Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Bakalářská práce

Sada úloh pro výuku programování

Plzeň 2017 Ondřej Skála

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta aplikovaných věd
Akademický rok: **2016/2017**

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Ondřej SKÁLA**
Osobní číslo:	**A14B0351P**
Studijní program:	**B3902 Inženýrská informatika**
Studijní obor:	**Informatika**
Název tématu:	**Sada úloh pro výuku programování**
Zadávající katedra:	**Katedra informatiky a výpočetní techniky**

Zásady pro vypracování:

1. Prostudujte a stručně popište stránky nejznámějších online systémů, které podporují přípravu na ACM soutěže v programování.
2. Na validátoru Uva Online Judge (http://uva.onlinejudge.org) či jiném vyberte sadu zajímavých úloh vhodných pro výuku programování na počátku bakalářského studia.
3. Vypracujte české verze zadání úloh. Proveďte rozbor zadání úloh a analýzu možných algoritmů řešení.
4. Úlohy naprogramujte v Javě a funkčnost programů ověřte na příslušném validátoru.

Rozsah grafických prací: **dle potřeby**

Rozsah kvalifikační práce: **doporuč. 30 s. původního textu**

Forma zpracování bakalářské práce: **tištěná**

Seznam odborné literatury:

dodá vedoucí bakalářské práce

Vedoucí bakalářské práce: **Ing. Arnoštka Netrvalová, Ph.D.**
Katedra informatiky a výpočetní techniky

Datum zadání bakalářské práce: **10. října 2016**

Termín odevzdání bakalářské práce: **4. května 2017**

Doc. RNDr. Miroslav Lávička, Ph.D.
děkan

L.S.

Doc. Ing. Přemysl Brada, MSc. PhD.
vedoucí katedry

V Plzni dne 13. října 2016

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 25. dubna 2017

Ondřej Skála

Poděkování

Mé poděkování patří Ing. Arnoštce Netrvalové, Ph.D., za odborné vedení, cenné rady, trpělivost a ochotu, kterou mi v průběhu zpracování bakalářské práce věnovala.

Abstract

This thesis deals with well-known online systems for programming training, which are described from the point of view of using of these systems and their focus. In the first part of thesis there are also described task descriptions.

In the second part of the thesis there is described set of chosen tasks for teaching of programming on Faculty of Applied Sciences of University of West Bohemia. For each chosen task there are described task analysis, algorithms suitable for solving of the task and software solution. The correctness of each of software solutions was checked on the validator of appropriate system for programming training.

Abstrakt

Práce se ve své první části zabývá známými systémy pro nácvik programování, které popisuje z hlediska práce s danými systémy a jejich zaměření. Rovněž jsou v první části práce popsána zadání úloh na serverech těchto systémů.

Ve druhé části práce jsou uvedeny vybrané úlohy vhodné pro výuku programování na Fakultě aplikovaných věd Západočeské univerzity v Plzni. U jednotlivých úloh je provedena analýza, popsány algoritmy vhodné pro řešení úloh a dále je uvedeno programové řešení úloh. Správná funkčnost programových řešení úloh byla zkontrolována validátorem příslušného systému pro nácvik programování.

Obsah

1 Úvod

V poslední době se na internetu formují velké komunity programátorů kolem online systémů, jejichž cílem je usnadnit začátečníkům počátky programování a pokročilým programátorům umožnit rozšíření jejich programátorských dovedností. Dalším cílem systémů pro nácvik programování je příprava programátorů na ACM soutěže.

První část práce popisuje známé systémy pro nácvik programování a přípravu na ACM soutěže. Popis systémů se zaměřuje především na způsob práce s danými systémy, počet podporovaných programovacích jazyků, zaměření těchto systémů a v neposlední řadě i na soutěže, které jsou komunitou při těchto systémech pořádány.

Hlavním cílem bakalářské práce je výběr zajímavých úloh vhodných pro výuku programování na počátku bakalářského studia. K vybraným úlohám jsou připraveny české verze zadání úloh, rozbor zadání úloh s analýzou možných algoritmů řešení a rovněž vzorový zdrojový kód programového řešení úloh v programovacím jazyce Java.

2 Online systémy pro nácvik programování

Hlavním účelem online systémů pro nácvik programování je zdokonalování programátorských dovedností začínajících i pokročilých programátorů. Online systémy proto nabízejí svým uživatelům katalogy úloh, z nichž si může uživatel konkrétní úloh vybrat a vyřešit.

Uživatelem odeslané řešení je na serveru online systému nejprve přeloženo, čímž se zkontroluje správná syntaxe odeslaného zdrojového kódu. Pokud překlad programu proběhne správně, je následně program spuštěn. Po překladu programu systém zkontroluje správnou funkci programu, kdy systém programu zadá předem připravenou množinu vstupních dat a následně výstup programu zvaliduje proti předem připraveným vzorovým výstupním datům. Pokud se výstupní data programu shodují s předem připravenými výstupními daty, online systém prohlásí odeslané řešení jako správné, v opačném případě online systém odeslané řešení odmítne jako nesprávné.

Online systémy pro nácvik programování typicky umožňují zobrazit pořadí uživatelů s nejvyšším počtem odeslaných správných řešení, která byla vyhodnocena jako správná, a dále pořadí uživatelů podle času, který vyžadoval odeslaný program pro svůj běh se vzorovými vstupními daty [17].

Na serverech online systémů jsou rovněž často pořádány programátorské soutěže, mezi něž patří například ACM International Collegiate Programming Contest. Systémy umožňují svým uživatelům na tyto soutěže trénovat.

