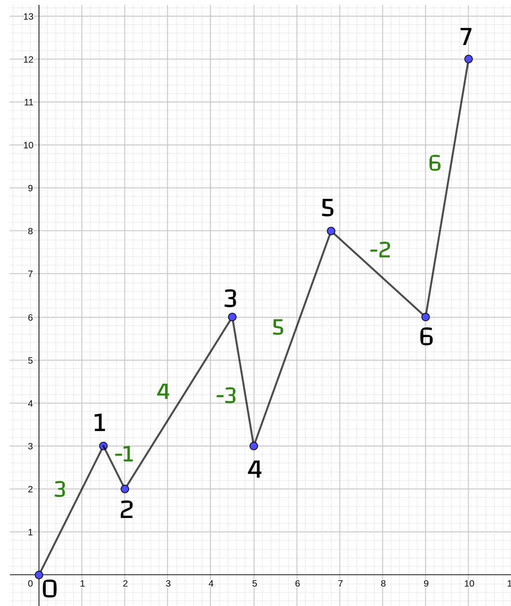
PRASK
V. ročník, 2018/19
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

Vzorové riešenia 1. kola letnej časti

1. Prefixová expedícia

vzorák napísal(a) Roman
(max. 15 b za riešenie)

Táto úloha bola venovaná prefixovým súčtom, ktoré sú neoddeliteľnou súčasťou výbavy programátora. Pochopenie vzorového riešenia vám určite pomôže pri riešení mnohých úloh v budúcnosti.



A	3	-1	4	-3	5	-2	6

P	0	3	2	6	3	8	6	12

Podúloha A

Vašou úlohou bolo nájsť všetky vrcholy k také, že

$$A[0] + \cdots + A[k-1] = A[k] + \cdots + A[N-1]$$

V zadaní bolo spomenuté, že pole prefixových súčtov P obsahuje na i-tej pozícii súčet všetkých čísel, ktoré sú v pôvodnej postupnosti na pozíciách od 0 po $i-1$. To je vlastne prvý súčet z rovnosti vyššie. Teda pre každý vrchol k platí, že

$$P[k] = A[0] + \cdots + A[k-1]$$

Napríklad pre vrchol 4 je súčet $A[0] + A[1] + A[2] + A[3]$ rovný 3, a preto aj hodnota $P[4]$ je 3.

Všimnite si, že súčet na druhej strane rovnosti, $A[k] + \cdots + A[N-1]$, je rovný celkovému súčtu prvkov poľa A bez ľavej strany z pôvodnej rovnosti. Formálne

$$A[k] + \cdots + A[N-1] = (A[0] + \cdots + A[N-1]) - (A[0] + \cdots + A[k-1])$$

Posledné pozorovanie je, že celkový súčet poľa A nájdeme v poli prefixových súčtov na poslednom, N-tom mieste.

Aby k bolo stredovým vrcholom, musí preto platiť

$$P[k] = P[N] - P[k]$$

Pri riešení teda stačí prejsť v jednom cykle postupne všetky možné k, a pri každom overiť podmienku uvedenú vyššie.

```
pre k := 0 do N:
    ak (P[k] == P[N] - P[k]):
        vypis(k + ' je stredovy vrchol')
```

Pri každom vrchole sa pozrieme na 3 hodnoty poľa $P[]$, spravíme jedno porovnanie a prípade niečo vypíšeme. Celkovo teda pre jeden vrchol vykonáme *konštantne* veľa operácii. To znamená, že počet operácií pre jeden vrchol nezávisí od vstupných hodnôt, napríklad od hodnoty N. Vrcholov máme $N+1$, takže program vykoná $3 \cdot (N+1)$ prístupov k prvkom poľa P. To je lineárne veľa v závislosti od N, a preto má naše riešenie časovú zložitosť $O(N)$. Uvedomte si, že rýchlejšie riešenie nevymyslíme, pretože v najhoršom prípade by každý z vrcholov bol stredový (samé hodnoty 0) a v takom prípade by sme museli vypísať všetkých $N+1$ vrcholov. Operácia vypísania $N+1$ prvkov má zložitosť $O(N)$, takže aj keby sme našli všetky stredové vrcholy rýchlejšie ako v $O(N)$, ich výpis nám zaberie $O(N)$ operácii.

Podúloha B

Kedy je úsek medzi dvoma vrcholmi a a b ($a < b$) rovný 0? V zadanej úlohe túto podmienku spĺňajú napr. vrcholy 1 a 4 a tiež 34 a 6. Zjavne musí platiť

$$A[a] + A[a+1] + \cdots + A[b-1] = 0$$

Súčet na ľavej strane môžeme dostať aj tak, že odčítáme súčet prvkov po $A[a-1]$ od súčtu po $A[b-1]$. Teda

$$A[a] + A[a+1] + \cdots + A[b-1] = (A[0] + \cdots + A[b-1]) - (A[0] + \cdots + A[a-1])$$

Na pravej strane dostávame prefixové súčty $P[b]$ a $P[a]$ takže skrátene

$$A[a] + A[a+1] + \cdots + A[b-1] = P[b] - P[a]$$

Tu zároveň dostávame riešenie nasledujúcej podúlohy, ktorá sa pýta, ako vypočítať súčet úseku od a po $b-1$. Pokračujme ale ďalej. My teraz chceme, aby bol súčet medzi a a b rovný 0. Preto môžeme do rovnice vyššie namiesto všeobecného súčtu napísať práve 0, ktorú chceme dostať.

$$0 = P[b] - P[a]$$

Odtiaľ vidíme, že tento prípad nastane práve vtedy, keď $P[a] = P[b]$. Preto môžeme povedať, že súčet medzi vrcholmi a a b je rovný 0 ak sa prefixové súčty $P[a]$ a $P[b]$ rovnajú.

Podúloha C

Ako bolo avizované v predošlom riešení, súčet $A[a]+A[a+1]+\cdots+A[b-1]$ pre $a < b$ vieme získať pomocou prefixových súčtov ako rozdiel $P[b]$ a $P[a]$.

$$A[a] + A[a+1] + \cdots + A[b-1] = P[b] - P[a]$$

Ak už máme vypočítané prefixové súčty, stačí nám jediná aritmetická operácia na vypočítanie celého súčtu úseku. Namiesto postupného pričítavania hodnôt medzi a a $b-1$, ktorých by mohlo byť až N máme zakaždým zaručené iba jedno odčítanie. Takéto riešenie je teda oveľa rýchlejšie.

