dva druhy hrán: stromové a spätné. Stromové hrany sú tie, ktorými DFS prechádza do ešte nenavštívených vrcholov; spätné hrany (znázornené prerušovanou čiarou) vedú do navštívených vrcholov.





Pre každý vrchol x si zapamätáme jeho hĺbku $h(x)$ a najmenšiu hĺbku $u(x)$, do ktorej sa vieme dostať, ak pôjdeme z neho nadol po niekoľkých stromových hranách a nahor po jednej spätnej (na obrázku je horné číslo $h(x)$ a dolné $u(x)$). Hodnotu $u(x)$ si vieme pri prehľadávaní do hĺbky zistiť ľahko, je to minimum zo synovských hodnôt u a hĺbok vrcholov, kam sa vieme dostať priamo z x spätnými hranami.

Vrchol x je artikuláciou, keď existuje nejaký jeho syn y, pre ktorého platí $u(y) \leq h(x)$, čiže sa nevie dostať nad vrchol x. Na obrázku je takým vrcholom 7, lebo jeho syn 8 sa nevie dostať nad neho. Výnimkou pri tomto pravidle je koreň stromu. Ten je artikuláciou len vtedy, keď má viacerých synov.

Poznamenajme, že skoro rovnakým spôsobom vieme hľadať mosty (hrany, po ktorých odobratí prestane byť graf súvislý). Stromová hrana vchádzajúca do y je mostom, ak $u(y) \leq d(y)$, čiže nemáme inú možnosť, ako obísť túto hranu (spätná hrana mostom byť nemôže).

Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

Zametanie

Iný prístup, ktorý nám vyrieši náš problém, je síce pomalší ako hľadanie artikulácií, ale nič nám nepokazí, lebo nie je pomalší ako Dijkstrov algoritmus. Predstavme si, že máme nekonečne veľa zbojníkov nahromadených v štartovacom vrchole. A teraz im povieme, aby sa posunuli o 1 jednotku po všetkých najkratších cestách k cieľovému vrcholu (zbojníci sa rozdelia, ak to bude potrebné; na každej ceste ich ale stále zostane nekonečne veľa). Takto si ich posunieme až k cieľovému vrcholu. A kedykoľvek sa nám počas cesty všetci zbojníci znova nahromadili v jednom vrchole, tak tento vrchol určite zruší všetky najkratšie cesty.

Na efektívnu implementáciu si potrebujeme uvedomiť dve veci. Prvou je, že si nám stačí pamätať počet skupín zbojníkov (a vrchol ruší najkratšie cesty, ak počas prechodu ním máme iba jednu skupinu zbojníkov). Navyše, počet skupín sa nám mení iba vo vrcholoch. Takže si najprv utriedime vrcholy v osekanom grafe podľa vzdialenosti od začiatku a spracúvame ich v tomto poradí.

Spočítame si, koľko hrán do týchto vrcholov vstupuje a koľko vystupuje. Ak do vrcholu vstupuje X hrán, teda X skupín zbojníkov, tak sa z nich stane jedna skupina – počet skupín sa zmenší o $X - 1$. Ak z vrcholu vychádza Y hrán, tak sa z jednej skupiny stane Y skupín, čiže počet skupín sa zväčší o $Y - 1$. Takže pri spracovaní vrcholu najprv patrične znížime počet skupín, potom skontrolujeme, či náhodou nie je rovný jednej a potom ho zväčšíme. Uvedomte si, že je nám úplne jedno, v akom poradí spracúvame vrcholy, ktorých vzdialenosť od začiatku je rovnaká. Totiž pre tieto vrcholy nikdy počet skupín nemôže klesnúť na 1. Zložitosť tohoto postupu je $O(N \log N + M)$.

Celkovo vzorové riešenie má časovú zložitosť $O((N + M) \log N)$ a pamäťovú $O(N + M)$.

Listing programu:

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

#define INF 2000000000
int N, M;

struct Vrchol {
  vector<pair<int, int> > next;   // susedia - vrchol, cena
  int d[2];                        // najlepsia najdena vzdialenost
                                   // 0 - od startu, 1 - od ciela
  int in, out;                     // pocet vchadzajucich a vychadzajucich
                                   // hran v osekanom grafe
  Vrchol() {
    d[0] = d[1] = INF;
    in = out = 0;
  }
};

vector<Vrchol> v;

// Nechceme kopirovat kod, napiseme si proceduru
void dijkstra(int start, int index) {
  // Vo fronte mame dvojicu (vzdialenost, index)
  priority_queue<pair<int, int>, vector<pair<int, int> >, greater<pair<int, int> > > fr;
  fr.push(make_pair(0, start));
  while (!fr.empty()) {
    int dist = fr.top().first;
    int x = fr.top().second;
    fr.pop();
```


Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

```
    if (v[x].d[index] <= dist) continue; // Skontrolujeme, ci sme vrchol uz neuzavreli
    v[x].d[index] = dist;
    for (int i = 0; i < v[x].next.size(); i++) {
      fr.push(make_pair(dist + v[x].next[i].second, v[x].next[i].first));
    }
  }
}

int main() {
  scanf("%d %d", &N, &M);
  v.resize(N);
  for (int i = 0; i < M; i++) {
    int a, b, c;
    scanf("%d %d %d", &a, &b, &c);
    a--; b--;
    v[a].next.push_back(make_pair(b, c));
    v[b].next.push_back(make_pair(a, c));
  }
  int start, end;
  scanf("%d %d", &start, &end);
  start--; end--;
  dijkstra(start, 0);
  dijkstra(end, 1);

  // Pre kazdu hranu ideme zistit, ci cez nu ide najkratsia cesta
  // Zaroven upravime pocty hran vchadzajucich a vychadzajucich z vrcholu
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < v[i].next.size(); j++) {
      if (v[i].d[0] + v[i].next[j].second + v[v[i].next[j].first].d[1]
          == v[end].d[0]) {
        v[i].out++;
        v[v[i].next[j].first].in++;
      }
    }
  }
  vector<pair<int, int> > used_v; // Vrcholy cez ktore nieco ide
                                  // (vzdialenost od zaciatku, index)
  for (int i = 0; i < N; i++) {
    if (v[i].out != 0 || v[i].in != 0)
      used_v.push_back(make_pair(v[i].d[0], i));
  }
  sort(used_v.begin(), used_v.end());
  int groups = 0;
  int out = -1;
  for (int i = 0; i < used_v.size(); i++) {
    groups -= (v[used_v[i].second].in - 1);
    if (groups == 1 && used_v[i].second != start && used_v[i].second != end) {
      out = used_v[i].second;
      break;
    }
    groups += (v[used_v[i].second].out - 1);
  }
  if (out != -1)
    printf("%d\n", out + 1);
  else
    printf("-1\n");
}
```


Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09