Mezi známé systémy pro nácvik programování můžeme zařadit například systémy:

- CodeChef,
- CodinGame,
- HackerRank,
- Sphere Online Judge,
- TopCoder,
- UVa Online Judge,
- Timus Online Judge.

2.1 CodeChef

Systém CodeChef (viz obrázek 2.1) je projektem indické společnosti Directi [1], která na trhu působí jako registrátor doménových jmen. Systém CodeChef umožňuje nácvik programování ve více než 35 programovacích jazycích, mezi něž patří nejen obvyklé jazyky jako např. Java, ANSI C, C++, Python, LISP, ale rovněž jsou mezi podporovanými jazyky zastoupeny některé méně obvyklé programovací jazyky jako např. F#, Nice, LUA.

Obrázek 2.1: Logo systému CodeChef
Zdroj: `http://www.codechef.com` [cit. 2016/10/17]

Server každý měsíc pořádá několik soutěží, v nichž mohou uživatelé vyhrát hodnotné ceny. Mezi soutěže, které jsou pořádány každý měsíc, patří například soutěž „Cook-Off“, která obsahuje několik úloh, na jejichž vyřešení mají soutěžící 2,5 hodiny.

Pro uživatele jsou na serveru připraveny návody k některým úlohám na tomto serveru. Dále se zde nachází rozpracované kurzy pro témata týkající se algoritmizace. V době psaní této části práce (říjen 2016) bylo dostupných 8 kurzů, dalších 5 kurzů čekalo na dokončení.

2.1.1 Zadání úloh

Pro uživatele jsou v katalogu příkladů úlohy rozděleny do pěti skupin - úlohy pro začátečníky, lehké úlohy, středně těžké úlohy, těžké úlohy a výzvy. Díky rozdělení problémů podle obtížnosti není programátor začátečník frustrován, že není schopen vyřešit pro něj složitou úlohou, a zároveň pokročilý programátor není „nuděn“ příliš jednoduchými zadáními úloh. Uživatelům jsou dále přístupné odeslané zvalidované zdrojové kódy ostatních uživatelů, tedy uživatelé se mohou podívat na správná řešení daného problému odeslaná jinými uživateli.

Zadání úloh jsou dostupná v angličtině, mandarínské čínštině, ruštině a vietnamštině. U každé úlohy je popsán problém, struktura vstupních dat, očekávaná struktura výstupních dat, omezení vstupních dat, ukázkový vstup a výstup. Dále jsou u každé úlohy uvedena omezení systémových prostředků,

které může uživatelem odeslaný program využívat. Jde o časový limit, během něhož musí program vyřešit zadaný problém pro daná vstupní data, a maximální možnou velikost zdrojového kódu odeslaného uživatelem k validaci. Zadání úlohy rovněž obsahuje seznam programovacích jazyků, v nichž může být řešení dané úlohy zrealizováno.

U některých úloh ze sekce začátečnických, lehkých a středně těžkých úloh zadání rovněž obsahuje stručné vysvětlení dané úlohy. Úloha také může být složena z více podúloh. V případě úlohy s podúlohami řešitel obdrží alespoň část bodů, pokud je program schopen v daném časovém limitu vyřešit omezený rozsah vstupních dat, ale není dostatečně rychlý pro vyřešení větších vstupních dat v zadaném časovém limitu.

2.1.2 Možné výsledky validace úloh

Systém CodeChef podle výsledku proběhlé validace uživateli oznámí jeden z následujících výsledků validace: [2]:

- **Compile Error** - v případě syntaktické chyby v odeslaném řešení nebo pokud uživatel nedodržel závazné pokyny (například v případě programovacího jazyka Java není třída, ve které je uložena metoda `main`, pojmenovaná `Main`);
- **Time Limit Exceeded** - v případě, kdy program nevyřeší řešenou úlohu pro daná vstupní data v časovém limitu, který je uveden v zadání každé z úloh;
- **Runtime Exception** nebo **Runtime Error** - v případě, kdy během vykonávání programu dojde k chybě, například vyhození výjimky;
- **SIGSEGV** - v případě, kdy program přistupuje k neplatné referenční proměnné nebo dojde k porušení ochrany paměti (anglicky „segmentation fault“);
- **SIGABRT** - v případě přerušení programu z důvodu fatální chyby (například v případě jazyka C++ není splněn predikát obsažený v příkazu `assert`);
- **SIGFPE** - v případě chyby výpočtu s plovoucí desetinnou čárkou (například pokud je v programu obsažena instrukce dělení nulou);
- **NZEC** - v případě, kdy návratový kód spuštěného programu není roven nule (například pokud v případě programovacího jazyka C metoda `main` vrací jinou hodnotu než nula, v případě programovacího jazyka

Java může jít o případ, kdy je během běhu programu vyhozena výjimka);

- **Wrong Answer** - v případě, kdy se výstup hodnoceného programu neshoduje se vzorovým výstupem pro danou úlohu a daná vzorová vstupní data;
- **Accepted** - v případě, kdy výstup hodnoceného programu je správný, tedy pokud se shoduje se vzorovým výstupem pro danou úlohu a daná vzorová vstupní data.

2.2 CodinGame

Francouzský systém CodinGame (viz obrázek 2.2) je provozován stejnojmenným francouzským startupem. Systém umožňuje svým uživatelům rozšiřování programátorských dovedností v 25 programovacích jazycích, mezi nimiž nechybí běžní zástupci jako například ANSI C, Java a PHP. Rovněž jsou zastoupeny i méně obvyklé programovací jazyky, například Clojure, Dart a Swift [3].