Podúloha D

Ako prvé riešenie by nám mohlo napadnúť i-tu pozíciu P spočítať tak, že spočítame všetky prvky od 0 po $i-1$, teda

```
pre i := 0 do N:
    sucet := 0
    pre j := 0 do i - 1:
        sucet := sucet + A[j]

    P[i] := sucet
```

Toto je korektné riešenie, ale zbytočne pomalé. V našom programe $P[i]$ počítame tak, že sčítame postupne všetky prvky $A[0], \dots, A[i-1]$. Potom pre $i+1$ znovu sčítavame postupne $A[0], \dots, A[i-1], A[i]$. Všimnite si, že medzi členmi i a $i+1$ poľa P nie je veľký rozdiel, konkrétne $P[i+1]$ obsahuje oproti $P[i]$ hodnotu $A[i]$ navyše. Veľa roboty tak robíme opakovane. Tento fakt vieme využiť nasledovne:

```
P[0] := 0
pre i := 0 do N - 1:
    P[i + 1] := P[i] + A[i]
```

Je jasné, že druhé riešenie je výrazne rýchlejšie, prvý cyklus je totiž rovnaký, v prvom v prípade sa v ňom však opakujeme, v druhom už nie. Toto zlepšenie je však výrazné aj v časovej zložitosti. Keď sa pozrieme na počet sčítaní, ktoré musíme spraviť, v prvom prípade to bude

$$1 + 2 + 3 + \cdots + N = \frac{N(N+1)}{2}$$

, pretože v prvom opakovaní spravíme jedno sčítanie, v druhom dve, atď[1]. Takýto počet operácií potom zaokrúhlime na N^2 a povieme, že časová zložitosť je $O(N^2)$.

Druhé riešenie však spraví iba N sčítaní a jeho časová zložiosť je preto iba $O(N)$, čo je opäť raz optimálne, keďže hodnoty poľa A musíme v čase $O(N)$ načítať.

Podúloha E

Táto podúloha má aj svoju ťažšiu verziu. Jej zadanie nájdeš na konci vzoráku.

V tejto podúlohe bolo vašou úlohou vypísať všetky dvojice (a, b), pre ktoré platí $P[b] - P[a] = k$. Nasledujúce riešenie prejde v cykle všetky dvojice (i, j) také, že $i < j$ a overí podmienku zo zadania.

```
pre i := 0 do N:
    pre j := i + 1 do N:
        ak (P[j] - P[i] == k):
            vypis(i + ' ' + j)
```

Všetkých dvojíc je spolu $\frac{N(N+1)}{2}$. Prečo? Predstavte si, že pre vrchol 0 vyberáme druhý vrchol do dvojice. Máme N možností $(1, 2, \dots, N)$. Ak pre vrchol 1 vyberáme druhý vrchol do dvojice, máme $N-1$ možnosti $(2, 3, \dots, N)$. Ak by sme takto pokračovali ďalej, dostaneme spolu počet všetkých dvojíc

$$N + (N-1) + \cdots + 2 + 1 + 0$$

Ak tento súčet otočíme, je to opäť súčet prvých N prirodzených čísel, s ktorým sme sa už stretli v podúlohe D. V poznámke pod vzorákom sa dozviete prečo sa tento súčet rovná $\frac{N(N+1)}{2}$.

V podúlohe D sme tiež zistili, že tento počet rastie kvadraticky v závislosti na N. To znamená, že naše riešenie má kvadratickú časovú zložitosť $O(N^2)$, pretože každú dvojicu prejde práve raz a spraví pre ňu konštantne veľa operácií (pristúpi k 2 prvkom poľa P, vyhodnotí podmienku a prípadne vypíše hlášku na výstup). Uvedomte si, že lepšie riešenie nevymyslíme, pretože v najhoršom prípade by každá z dvojíc vrcholov mohla mať medzi sebou rozdiel k a všetky by sme ich tak museli vypísať.

Ťažšia verzia tejto úlohy sa nepýta na všetky dvojice s rozdielom k, ale iba na ich **počet**. Výstupom tak bude už iba jedno číslo, čiže časová zložitosť riešenia nie je limitovaná veľkosťou výstupu ako v pôvodnej podúlohe. Prídete na to ako efektívne vyriešiť túto úlohu?

[1] Pre súčet prvých N prirodzených čísel platí $1 + 2 + \dots + N = \frac{N(N+1)}{2}$. Skúsme pre párne N sčítať prvé číslo s posledným, druhé s predposledným, atď. Výsledok bude zakaždým $N+1$. Dvojíc s takýmto súčtom sme vytvorili N, a preto je celkový výsledok $\frac{N(N+1)}{2}$. Skúste si rozmyslieť podobnou úvahou, prečo vzorec platí aj pre nepárne N.

2. Rafinované vyzvedanie

vzorák napísal(a) Timka
(max. 15 b za riešenie)

Podúloha a.

Lucka vie svoje otázky prispôsobovať Evkiným odpovediam.

Asi najjednoduchšie by bolo pýtať sa na jednotlivé dátumy pekne zaradom. Narodila sa Evka prvého? Druhého? Tretieho? Takto sa však pri najhoršom spýtame 30 otázok, čo je pomerne veľa.

Prvým zlepšením by mohlo byť pýtanie sa na deň deň v týždni, v ktorom sa Evka narodila. Bol to pondelok? Utorok? V najhoršom prípade by sa nám však mohlo stať, že položíme Evke šesť otázok na zistenie dňa v týždni a potom ešte 4 otázky na zistenie presného dátumu, keďže v jednom mesiaci môže byť najviac päť rovnakých dní. Lucke teda stačí 10 otázok.

Keď sme v predošlej úvahe rozmýšľali nad dňami v týždni, každou otázkou sme vylúčili jednu sedminu možností. Čo keby sme vylúčili nejakú väčšiu časť? Napríklad rovno polovicu. Prvou otázkou sa spýtame, či má narodeniny v prvej polke mesiaca. To nám hneď vylúči polovicu dní. Ak Evka odpovie áno tak sa jej spýtame či ich má v prvej štvrťke. Naopak ak Evka odpovie nie, tak sa jej spýtame, či má narodeniny v Tretej štvrtine mesiaca. Podľa Evkinej odpovede prispôsobíme a vyberieme jednu z otázok: či sa narodila v prvej, tretej, piatej alebo siedmej osmine mesiaca. Rovnako sa potom spýtame na šestnástiny a neskôr aj na 32-tiny.

Po piatich otázkach sa dozvieme presnú 32-tinu mesiaca, v ktorej má Evka narodeniny, teda konkrétny deň.

Podúloha b.

V tejto podúlohe už Lucka svoje otázky prispôsobovať nevie, ale musí ich napísať všetky dopredu.

V predošlej podúlohe sa nám podarilo zistiť dátum Evkiných narodenín na 5 otázok, skúsme teda otázky trochu poupraviť tak, aby sme ich mohli použiť aj v tejto podúlohe.

Prvá otázka ostane rovnaká, ak chceme zistiť, v ktorej polovici sa nachádza dátum narodením, musíme sa spýtať otázku "Narodila si sa medzi 1. až 15. dňom mesiaca?".

Druhá otázka sa však už líšila podľa prvej odpovede. Buď sme sa pýtali na prvú alebo tretiu štvrtinu. Skúsme sa preto spýtať iba prvú z nich: "Narodila si sa medzi 1. až 7. dňom mesiaca?". Ak dostaneme odpoveď áno, štvrtinu máme určenú. V prípade nie sa vieme pozrieť na prvú odpoveď. Ak nám totiž napísala odpovede "áno" (v prvej polovici), "nie" (v prvej štvrtine), musela sa narodiť v druhej štvrtine. Problémom je však tretia a štvrtá štvrtina.

Skúsme sa spýtať komplikovanejšiu druhú otázku: "Narodila si sa buď medzi 1. až 7. dňom **alebo** medzi 16. až 23. dňom?". Z odpovede "áno" nevieme zistiť, v ktorej z týchto štvrtín to bolo. Ak však zoberieme do úvahy aj odpoveď na prvú otázku, polovicu, odpoveď je jasná. Na tieto dve otázky sme mohli dostať štyri možné kombinácie odpovedí a to je aj počet štvrtín, ktoré určujú.

Zvyšné otázky potom vieme skonštruovať ľahko. Treťou sa spýtame, či sa narodila v prvej, tretej, piatej alebo siedmej osmine mesiaca. Opäť, spolu s odpoveďami na predchádzajúce určíme konkrétnu osminu. Toto isté ešte spravíme pre 16-tiny a 32-tiny.

Týchto 5 otázok môže Lucka položiť Evke naraz a z odpovedí vie určiť presný dátum narodenia.

Okrem toho existuje aj mierne jednoduchšia sada otázok, aj keď o niečo trikovejšia. Vyzerá takto:

- Dáva číslo tvojho dňa narodenia zvyšok najviac 15 po delení 32? (polovica)
- Dáva číslo tvojho dňa narodenia zvyšok najviac 7 po delení 16? (štvrtina)
- Dáva číslo tvojho dňa narodenia zvyšok najviac 3 po delení 8? (osmina)
- Dáva číslo tvojho dňa narodenia zvyšok najviac 1 po delení 4? (16-tina)
- Dáva číslo tvojho dňa narodenia zvyšok 0 po delení 2? (32-tina)

Rozmyslite si, prečo tieto otázky fungujú.

Podúloha c.

Označme si náš počet otázok n. Chceme aby sme po ľubovoľnej kombinácii Evkiných odpovedí vedeli presne povedať, kedy má narodeniny. To znamená, že každá možná kombinácia, nám musí presne určovať jeden dátum. Teda najviac rôznych dátumov vieme našimi n otázkami pokryť, keď každá možná kombinácia odpovedí nás dovedie k inému dátumu.

Keďže na každú otázku máme dve možnosti odpovede, tak každou ďalšou otázkou sa nám zdvojnásobí počet možných kombinácií odpovedí na jednotlivé otázky. Takže vlastne pre n otázok môžeme mať najviac 2^n možných kombinácií odpovedí.

Na to, aby sme vedeli určiť presný dátum narodenia musí platiť, že $2^n \geq 31$, teda že vieme rozlíšiť každý možný dátum. A najmenšie n, ktoré toto spĺňa je práve $n = 5$. Z toho vyplýva, že 5 otázok je skutočne optimálny počet.

vzorák napísal(a) Andrej
(max. 0 b za riešenie)

3. Akútny upgrade

Táto úloha bola už tradične špeciálna. V tomto prípade sme vám ukázali nový model počítača a nejaký nový spôsob programovania. O tom, ako blízko realite tento model je si môžeme porozprávať v ďalšom vzoráku. Ako prvý krok k vyriešeniu úloh bolo dobré pozrieť si nejaké príklady programov a pochopiť čo asi robia.

Podúloha A

Toto bola úloha, ktorá mala preveriť, či chápete ako model funguje. Na jej vyriešenie stačilo iba do rúry pridať číslo 3 a 5. Po operácii `mul` sa potom na koniec rúry vložil násobok čísla, ktoré tam bolo a druhého čísla v rúre, teda 3. Následným príkazom `add` sa odtiaľ toto číslo vzalo, sčítalo s 5 a vložilo na koniec rúry. Print už len vypísal jediné číslo v rúre – výsledok.

```
put 3; put 5
mul; add;
print
```

Pokiaľ náhodou nerozumiete ako program funguje, kľudne si zapnite simulátor a odkrokujte si ho.

Podúloha B

Táto podúloha vyzerá, že k riešeniu potrebuje cyklus. Istú formu cyklu sme si ale ukazovali v manuáli k simulátoru, medzi príkladmi programov. Ten však funguje donekonečna a to sa nám až tak nepáči. Tiež nezvyšuje nejakú premennú, ktorú by sme asi potrebovali poznať. Budeme teda postupovať tak, že si načítame vstup do registra x a potom budeme zvyšovať v cykle číslo v registri i a to aj vypisovať.

```
get x
label cyklus
put i; put 1; add; get i
put i; print
jump cyklus
```

Toto už vyzerá celkom dobre až na to, že ide o nekonečný cyklus. Čo teda chceme urobiť? Pokiaľ sa v registri i nachádza to, čo registri x a už sme i vypísali, chceme skončiť. Ako skončiť? Skočíme jednoducho na nejaké návestie mimo cyklu a tam v pokoji skončíme.

Náš program bude vyzerať nejak takto:

```
get x

label cyklus
put i; put 1; add; get i
put i; print
jeq i x koniec
jump cyklus

label koniec
```

Podúloha C

Super, vytvorili sme si celkom pekný cyklus, ktorý by sa dal možno aj v budúcnosti použiť. Ak by sme boli požiadaní aby sme napísali riešenie pre túto úlohu v svojom obľúbenom programovacom jazyku, čo by sme urobili? Stačil by nám jeden for cyklus a dve premenné, v ktorých by sme si pamätali posledné a predposledné vypočítané fibonacciho číslo. V každom kroku cyklu by sme vypísali posledné číslo a obidve tieto premenné aktualizovali. Ak si tieto premenné označíme a a b, chceli by sme urobiť $a = b$ a $b = a + b$. Inak povedané, predposledné číslo bude mať hodnotu posledného a posledné bude súčtom týchto dvoch čísel. Takýto presun samozrejme nejde urobiť naraz a budeme na to potrebovať pomocnú premennú, to nám ale nevadí.

```
get n
put 1; get b

label cyklus
put i; put 1; add; get i

jeq i n koniec
put b; put a; put b; get a; add; get b
jump cyklus

label koniec
put a; print a
```

Pokiaľ program príliš nechápete, krokujte si ho a všímajte si najmä hodnoty v registroch i, a, b.