Obrázek 2.2: Logo systému CodinGame
Zdroj: `http://www.codingame.com` [cit. 2016/10/19]

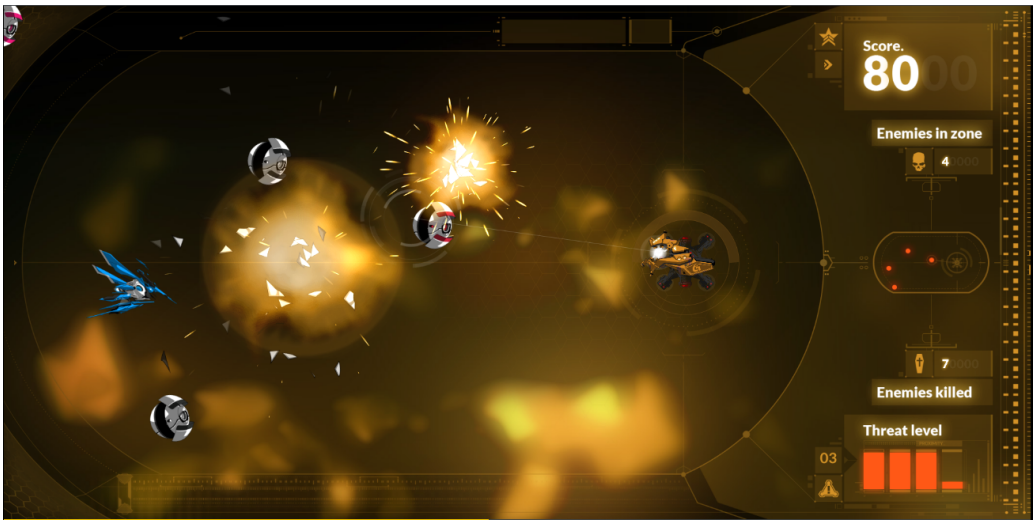
Jedním z cílů systému CodinGame je umožnit společnostem hledat nové zaměstnance mezi uživateli systému. Společnosti hledající zaměstnance mají možnost na tomto serveru pořádat a sponzorovat soutěže a mezi soutěžícími dané soutěže pak mohou hledat nové zaměstnance pro rozšíření řad stávajících zaměstnanců. Soutěže mohou probíhat online, v tomto případě mají soutěže delší časové okno, během něhož mohou uživatelé soutěžit. Druhou možností jsou soutěže, které probíhají na jednom místě v jeden čas. Všechny soutěže jsou plně anonymní a společnosti hledající nové zaměstnance mohou obdržet osobní údaje soutěžícího pouze s výslovným souhlasem uživatele, jehož osobní údaje požadují.

2.2.1 Zadání úloh

Úlohy, na tomto serveru nazývané puzzle, jsou v případě úloh, které jsou určeny pro trénování programování, rozděleny na úlohy jednoduché, středně

těžké, těžké, velmi těžké, úlohy připravené jinými uživateli a úlohy týkající se strojového učení. U každé z trénovacích úloh je uvedeno, jaké oblasti se daná úloha týká a obtížnost úlohy, dále co by si měl uživatel vyřešením dané úlohy procvičit a odkazy na externí zdroje, které se týkají úlohy. Uživatel si může zobrazit algoritmus pro vyřešení dané úlohy, pseudokód řešení i řešení zapsané v uživatelem vybraném programovacím jazyce.

Každá úloha je koncipována jako počítačová hra. Příkladem může být úvodní hra s názvem Onboarding, v níž má uživatel za úkol bránit planetu před invazí mimozemských lodí. Vstupem programu je název dvou mimozemských lodí a jejich vzdálenost od planety. Pro vyřešení úlohy musí uživatel ve smyčce sestřelovat bližší loď, přičemž sestřelení uživatel provede vypsáním jména lodi na konzoli. Prostředí systému umožňuje vizualizaci běhu programu, kdy uživatel vidí průběh hry, v případě úlohy Onboarding dochází k sestřelování lodí, jak je zřejmé z obrázku 2.3.



Obrázek 2.3: Vizualizace průběhu programového řešení úlohy Onboarding

2.2.2 Možné výsledky validace úloh

V případě syntaktické chyby v programu systém uživateli ve webovém rozhraní přímo zobrazí problematickou část kódu, v níž se nachází syntaktická chyba. V případě, kdy překlad programu proběhne bezproblémově, začíná vlastní hra. Uživatel vyhraje, pokud jeho program správně vyřeší úlohu pro daná vstupní data, nebo prohraje, pokud řešení odeslané uživatelem není schopno správně vyřešit danou úlohu.

Hodnocení programu sestává z jednoho nebo více testovacích případů. Pokud uživatelův program vyřeší alespoň jeden z testovacích případů, je

validované řešení procentuálně ohodnoceno, přičemž hodnocení odpovídá poměru vyřešených testovacích případů vůči celkovému počtu testovacích případů.

2.3 HackerRank

Primárním cílem systému HackerRank (viz obrázek 2.4), provozovaného stejnojmennou společností, je usnadnit společnostem, které hledají nové zaměstnance, hledání nových talentů pro rozšíření řad stávajících zaměstnanců. Systém proto hodnotí své uživatele na základě jejich programátorských schopností, což umožňuje společnostem redukovat čas nutný pro výběr kandidátů vhodných pro nabízené pracovní pozice [4]. K hodnocení uživatelů je na serveru používán hodnotící systém založený na ELO, pomocí něhož je pro každého uživatele, který se účastní soutěže, spočítáno pravděpodobné umístění v soutěži. Pokud je uživatelem dosažené umístění v rámci dané soutěže lepší, než bylo očekáváno, tak se hodnocení uživatele zvýší, v opačném případě dojde ke snížení hodnocení uživatele [5].

Obrázek 2.4: Logo systému HackerRank
Zdroj: `http://upload.wikimedia.org` [cit. 2016/10/20]

Na serveru jsou pořádány soutěže, přičemž hodnocení uživatele v některých z pořádaných soutěží slouží pro výpočet ohodnocení uživatele. Soutěže jsou na serveru roztříděny do kategorií, kterých se daná soutěž dotýká. Z kategorií lze jmenovat například umělou inteligenci, algoritmizaci, funkcionální programování, některé soutěže pak jsou zařazeny do kategorií podle podporovaného programovacího jazyka, například v minulosti šlo o soutěže zaměřené na programovací jazyky Java, C++ a SQL.

2.3.1 Zadání úloh

V části systému věnované trénování programování jsou úlohy koncipované jako kurzy, které se týkají nějakého problému. Jsou zde zařazeny jak kurzy konkrétních programovacích jazyků (například Java, C++, Python, Ruby,

SQL), tak i kurzy, které se zabývají konkrétní oblastí (například umělá inteligence, datové struktury, distribuované systémy, funkcionální programování, databáze a další).

Podle typu kurzů se pak liší výčet podporovaných jazyků, kdy u kurzů týkajících se konkrétního programovacího jazyka může být povolený jen jeden jazyk, nicméně u kurzů zabývajících se nějakou problémovou oblastí, jako je například kurz týkající se datových struktur, je povolených jazyků více. Nicméně i v tomto případě může být seznam povolených programovacích jazyků omezený vzhledem k vlastnostem konkrétního jazyka (například v případě kurzu týkajícího se spojového seznamu jsou povoleny jen některé objektově orientované programovací jazyky - Java, C++, Python). U kurzů, u nichž není výběr jazyka nijak omezen, si může uživatel vybrat z dostatečného množství jazyků, mezi nimiž nechybí běžní zástupci (například C, C++, Java), méně obvyklé programovací jazyky (například Nim, Julia, Clojure) a nalezneme zde i některé ezoterické programovací jazyky (například LOLCODE, Whitespace).

U každé úlohy je uveden stručný rozbor úlohy, náročnost úlohy a maximální bodové ohodnocení úlohy. Uživatel získá maximální počet bodů, pokud odeslaný program správně vyřeší všechny testovací případy, méně bodů pak uživatel obdrží v případě, kdy jeho program není schopen vyřešit všechny testovací případy, avšak vyřeší alespoň jeden testovaný případ.

2.3.2 Možné výsledky validace úloh

Po odeslání řešení uživatelem systém program přeloží a spustí. V případě syntaktické chyby v programu systém uživateli ve webovém rozhraní přímo zobrazí chybu. Pokud překlad proběhne správně, systém spustí program a zkontroluje výstupní data. V případě chybného výstupu systém uživateli zobrazí vstupní data a výstup programu pro daný testovací případ. Pokud je výstup programu správný, systém uživateli oznámí úspěšnou validaci programu.

2.4 Sphere Online Judge

Systém Sphere Online Judge (viz obrázek 2.5) je provozován polskou společností Sphere Research Labs. Systém v době psaní této části práce (říjen 2016) podporuje více než 45 programovacích jazyků [6], mezi nimiž jsou běžní zástupci jako například C, C++, Java; zástupci méně používaných programovacích jazyků, mezi nimiž nalezneme například Nice, Nim, LUA; a

dále některé ezoterické programovací jazyky, z nichž je zastoupen například Whitespace [6].

Na serveru je pořádáno mnoho soutěží, systém Sphere Online Judge od doby svého vzniku hostoval více než 4000 soutěží [7].

Obrázek 2.5: Logo systému Sphere Online Judge
Zdroj: `http://spoj.com` [cit. 2016/10/20]

2.4.1 Zadání úloh

Server nabízí více než 13000 úloh pro trénování programování [6], jejichž zadání jsou dostupná ve více jazycích, konkrétně v angličtině, polštině, vietnamštině, portugalštině a dalších jazycích.

Úlohy jsou rozděleny do čtyř kategorií podle možných výsledků validace [8]. Jde o klasické úlohy, kde jsou možné dva výsledky validace - úspěšná validace nebo špatná odpověď, dále jsou zde úlohy „výzvy“, kde je uživatel hodnocen na základě kvality jeho řešení (například na základě velikosti zdrojového kódu odeslaného řešení nebo na základě rozdílu přesného řešení a programem vypočteného přibližného řešení). Dalším typem úloh je typ „tutorial“, které odpovídají typu klasických úloh, ale jde o jednodušší úlohy nebo úlohy s obecně známým algoritmem řešení, což tento typ úloh předurčuje k výukovým účelům. Posledním typem jsou úlohy typu „partial“, které odpovídají úlohám typu „výzva“, ale jsou podobně jako úlohy typu „tutorial“ určeny pro výukové účely.