Podúloha D

Najjednoduchší spôsob ako zistiť počet deliteľov nejakého x je prejsť všetky čísla menšie rovné ako x a každé otestovať, teda zistiť či delí x.

```
get n

label cyklus
put i; put 1; add; get i
put n; put i; mod

get z ; jz z pridaj
label sem

jeq i n koniec
jump cyklus

label pridaj
put a; put 1; add; get a
jump sem

label koniec
put a; print a
```

Na návestie/label pridaj skočíme, pokiaľ sme v cykle zistili, že aktuálne i delí n. To sme zistili tak, že sme do rúry vhodili n a i a zistili, či sa zvyšok n po delení i rovná 0. Pokiaľ áno, skočíme na návestie pridaj a odtiaľ skočíme potom naspäť do cyklu.

Podúloha E

Tu budeme musieť zvoliť iný prístup. Problémom je, že nevieme použiť rúru. Ak tam niečo vložíme, ako vieme, že sme narazili na nejaké naše číslo a nie na čísla, ktoré tam boli už predtým? Zachráni nás iba to, že čísla na vstupe sú kladné. Pokiaľ si teda vložíme za tieto čísla nulu, vieme odhaliť, že veci za ňou sme si tam vložili my a nie sú súčasťou vstupu.

Pre každé číslo na vstupe bude stačiť, ak si za našu zarážku vložíme hodnotu 1. Tým sa nám úloha zmení a v tomto momente chceme sčítať všetky čísla v rúre. V rúre je niekoľko 1 a my chceme zistiť koľko. Zjavne nám stačí sčítavať čísla pokým to ide. Zavolať funkciu `add` však nemôžeme pokiaľ nemáme v rúre aspoň dve čísla. Preto musíme vždy skontrolovať či máme v rúre aspoň dve čísla, ak áno, tak môžeme zavolať `add`. Pokiaľ nie, tak číslo, ktoré je v rúre jediné je náš želaný súčet.

```
put 0
label cyklus1
get a; jz a spocitaj
put 1
jump cyklus1
```

```
label spocitaj
get a;
jempty koniec
get b; put a; put b; add
jump spocitaj

label koniec
put a; print
```

Po prvom cykle, ktorý skončí keď narazí na 0 je iba nejaký počet jednotiek. Za každé číslo na vstupe je tam teraz jedna jednotka. V cykle spočítaj teda už iba "bezpečne" sčítavame, pokým nám neostane v rúre iba jedno číslo. Bezpečne v tom zmysle, že keby len voláme dookola `add`, môže sa nám stať, že v rúre budeme mať už iba jedno číslo a náš program spadne. Preto pred každým sčítaním otestujeme, či sú v rúre aspoň dve čísla. Na to použijeme `jempty`.

Podúloha F

Táto podúloha sa celkom podobá na podúlohu E a použijeme v nej tiež podobný trik s 0 ako zarážkou. Táto podúloha však vyžaduje trochu väčšiu kreativitu pri riešení. Čo si môžeme chcieť dať za zarážku? Najvhodnejšie by bolo asi zvyšky po delení jednotlivých čísel 3. Na to však nevieme vhodne použiť operáciu `mod`. Z dôvodu, že tej bude chýbať druhý parameter. Prvé prejdenie čísel bude teda musieť byť akási predpríprava na toto. Za každé číslo si vhodíme do rúry toto číslo a 3. To nám umožní v ďalšom prechode hádzať za zarážku rovno zvyšok po delení 3. Pokiaľ rúra vyzerala: `10 15 7 3 8`, bude po takomto preiterovaní vyzerať takto: `10 3 15 3 7 3 3 3 8 3`. Všetky tieto čísla sú kladné, teda v ďalšom prechode môžeme znova 0 použiť ako zarážku.

Tu nás však PRAStarý Kalkulátor sklame. On totiž nevie zistiť, či je na začiatku rúry 0. Predtým musí túto 0 načítať. Je teda jasné, že pri prechode týchto čísel budeme musieť vždy najprv číslo načítať, zistiť či nie je 0 a potom až použiť `mod`. Tomu však bude chýbať prvý parameter, ktorý sme práve využili na test, či to nie je naša zarážka. Z tohto dôvodu sa nám oplatí si do rúry vkladať pri prvom prechode samotné číslo 2 krát. Rúra teda bude vyzerať po našom zamyslení takto: `10 10 3 15 15 3 7 7 3 3 3 3 8 8 3`. Teraz už vieme opakovať postup načítaj číslo, ak to je 0, skonči. Ak to nie je 0, zavolaj funkciu `mod`. Po tomto prechode máme v rúre akékoľvek čísla, ktoré reprezentujú zvyšky jednotlivých čísel po delení 3. Je zjavné, že nám už stačí len odpočítať počet 0. To si zjednodušíme ešte tým, že si okrem zvyšku po delení budeme do rúry vkladať aj dané číslo. Aby sa nám znovu dala 0 použiť ako zarážka. V predposlednom prechode nám teda rúra bude vyzerať takto: `10 1 15 0 7 1 3 0 8 2`. Po tomto už len stačí rúru prejsť a za 0, ktorú môžeme opäť použiť ako zarážku, si za každý zvyšok 0 hodiť 1. Vtedy sa nám problém zmení znova na spočítanie jednotiek v rúre, čo sme vyriešili už v predchádzajúcej úlohe.

```
put 0
label cyklus1
get a; jz a koniec1
put a; put a; put 3
jump cyklus1

label koniec1

put 0
label cyklus2
get a; jz a koniec2
put a; mod
jump cyklus2

label koniec2

put 0
label cyklus3
get a; jz a koniec3
get z; jz z pridaj
jump cyklus3
```

```
label pridaj
put 1
jump cyklus3

label koniec3

label spocitaj
get k;
jempty koniec
get l; put k; put l; add
jump spocitaj

label koniec
put k; print
```

Tým, že sa program skladá z viacerých častí, odporúčame si po jednotlivých častiach vložiť príkaz end. Tým ľahko zistíte, ako vyzerá rúra po jednotlivých väčších blokoch kódu.

4. Slizká pochúťka

vzorák napísal(a) Dano
(max. 15 b za riešenie)

V tejto úlohe sme dostali q úsekov dážďovky, ktoré máme natrieť omáčkou. Mali sme určiť, ako bude dážďovka vyzerať na konci. Chceli sme teda pre každé políčko dážďovky zistiť, či bude, alebo nebude natreté.