Popis zadání úlohy sestává z popisu úlohy, popisu vstupních dat, popisu požadované struktury výstupních dat a příkladů vstupu a požadovaného výstupu. Dále jsou u popisu úlohy uvedeny informace o časovém limitu, během něhož musí program vyřešit zadaný problém pro daná vstupní data, a o maximální možné velikosti souboru se zdrojovým kódem odeslaného uživatelem k validaci. Zadání úlohy rovněž obsahuje seznam programovacích jazyků, v nichž může být programové řešení dané úlohy napsáno.

2.4.2 Možné výsledky validace úloh

Systém Sphere Online Judge podle výsledku proběhlé validace uživateli oznámí jeden z následujících výsledků validace [8]:

- **Compile Error** - v případě syntaktické chyby v odeslaném řešení nebo pokud uživatel nedodržel závazné pokyny;
- **Time Limit Exceeded** - v případě, že program nevyřeší řešenou úlohu pro daná vstupní data v časovém limitu, který je uveden v zadání každé z úloh;
- **Runtime Error** - v případě, že během vykonávání programu dojde k chybě, například vyhození výjimky;
- **Wrong Answer** - v případě, kdy se výstup hodnoceného programu neshoduje se vzorovým výstupem pro danou úlohu a daná vzorová vstupní data;
- **Accepted** - v případě, kdy výstup hodnoceného programu je správný, tedy pokud se shoduje se vzorovým výstupem pro danou úlohu a daná vzorová vstupní data;

V případě chyby Runtime Error je uživateli zobrazena zpráva, kde je chyba blíže specifikována. Může se jednat o následující stavy:

- **SIGSEGV** - v případě, kdy program přistupuje k neplatné referenční proměnné nebo dojde k porušení ochrany paměti (anglicky „segmentation fault“);
- **SIGFXSZ** - v případě, kdy program vytiskne příliš mnoho dat na výstup;
- **SIGABRT** - v případě přerušení programu z důvodu fatální chyby (například v C++ není splněn predikát obsažený v příkazu `assert`);
- **SIGFPE** - v případě chyby výpočtu s plovoucí desetinnou čárkou (například dělení nulou);
- **NZEC** - v případě, kdy návratový kód spuštěného programu není roven nule (například pokud v případě programovacího jazyka C metoda `main` vrací jinou hodnotu než nula, v případě programovacího jazyka Java může jít o případ, kdy je během běhu programu vyhozena výjimka);
- **Other** - v případě jiného problému, který vedl k předčasnému ukončení programu.

2.5 TopCoder

Systém TopCoder (viz obrázek 2.6) je provozován společností Appirio, která nabízí služby v oblasti crowdsourcingu [9].

Stěžejním cílem portálu je crowdsourcing, kdy uživatelé v rámci soutěží vykonávají nějaký úkol v zájmu společnosti, která danou soutěž vypsala. Společnost po skončení soutěže vybere nejlepší řešení a jejich autory finančně odmění. Nejlepší řešení pak společnost může použít v rámci své podnikatelské aktivity.

Obrázek 2.6: Logo systému TopCoder
Zdroj: `http://topcoder-qa.com` [cit. 2016/10/20]

Systém TopCoder se zaměřuje na hledání programátorů a dalších IT specialistů, kteří řeší výzvy vypisované zákazníky systému. V současné době komunita systému TopCoder zahrnuje jeden milion členů a ročně je vypsáno více než 7 tisíc crowdsourcingových výzev [9].

Systém rovněž obsahuje systém pro nácvik programování nazvaný TopCoder Arena. Systém v době psaní této části práce (říjen 2016) podporuje pouze 5 programovacích jazyků, kterými jsou Java, C++, C#, VB.NET a Python.

2.5.1 Zadání úloh

Úlohy jsou rozděleny podle své obtížnosti na úlohy lehké, středně těžké a těžké. U každé úlohy je rovněž zobrazen počet bodů, který lze za vyřešení úlohy získat. Počet bodů je přímo úměrný obtížnosti dané úlohy.

Na rozdíl od jiných validátorů systém TopCoder nepožaduje napsání celého funkčního programu, ale požaduje pouze metodu, resp. funkci, která jako svoji návratovou hodnotu vrací řešení pro daná vstupní data. Vstupní data jsou metodě, resp. funkci, předána ve formě formálních parametrů. Vzhledem k tomuto zadání úlohy rovněž obsahuje definici požadované metody, resp. funkce. Obsah této definice se liší podle použitého programovacího jazyka, pro programovací jazyk Java jde o název třídy, název metody, typ formálních parametrů dané metody a typ návratové hodnoty.

Zadání dále obsahuje omezení na systémové prostředky, které mohou být použity odeslaným řešením. Jde o časový limit, během něhož program

musí vyřešit úlohu pro všechny sady vstupních dat, a maximální množství operační paměti, kterou může program alokovat.

V zadání jsou rovněž uvedena omezení, která jsou kladena na vstupní data. Poslední částí zadání jsou příklady vstupních dat a jim odpovídající výstupní data. Rovněž zde jsou v případě trénovacích úloh uvedena vysvětlení výpočtu pro daná vstupní data.

2.5.2 Možné výsledky validace úloh

Uživatel nejprve musí své řešení přeložit. V případě, kdy zdrojový kód obsahuje syntaktické chyby, je uživateli zobrazena chybová hláška, v níž je rovněž uvedena problematická část kódu. Pokud překlad proběhne bez problémů, je uživateli oznámeno, že překlad proběhl úspěšně.

V této chvíli uživatel může odeslat své řešení. Uživateli je zobrazeno, kolik za své řešení obdrží bodů. Počet bodů, které uživatel obdrží za vyřešení úlohy, je nepřímo úměrný času, který uživatel na vyřešení úlohy potřeboval [10], tedy čím déle uživatel úlohu řešil, tím méně bodů může získat. Po odeslání řešení není automaticky spuštěna validace, validaci musí manuálně spustit uživatel. Po validaci je uživateli zobrazena informace, zdali validace proběhla úspěšně nebo neúspěšně.

2.6 UVa Online Judge

Systém UVa Online Judge (viz obrázek 2.7) je provozován španělskou Universidad de Valladolid a je zaměřen na nácvik programování a přípravu na ACM soutěže. Systém v době psaní této části (říjen 2016) podporuje 6 programovacích jazyků, kterými jsou ANSI C, Java, C++, C++11, Pascal a Python. Na serveru jsou, podobně jako na jiných obdobných serverech, pořádány soutěže v programování.

Obrázek 2.7: Logo systému UVa Online Judge
Zdroj: `http://disi.unal.edu.co` [cit. 2016/11/10]

2.6.1 Zadání úloh

Úlohy jsou na serveru UVa Online Judge rozděleny dle svého původu. Jde například o úlohy z proběhlých soutěží, úlohy typu „Programming Challenges“ a další. Úlohy nejsou rozděleny podle své obtížnosti nebo tématu, kterého se úloha týká.

Zadání každé úlohy sestává z popisu problému, popisu struktury vstupních dat, požadované struktury výstupních dat, ukázkové sady vstupních dat a k této sadě odpovídajících výstupních data. Zadání úlohy dále obsahuje časový limit, během něhož musí program vyřešit úlohu pro všechny sady vstupních dat. V zadání úloh jsou rovněž dostupné statistiky, v nichž se může uživatel informovat o výsledcích validace všech odeslaných řešení všech uživatelů pro konkrétní úlohu a dále o programovacích jazycích, v nichž byla odeslaná řešení realizována.

Ve statistikách je uživatel rovněž informován o dvaceti nejlepších úspěšných řešeních z hlediska doby jejich běhu. Uživatel se může podívat, v jakých programovacích jazycích byla nejlepší řešení realizována a také na čas nutný pro běh těchto řešení.

2.6.2 Možné výsledky validace úloh

Systém UVa Online Judge uživateli po validaci odeslaného řešení oznámí jeden z výsledků [12]:

- **Accepted** - v případě, kdy výstup hodnoceného programu je správný, tedy pokud se shoduje se vzorovým výstupem pro danou úlohu a daná vzorová vstupní data;
- **Presentation Error** - v případě, kdy je výstup programu správný, ale neodpovídá požadované struktuře výstupu (například přebývající netisknutelný znak);
- **Wrong Answer** - v případě, kdy se výstup hodnoceného programu neshoduje se vzorovým výstupem pro danou úlohu a daná vzorová vstupní data;
- **Compilation Error** - v případě syntaktické chyby v odeslaném řešení nebo pokud uživatel nedodržel závazné pokyny (například v případě programovacího jazyka Java není třída, ve které je uložena metoda `main`, pojmenovaná `Main`);
- **Runtime Error** - v případě, že během vykonávání programu dojde k chybě, například vyhození výjimky;

- **Time Limit Exceeded** - v případě, že program nevyřeší řešenou úlohu pro daná vstupní data v časovém limitu, který je uveden v zadání každé z úloh;
- **Memory Limit Exceeded** - v případě, kdy program používá více operační paměti, než je povoleno;
- **Output Limit Exceeded** - v případě, kdy program na výstup vypisuje příliš mnoho informací;
- **Submission Error** - v případě chyby při odesílání řešení na server;
- **Restricted Function** - v případě, kdy program používá funkci, která je potenciálně nebezpečná a je proto zakázána;
- **Can't Be Judged** - v případě, kdy server nemá k dispozici vstupní a výstupní validační data pro kontrolu uživatelem odeslaného řešení.

2.7 Timus Online Judge

Ruský systém Timus Online Judge je provozován Uralskou federální univerzitou. Systém je zaměřen na nácvik programování a přípravu na ACM soutěže, z nichž systém přejímá některé úlohy do svého katalogu úloh. Systém Timus Online Judge v době psaní této části práce (říjen 2016) podporuje 11 programovacích jazyků, kterými jsou C, C++, Pascal, Java, C#, Go, Python, Ruby, Haskell, Scala a Rust [11].

2.7.1 Zadání úloh

Úlohy jsou na serveru Timus Online Judge rozděleny dle více kritérií. Na serveru nalezneme úlohy rozdělené podle témat, jehož se jednotlivé úlohy týkají (dynamické programování, datové struktury, grafové úlohy, ...), nebo podle soutěží, z nichž byly úlohy přebrány.

U každé úlohy je uvedeno, kolik uživatelů úlohu úspěšně vyřešilo a numericky vyjádřená obtížnost úlohy. Uživatel si může zobrazit výsledky ostatních uživatelů, kde je zobrazen čas, po který program běžel a programovací jazyk, v němž bylo napsáno programové řešení dané úlohy, avšak uživateli nejsou dostupné zdrojové kódy programových řešení ostatních uživatelů.

Zadání úloh obsahuje „příběh v pozadí“, popis úlohy, popis vstupních dat a na vstupní data kladená omezení, popis požadované struktury výstupu programu, omezení na čas, po který může program běžet a maximální množství paměti, které může program alokovat.