Hrubá sila

Spraviť bruteforce na túto úlohu je pomerne priamočiare. Majme pole dĺžky n so samými nulami. Nazvime ho `d[]` a bude reprezentovať, neprekvapivo, dážďovku. Hodnota 0 predstavuje nenatreté políčko, hodnota 1 bude predstavovať natreté políčko.

Pre každý úsek $[a, b]$ zo vstupu prejdeme všetky políčka v poli `d[]` od a-teho po b-te políčko a na každé z nich nastavíme hodnotu 1. To bude znamenať, že je natreté. Na konci nám teda stačí iba vypísať obsah poľa `d[]`.

Každý úsek na vstupe môže mať dĺžku najviac n. Úsekov je q, takže časová zložitosť tohto riešenia je $O(nq)$. Pamätáme si iba dážďovku, takže pamäťová zložitosť je $O(n)$.

Listing programu (Python)

```python
n, q = [int(x) for x in input().split()]

dazdovka = [0] * n

# Prejdeme cez vsetky useky
for _ in range(q):
    a, b = [int(x) for x in input().split()]
    # Aktualny usek postupne cely natrieme
    for i in range(a - 1, b):
        dazdovka[i] = 1

print(*dazdovka)
```

Listing programu (C++)

```cpp
#include <bits/stdc++.h>

using namespace std;

int main() {
    int n, q, a, b;
    cin >> n >> q;
    vector<int> dazdovka(n, 0);
    // Prejdeme cez vsetky useky
    for (int Q = 0; Q < q; Q++) {
        cin >> a >> b;
        // Aktualny usek dazdovky cely natrieme
        for (int i = a - 1; i <= b - 1; i++)
            dazdovka[i] = 1;
    }
    // Vypiseme dazdovku
    cout << dazdovka[0];
    for (int i = 1; i < n; i++)
        cout << ' ' << dazdovka[i];
    cout << '\n';
```

```
    return 0;
}
```

Lepšie riešenie

Prečo je riešenie hrubou silou také pomalé? Lebo sa veľakrát pozerá na tie isté políčka dážďovky. Možno by sa dalo spraviť to, že dva prekrývajúce sa úseky spojíme do jedného, dlhšieho. Ak takto úseky pospájame, ostanú nám na konci iba neprekrývajúce sa úseky. To znamená, že súčet dĺžok všetkých týchto úsekov bude nanajvýš n.

Ako to ale spraviť efektívne? Povedzme si, že chceme ísť po dážďovke zľava doprava a nikdy sa nechceme vracať na políčka, ktoré sme už prešli. Načítajme si teda najskôr všetky úseky a usporiadajme si ich podľa začiatku. To znamená, že úsek, ktorý začína viac vľavo spracujeme skôr ako ten, ktorý začína viac vpravo. Rovnako ako v riešení hrubou silou, majme pole `d[]` veľkosti n so samými nulami.

Zoberme prvý úsek, nazvime ho `A`, a prejdime všetky políčka dážďovky, ktoré reprezentuje. Každému z týchto políčok v poli `d[]` nastavíme hodnotu 1. Nastavenie hodnoty 1 v poli `d[]` budeme odteraz nazývať natretím políčka dážďovky.

Pozrime sa na druhý úsek v usporiadanom poradí a nazvime ho `B`. Vieme, že `B` určite nezačína skôr, ako `A`, lebo sme ich usporiadali podľa začiatkov. Nastať teda mohol iba jeden z nasledovných prípadov:

1. `B` je podúsekom `A`, teda `B` začína aj končí skôr ako `A`
2. `B` začína v `A` a končí neskôr, ako `A`
3. `B` začína neskôr, ako `A` končí

V prvom prípade môžeme úsek `B` odignorovať, lebo všetky políčka, ktoré tento úsek zasahuje boli už natreté úsekom `A`.

V druhom prípade niektoré políčka zo začiatku úseku `B` už boli natreté úsekom `A`. Pri natieraní teda stačí pokračovať od konca úseku `A` až po koniec `B`.

No a v treťom prípade môžeme natrieť celý úsek `B`, lebo vieme, že žiadne z jeho políčok doteraz neboli natreté.

Môžeme si všimnúť, že nikdy nepotrebujeme vedieť, kedy začínal naposledy spracovaný úsek. Jediné, čo nás zaujíma je to, ktoré políčko dážďovky sme natreli ako posledné. Stačí nám teda mať jednu premennú, ktorá bude hovoriť, ktoré natreté políčko dážďovky sa nachádza najviac vpravo. Ak teda spracujeme úsek, ktorý končil ďalej, ako bola hodnota tejto premennej v čase jeho spracovania, tak túto premennú nastavíme na koniec tohto úseku.

Máme teda algoritmus, ktorý sa na každé z n políčok dážďovky pozrie najviac raz. Z toho vyplýva časová zložitosť $O(n)$. Pozor ale na to, čo robíme s úsekmi. Usporiadavame ich, čo trvá čas $O(q \log q)$. Naše riešenie má teda zložitosť $O(n + q \log q)$. Pamäťová zložitosť je $O(n + q)$, pretože si pamätáme všetky úseky a celú dážďovku.

Všimnime si, že začiatku a konce zadaných úsekov sú z rozsahu od 1 po n. Pri triedení čísel s takýmto obmedzením vieme využiť aj rýchlejšie triedenia, napríklad countsort, ktorého zložitosť je $O(q+n)$. Tým zlepšíme aj celkovú časovú zložitosť na $O(n + q)$. Ako si však ukážeme o chvíľu, túto zložitosť vieme dosiahnuť aj bez toho, aby sme poznali niektorý z týchto špeciálnych triediacich algoritmov.