2.7.2 Možné výsledky validace úloh

Podle výsledku validace odeslaného řešení je uživateli oznámen jeden z následujících výsledků [11]:

- **Accepted** - v případě, kdy výstup hodnoceného programu je správný, tedy pokud se shoduje se vzorovým výstupem pro danou úlohu a daná vzorová vstupní data;
- **Compilation Error** - v případě syntaktické chyby v odeslaném řešení nebo pokud uživatel nedodržel závazné pokyny (například v případě programovacího jazyka Java zdrojový kód obsahuje více než jednu veřejnou třídu);
- **Wrong Answer** - v případě, kdy se výstup hodnoceného programu neshoduje se vzorovým výstupem pro danou úlohu a daná vzorová vstupní data;
- **Time Limit Exceeded** - v případě, že program nevyřeší řešenou úlohu pro daná vstupní data v časovém limitu, který je uveden v zadání každé z úloh;
- **Memory Limit Exceeded** - v případě, kdy program používá více operační paměti, než je povoleno;
- **Output Limit Exceeded** - v případě, kdy program na výstup vypisuje příliš mnoho informací;
- **Idleness Limit** - v případě, kdy je program příliš dlouho nečinný;
- **Runtime Error** - v případě, že během vykonávání programu dojde k chybě, například vyhození výjimky;
- **Restricted Function** - v případě, kdy program používá funkci, která je potenciálně nebezpečná a je proto zakázána (například komunikace po síti nebo přístup k souborům).

Každá úspěšně zvalidovaná úloha je ohodnocena, hodnocení je spočítáno podle doby, kterou validovaný program potřeboval pro řešení úlohy [11].

3 Vybrané úlohy vhodné pro výuku programování

Vybrané úlohy vhodné pro výuku programování na počátku bakalářského studia jsou uvedeny v tabulce 3.1. Úlohy byly vybrány z katalogu úloh validátoru UVa Online Judge, neboť tento systém je využíván v rámci některých předmětů vyučovaných na Katedře informatiky a výpočetní techniky Fakulty aplikovaných věd Západočeské univerzity v Plzni a někteří studenti by proto měli mít alespoň základní znalost práce s tímto serverem.

Z hlediska výuky programování můžeme za výhodu považovat, že uživatelé systému nemají k dispozici zvalidovaná řešení jiných uživatelů a také v zadání úloh není explicitně uvedeno, které oblasti se daná úloha týká. Studenti tedy musí samostatně vyřešit, jaké datové struktury a algoritmy použijí.

Zdrojové kódy programového řešení všech úloh jsou umístěny na přiloženém CD.

Název úlohy	**Číslo úlohy**	**Téma úlohy**
Where's Waldorf?	10010	Řetězce, pole
Flip-Flop the Squarelotron	10016	Pole
Automated Judge Script	10188	Řetězce
Strategy Game	12959	Pole
Newspaper	11340	Řetězce
Huatuos Medicine	12992	Proměnné nebo rekurze
Tri-du	12952	Podmínky
Joseph's Cousin	10015	Zřetězený spojový seznam
Knuth's Permutation	10063	Rekurze
Printer Queue	12100	Prioritní fronta
Erdös Numbers	10044	Nejkratší cesta v grafu
Prime Path	12101	Nejkratší cesta v grafu
Counting Stars	11244	Prohledávání do hloubky (DFS)
Shortest Names	12506	Datová struktura Trie
All Roads Lead Where?	10009	Nejkratší cesta v grafu
Bicoloring	10004	Určení bipartity grafu
The Tourist Guide	10099	Dynamické programování
Vacation	10192	Nejdelší společná podposloupnost
Sending Email	10986	Nejkratší cesta v grafu
Expensive Subway	11710	Minimální kostra grafu

Tabulka 3.1: Úlohy vybrané pro výuku programování

3.1 Joseph's Cousin

3.1.1 Originální zadání úlohy

10015 - Joseph's Cousin

The Josephs problem is notoriously known. For those who are not familiar with the problem, among n people numbered $1; 2; \ldots; n$, standing in circle every mth is going to be executed and only the life of the last remaining person will be saved. Joseph was smart enough to choose the position of the last remaining person, thus saving his life to give the message about the incident.

Although many good programmers have been saved since Joseph spread out this information, Josephs Cousin introduced a new variant of the malignant game. This insane character is known for its barbarian ideas and wishes to clean up the world from silly programmers. We had to infiltrate some the agents of the ACM in order to know the process in this new mortal game.

In order to save yourself from this evil practice, you must develop a tool capable of predicting which person will be saved.

The Destructive Process

The persons are eliminated in a very peculiar order; m is a dynamical variable, which each time takes a different value corresponding to the prime numbers succession $(2, 3, 5, 7, \ldots)$. So in order to kill the i-th person, Joseph's cousin counts up to the i-th prime.

Input

It consists of separate lines containing $n[1..3501]$, and finishes with a '0'.

Output

The output will consist in separate lines containing the position of the person which life will be saved.

Sample Input

```
6
0
```

Sample Output

```
4
```

Originální zadání úlohy je dostupné na adrese: `https://uva.onlinejudge.org/external/100/10015.pdf` [cit. 2016/11/02].

3.1.2 Česká verze zadání úlohy

10015 - Josefova sestřenice

Josef se svou sestřenicí nevychází příliš dobře. Je tomu tak i z toho důvodu, že Josefova sestřenice pořádá prazvláštní rituály, mezi něž patří i „vražedné kolečko“. Během tohoto rituálu n lidí očíslovaných $1; 2; \ldots; n$ stojí v kruhu a jsou postupně popravováni. Pouze poslední osoba z kruhu je ušetřena a přežije. Josef svoji sestřenici přelstil a vždy si stoupl v kruhu tak, aby byl poslední, a proto dosud vždy přežil. Díky tomu nás mohl informovat o podivném rituálu své sestřenice.