Listing programu (Python)

```python
class Usek:
    def __init__(self, zac, kon):
        self.zaciatok = zac
        self.koniec = kon


# Ak chceme riesenie v O(n + q), tak tu pouzijeme radix sort.
# Mu pouzijeme funkciu sorted, takze nase riesenie bude v O(n + q log q)
def usporiadaj(u):
    # Useky chceme usporiadat podla zaciatku od najmensich
    return sorted(u, key=lambda x: x.zaciatok)


if __name__ == '__main__':
    n, q = [int(x) for x in input().split()]
    useky = []
    for _ in range(q):
        a, b = [int(x) for x in input().split()]
        useky.append(Usek(a - 1, b - 1))

    useky = usporiadaj(useky)

    dazdovka = [0] * n
    najdalej = -1
    for usek in useky:
        # Prvy pripad
```

```
        if usek.koniec <= najdalej:
            continue
        # Druhy pripad
        elif usek.zaciatok <= najdalej < usek.koniec:
            for i in range(najdalej + 1, usek.koniec + 1):
                dazdovka[i] = 1
            najdalej = usek.koniec
        # Treti pripad
        else:
            for i in range(usek.zaciatok, usek.koniec + 1):
                dazdovka[i] = 1
            najdalej = usek.koniec
    print(*dazdovka)
```

Listing programu (C++)

```
#include <bits/stdc++.h>

using namespace std;

struct Usek {
    int zaciatok, koniec;
};

// Ak chceme riesenie v O(n + q), tak tu pouzijeme radix sort.
// Mu pouzijeme std::sort, takze nase riesenie bude v O(n + q log q)
void usporiadaj(vector<Usek> &useky) {
    // Useky chceme usporiadat podla zaciatku od najmensich
    sort(useky.begin(), useky.end(), [](const Usek &A, const Usek &B) {
        return A.zaciatok < B.zaciatok;
    });
}

int main() {
    int n, q;
    cin >> n >> q;
    vector<Usek> useky(q);
    for (int i = 0; i < q; i++) {
        cin >> useky[i].zaciatok >> useky[i].koniec;
        // Indexujeme od 0
        useky[i].zaciatok--;
        useky[i].koniec--;
    }

    usporiadaj(useky);

    vector<int> dazdovka(n, 0);
    int najdalej = -1; // Doteraz posledne natrete policko
    for (Usek usek: useky) {
        // Prvy pripad
        if (usek.koniec <= najdalej) {
            continue;
        }
        // Druhy pripad
        else if (usek.zaciatok <= najdalej && usek.koniec > najdalej) {
            for(int policko = najdalej + 1; policko <= usek.koniec; policko++)
                dazdovka[policko] = 1;
            najdalej = usek.koniec;
        }
        // Treti pripad
        else {
            for(int policko = usek.zaciatok; policko <= usek.koniec; policko++)
                dazdovka[policko] = 1;
            najdalej = usek.koniec;
        }
    }

    cout << dazdovka[0];
    for(int i = 1; i < n; i++)
        cout << ' ' << dazdovka[i];
    cout << '\n';
}
```

Vzorové riešenie

Optimálnu časovú zložitosť sme už síce získali drobným trikom v predchádzajúcom riešení, no poďme sa pozrieť na krajšie a jednoduchšie riešenie, ktoré má dokonca lepšiu pamäťovú zložitosť.

Opäť budeme mať pole dĺžky n, nazvime ho `p[]`. Tentokrát ale i-te políčko tohto poľa bude hovoriť, koľko úsekov na tomto políčku začína alebo končí. To znamená, že ak spracúvame úsek `[a, b]`, tak k `p[a]` pripočítame 1 a k `p[b + 1]` pripočítame -1. Kladné číslo označuje začiatok a záporné koniec. Naviac si všimnite, že koniec je v skutočnosti až jedno políčko za reálnych koncom a to preto, aby sme sme aj toto posledné políčko započítali do zadaného úseku.

Čo teraz môžeme robiť s týmto poľom? Môžeme pre každé políčko zistiť, koľko úsekov ho prekrýva. Majme počítadlo, ktoré hovorí, koľkými úsekmi je prekryté aktuálne políčko. Na začiatku bude mať počítadlo hodnotu 0. Teraz prejdime cez všetky políčka poľa `p[]` zľava doprava. Keď spracúvame i-te políčko, tak k nášmu počítadlu pripočítame hodnotu `p[i]`. Ak totiž na i-tom políčku začínajú nejaké úseky, tak ich počet chceme pripočítať k počítadlu. Ak nejaké úseky končili na políčku $i - 1$, tak ich teraz chceme od počítadla odpočítať.

Navyše, naše pole `p[]` bez problémov zvláda aj viaceré úseky začínajúce alebo končiace na tom istom políčku. Ak napríklad na pozícii a začínajú tri rôzne úseky, tak hodnotu `p[a]` sme zvýšili o 1 trikrát, preto je tam hodnota 3. A ak na políčku a začína úsek a zároveň na políčku $a - 1$ nejaký skončil, tak hodnota `p[a]` bude rovná 0. To je však v poriadku, pretože 0 znamená, že sa nič nemení.

No a tu už vidíme riešenie. Budeme prechádzať pole `p[]` a aktualizovať počítadlo úsekov. Ak po aktualizácii počítadla políčkom `p[i]` bude počítadlo väčšie, ako 0, tak vieme, že i-te políčko je natreté. Ak je počítadlo 0, tak je i-te políčko nenatreté, lebo ho neprekrýva žiaden úsek. Počítadlo určite nikdy nebude menšie, ako 0, pretože ak je niekde v poli `p[]` hodnota -1, tak niekde skôr musí byť jej zodpovedajúca $+1$.

Pre každý úsek zapíšeme iba dve hodnoty do poľa `p[]` a toto pole následne raz prejdeme. Výsledná časová zložitosť je teda $O(q+n)$. Pamätáme si však už len pole `p[]` (a jednu premennú na počítadlo), preto je pamäťová zložitosť $O(n)$.

Listing programu (Python)

```python
n, q = [int(x) for x in input().split()]

# -1 davame az za koniec useku, takze potrebujeme
# pole velkosti az n + 1 kvoli usekom konciacim
# na poslednom policku dazdovky
p = [0 for _ in range(n + 1)]

for i in range(q):
    a, b = [int(x) for x in input().split()]
    a -= 1 # Indexujeme od 0
    b -= 1 # Indexujeme od 0
    p[a] += 1
    p[b + 1] -= 1

dazdovka = [0] * n

pocitadlo = 0  # Kolko usekov prekryva aktualne policko
for i in range(n):
    pocitadlo += p[i]
    # Aktualne policko je prekryte aspon jednym usekom
    if pocitadlo > 0:
        dazdovka[i] = 1

print(*dazdovka)
```

Listing programu (C++)

```cpp
#include <bits/stdc++.h>

using namespace std;

int main() {
    int n, q;
    cin >> n >> q;
    // -1 davame az za koniec useku, takze potrebujeme
    // pole velkosti az n + 1 kvoli usekom konciacim
    // na poslednom policku dazdovky
    vector<int> p(n + 1, 0);
    for (int i = 0; i < q; i++) {
        int a, b;
        cin >> a >> b;
        a--; b--; // indexujeme od 0
        p[a] += 1;
        p[b + 1] -= 1;
    }
    // Kolko usekov prekryva aktualne policko
    int pocitadlo = 0;
    vector<int> dazdovka(n, 0);
    for (int i = 0; i < n; i++) {
        pocitadlo += p[i];
        // Aktualne policko je prekryte aspon jednym usekom
        if (pocitadlo > 0)
            dazdovka[i] = 1;
    }
    // Vypiseme dazdovku
    cout << dazdovka[0];
    for(int i = 1; i < n; i++)
        cout << ' ' << dazdovka[i];
    cout << '\n';

    return 0;
}
```

Iné riešenia

Riešení tejto úlohy existuje viac a nespomenieme tu všetky. Za zmienku ale stojí riešenie s neoptimálnou časovou zložitosťou $O(q \log n)$. Toto riešenie vie natrieť celý úsek v čase $O(\log n)$. To je veľmi jednoduché, ak na reprezentáciu dážďovky použijeme vhodnú dátovú štruktúru. Najjednoduchšou z možností je asi použiť intervalový strom s lazy propagáciou. Táto dátová štruktúra má široké uplatnenie a ak ju nepoznáte, určite si o

nej niečo prečítajte. Keď si o nej už niečo prečítate, vymyslieť toto riešenie bude triviálne a nechávame ho ako cvičenie pre vás.

5. Kandidát

vzorák napísal(a) Jitka
(max. 15 b za riešenie)

Našou úlohou bolo zistiť, či sa Emo stane prezidentom. Na vstupe sme dostali zoznam známostí v Trojstene a overovali sme, s kým sa Emo pozná priamo alebo ob jedného človeka. Vždy sme teda určili, či sa Emo pozná s každým a ak nie, tak aj s kým sa nepozná.

Pomalšie riešenie

Otázou je, ako si zapamätať známosti, ktoré v Trojstene sú. Prvou možnosťou je použiť maticu susedností, ktorú budeme volať `znamosti[][]`. Je to dvojrozmerné pole, ktoré má toľko riadkov a stĺpcov, koľko je občanov Trojstenu. Každému občanovi patrí jeden riadok, v ktorom si v stĺpcoch pre každého ďalšieho občana pamätáme, či sa s ním pozná alebo nie. To znamená, že na pozícii `znamosti[a][b]` je hodnota 1, ak sa občan a pozná s občanom b, inak je tam hodnota 0. Keď máme vytvorenú takúto prázdnu maticu, pridať nové známosti je jednoduché. Pridanie známosti medzi občanmi a a b znamená nastaviť hodnotu `znamosti[a][b]` a `znamosti[b][a]` na 1. No a nezabudneme si poznačiť, že Emo pozná aj Ema.

Ako pomocou tejto matice zistíme koho pozná Emo priamo a ob jedného človeka? Vytvorme si pole `poznam[]`, kde si budeme o každom občanovi pamätať, či ho Emo pozná alebo nie.

Priame známosti sú jednoduché. Pozrieme sa na riadok reprezentujúci Ema v dvojrozmernom poli `znamosti[][]` a postupne ho celý prejdeme. Takto sa spýtame na každého občana, či má priamu známosť s Emom a ak má zapíšeme si to do poľa `poznam[]`.

Emo pozná občana c ob jedného občana ak niekto, koho Emo pozná, pozná c. Teda pre každého občana x, o ktorom sme zistili, že ho Emo pozná sa teda pozrieme na x-tý riadok matice `znamosti[x][]`, kde sú všetci ľudia, ktorých x pozná a aj tých si zaznačíme do poľa `poznam[]`.

Nakoniec len prejdeme pole `poznam[]` a zistíme, či Emo pozná každého. Môžeme si vytvoriť pomocné pole `nepoznam[]`, kam si prepíšeme občanov, ktorých Emo nepozná. Ak pozná všetkých ľudí, vypíšeme `Ano`, ak nie vypíšeme `Nie` a prechodom pola `nepoznam[]` vypíšeme občanov, ktorých nepozná.

Pole `poznam[]` má len n políčok a prechádzame ho konštatne veľa krát. Vybudovanie dvojrozmerného poľa `znamosti[][]` nám však bude trvať $n \cdot n$ krokov, lebo pre každé políčko si musíme zaznačiť, či tá známosť existuje alebo nie a `znamosti[][]` majú n^2 políčok. Časová zložitosť tohto riešenie je teda $O(n^2)$. Najviac si pamätáme práve dvojrozmerné pole `znamosti[][]`, teda aj pamäťová zložistosť bude $O(n^2)$.

Listing programu (Python)

```python
n, m, e = map(int, input().split())

# vytvorime si dvojrozmerne pole znamosti a pole poznam
znamosti = [[0]*n for i in range(n)]
poznam = [0]*n
# znizime Emove cislo, aby sme mohli indexovat od 0
e -= 1
poznam[e] = 1

# zapiseme si znamosti do matice susednosti znamosti
for _ in range(m):
    a, b = map(int, input().split())
    # znizime cisla obcanov, aby sme mohli indexovat od 0
    a -= 1
    b -= 1
    znamosti[a][b] = 1
    znamosti[b][a] = 1

# prejdeme obcanov, ktorych Emo moze priamo poznat
for i in range(len(znamosti[e])):
    if znamosti[e][i] == 1:
        poznam[i] = 1
        # ked Emo pozna obcana, prejdeme aj jeho znamosti
        for j in range(len(znamosti[i])):
            if znamosti[i][j] == 1:
                poznam[j] = 1
nepoznam = []

# skontrolujeme ci Emo pozna kazdeho
if sum(poznam) == n:
    print("Ano")
else:
    print("Nie")
    # prepiseme obcanov, ktorych Emo nepozna do pomocneho pola
    for i in range(len(poznam)):
        if poznam[i] == 0:
            nepoznam.append(i+1)
    # vypiseme obcanov, ktorych Emo nepozna
    print(*nepoznam)
```

Listing programu (C++)

```
#include <bits/stdc++.h>

using namespace std;

int main()
{
    int n, m, e;
    cin >> n >> m >> e;

    // vytvorime si dvojrozmerne pole znamosti a pole poznam
    vector<vector<int>> znamosti(n, vector<int>(n, 0));
    vector<int> poznam(n, 0);
    // znizime cislo oznacujuce Ema, aby sme mohli indexovat od 0
    e--;
    poznam[e] = 1;

    // vyplnime maticu susednosti znamosti
    for (int i = 0; i < m; i++)
    {
        int a, b;
        cin >> a >> b;
        // znizime cisla, aby sme vedeli indexovat od 0
        a--;
        b--;
        znamosti[a][b] = 1;
        znamosti[b][a] = 1;
    }

    // prejdeme obcanov, ktorych Emo moze poznat priamo
    for (int i = 0; i < znamosti[e].size(); i++)
    {
        if (znamosti[e][i] == 1)
        {
            poznam[i] = 1;
            // ak Emo pozna obcana prejdeme aj jeho znamosti
            for (int j = 0; j < znamosti[i].size(); j++)
            {
                if (znamosti[i][j] == 1)
                {
                    poznam[j] = 1;
                }
            }
        }
    }

    // prepiseme si obcanov, ktorych Emo nepozna do pomocneho pola
    vector<int> nepoznam;
    for (int i = 0; i < poznam.size(); i++)
    {
        if (poznam[i] == 0)
        {
            nepoznam.push_back(i + 1);
        }
    }

    // skontrolujeme, ci pozna Emo kazdeho
    if (nepoznam.size() == 0)
    {
        cout << "Ano" << endl;
    }
    else
    {
        cout << "Nie" << endl;
        // vypiseme obcanov, ktorych Emo nepozna
        for (int i = 0; i < nepoznam.size(); i++)
        {
            cout << nepoznam[i];
            if (i < nepoznam.size() - 1)
            {
                cout << " ";
            }
            else
            {
                cout << endl;
            }
        }
    }
}
```

Vzorové riešenie

Predošlé riešenie je síce korektné, ale nie najrýchlejšie. Zamyslime sa, čo by sa dalo robiť lepšie oproti predchádzajúcemu riešeniu. V matici `znamosti` sme si pamätali všetko. Nás však zaujíma vzťah dvoch občanov iba ak sa poznajú, ak sa nepoznajú netreba nám to vedieť.

Prerobme si trochu dvojrozmerné pole `znamosti` na dvojrozmerný vektor. Vektor je pole, ktoré vie meniť veľkosť. Pre každého občana si budeme pamätať vektor občanov, ktorých priamo pozná. Tento spôsob zapamatania si vzťahov medzi občanmi sa nazýva zoznam susedov.

Zvyšok riešenia už je podobný ako v predošlom riešení. Vytvoríme si pole `poznam`, v ktorom si budeme pamätať koho Emo pozná. Najprv sem prepíšeme občanov, ktorých Emo pozná priamo zo `znamosti`. Samoz-

rejme nezabudneme, že Emo pozná Ema. Následne pre každého nájdeného občana, ktorého Emo pozná priamo, zapíšeme do `poznam` občanov, ktorých on pozná priamo zo `znamosti`.

Nakoniec, rovnako ako v pomalšom riešení, len prejdeme pole `poznam` a zistíme, či Emo pozná každého. Môžeme si vytvoriť pomocný vektor `nepoznam`, kde si prepíšeme občanov, ktorých Emo nepozná. Ak je tento vektor prázdny, tak Emo pozná každého a vypíšeme `Ano`, ak nie vypíšeme `Nie` a vektor `nepoznam`.

Zapísaním známostí do zoznamu susedov sme výrazne zlepšili časovú aj pamäťovú zložitosť. Pole `poznam` sa nezmenilo a ani počet jeho prechodov, to nás teda neovplyvňuje. `Znamosti` sú však pole všetkých n občanov a pre každého si pamätáme len občanov, ktorých priamo pozná a nie tých, ktorých nepozná. To znamená, že každý vzťah zo zadania si pamätáme iba 2 krát. 2 krát kvôli tomu, že pri vzťahu a, b, nielen občan a pozná občana b, ale aj občan b pozná občana a. Pamäťová zložitosť je teda $O(n + 2m)$. Časová zložitosť je rovnaká, lebo `znamosti` prejdeme konštantne veľa krát. V zložitostiach môžeme zanedbať konštanty, preto sa zložistosti zjednodušia na $O(n + m)$.

Listing programu (Python)

```python
n, m, e = map(int, input().split())

# vytvorime si dvojrozmerny vektor znamosti a pole poznam
znamosti = [[] for i in range(n)]
poznam = [0]*n
# znizime Emove cislo, aby sme mohli indexovat od 0
e -= 1
poznam[e] = 1

# nacitame do zoznamu susedov znamosti
for _ in range(m):
    a, b = map(int, input().split())
    # znizime cisla obcanov, aby sme mohli indeovat od 0
    a -= 1
    b -= 1
    znamosti[a].append(b)
    znamosti[b].append(a)

# pridame obcanov, ktorych Emo priamo pozna do pola poznam
for priamo_poznam in znamosti[e]:
    poznam[priamo_poznam] = 1
    # ak obcana Emo pozna, pridame aj jeho znamych do pola poznam
    for obcan in znamosti[priamo_poznam]:
        poznam[obcan] = 1
nepoznam = []
# skontrolujeme, ci Emo pozna vsetkych
if sum(poznam) == n:
    print("Ano")
else:
    print("Nie")
    # vypiseme obcanov, ktorych Emo nepozna
    for i in range(len(poznam)):
        if poznam[i] == 0:
            nepoznam.append(i+1)
    print(*nepoznam)
```

Listing programu (C++)

```cpp
#include <bits/stdc++.h>

using namespace std;

int main()
{
    int n, m, e;
    cin >> n >> m >> e;
    // vytvorime si dvojrozmerny vektor znamosti a pole poznam
    vector<vector<int>> znamosti(n);
    vector<int> poznam(n, 0);
    // znizime cislo Ema, aby sme mohli indexovat od 0
    e--;
    poznam[e] = 1;

    // nacitame do zoznamu susedov znamosti
    for (int i = 0; i < m; i++)
    {
        int a, b;
        cin >> a >> b;
        // znizime cisla obcanov, aby sme mohli indexovat od 0
        a--;
        b--;
        znamosti[a].push_back(b);
        znamosti[b].push_back(a);
    }

    // pridame obcanov, ktorych Emo priamo pozna do pola poznam
    for (auto priamo_poznam : znamosti[e])
    {
        poznam[priamo_poznam] = 1;
        // ak obcana Emo pozna pridame aj jeho znamych
        for (auto obcan : znamosti[priamo_poznam])
        {
```

```
            poznam[obcan] = 1;
        }
    }
    vector<int> nepoznam;
    // prepiseme si obcanov, ktorych Emo nepozna do pomocneho pola
    for (int i = 0; i < poznam.size(); i++)
    {
        if (poznam[i] == 0)
        {
            nepoznam.push_back(i + 1);
        }
    }
    // skontrolujme, ci Emo pozna kazdeho
    if (nepoznam.size() == 0)
    {
        cout << "Ano" << endl;
    }
    else
    {
        cout << "Nie" << endl;
        // vypiseme obcanov, ktorych Emo nepozna
        for (int i = 0; i < nepoznam.size(); i++)
        {
            cout << nepoznam[i];
            if (i < nepoznam.size() - 1)
            {
                cout << " ";
            }
            else
            {
                cout << endl;
            }
        }
    }
}
```