Mnoho životů programátorů bylo dosud zachráněno, nicméně Josefova sestřenice, která je známá především díky svým barbarským nápadům, mezi něž patří snaha vyhladit všechny špatné programátory na světě, vymyslela novou verzi tohoto příšerného rituálu. Abychom se dozvěděli nová pravidla tohoto rituálu, museli jsme infiltrovat několik našich agentů.

Abyste se zachránili před touto ďábelskou praktikou, musíte vytvořit program určující pozici osoby, která bude zachráněna a jako jediná z kruhu přežije.

Vražedný proces

Osoby jsou popravovány ve velice podivném pořadí, kdy m je proměnná, která po každé popravě nabývá jiné hodnoty. Hodnoty proměnné m odpovídají vzestupně seřazeným prvočíslům $(2, 3, 5, 7, \ldots)$. Josefova sestřenice následně počítá do tohoto prvočísla. Osoba, jejíž pozice odpovídá tomuto prvočíslu, je popravena a celý smrtící proces se opakuje, jen proměnná m nyní obsahuje další prvočíslo.

Vstup

Vstup sestává z řádek obsahující číslo $n[1..3501]$, z nichž každá řádka představuje jeden testovací případ. Vstup končí znakem „0“.

Výstup

Výstupem je číslo pozice osoby, která jako jediná nebude popravena. Pro každý testovací případ musí být číslo pozice uvedeno na novém řádku.

Vzorový vstup

```
6
0
```

Vzorový výstup

```
4
```

3.1.3 Analýza úlohy

Ze zadání úlohy je zřejmé, že naším úkolem je zvolit vhodnou datovou strukturu, která umožňuje zřetězený pohyb po elementech uložených v této struktuře. V rámci dané datové struktury musíme postupovat v kruhu, kdy se vždy posuneme o m pozic a osobu na této pozici zabijeme.

Například, pokud bychom řešili tuto úlohu pro případ šesti osob, na počátku bychom měli 6 osob očíslovaných 1, 2, 3, 4, 5, 6. V prvním kole je proměnná m rovna prvními prvočíslu, tedy $m = 2$. Počítáme od první osoby a počítání zastavíme u druhé osoby, kterou odstraníme ze seznamu přeživších osob. V dalším kole je m rovna hodnotě 3. Nyní začínáme počítat od osoby, která je další v pořadí po osobě zabité v předchozím kole. Nyní tedy počítáme od osoby 3, pokračujeme přes osobu 4 a zastavíme se na osobě 5, kterou zabijeme. V dalším kole je $m = 5$ a odpočítáváme od osoby číslo 6 a pokračujeme přes osoby s čísly 1, 3, 4, 6. Odpočítávání zastavíme u osoby 6, tu proto odstraníme ze seznamu. Analogickým postupem pokračujeme do doby, kdy náš seznam obsahuje poslední přeživší osobu, která jako jediná přežije.

Pro reprezentaci dat lze použít zřetězený spojový seznam, který umožňuje snadno procházet osoby v kruhu a zároveň i osoby odstranit, obojí lze dohromady provést s asymptotickou časovou složitostí $\mathcal{O}(n)$.

Není nutné simulovat postupné počítání v kruhu stojících osob, obzvláště v případě, kdy bychom spojový seznam prošli několikrát. Počet pozic, o který se máme posunout, je kongruentní modulo n k aktuálnímu prvočíslu uloženému v proměnné m. Hodnota n odpovídá aktuálnímu počtu prvků uložených ve spojovém seznamu. Počet pozic, o něž se musíme posunout, tedy odpovídá zbytku po celočíselném dělení m mod n.

3.1.4 Algoritmy vhodné pro řešení úlohy

Hlavním úskalím úlohy je volba vhodného algoritmu pro generování prvočísel vzhledem k časovému limitu na řešení úlohy. Časový limit v případě této úlohy činí na první pohled velkorysých 10 sekund, nicméně je nutné počítat s relativně vysokou maximální hodnotou proměnné n udávající počet prvočísel, které je třeba vygenerovat.

Řešení hrubou silou

V případě řešení hrubou silou (viz algoritmus 3.1) vyjdeme z definice prvočísla - číslo x je prvočíslo právě tehdy, když je beze zbytku dělitelné pouze číslem jedna a sebou samým. V případě tohoto algoritmu tedy prvočísel-

nost čísla x testujeme dělením čísla všemi čísly, která připadají v úvahu jako možní bezezbytkoví dělitelé testovaného čísla. Možnými děliteli jsou celá čísla patřící do intervalu $< 2, \sqrt{x} >$.

Asymptotická složitost algoritmu činí $\mathcal{O}(n \times \sqrt{n})$.

Algoritmus 3.1: Algoritmus hledání prvočísel hrubou silou

Data: Proměnná `pozadovano` obsahující požadovaný počet vygenerovaných prvočísel

Výsledek: Pole vygenerovaných prvočísel

```
1  int pocet = 0; // počet dosud vygenerovaných prvočísel
2  int[] pole; // pole dosud vygenerovaných prvočísel
3  int cislo = 2; // aktuálně testované číslo
4  while pocet < pozadovano do
5      boolean vysledek = true; // výsledek testu prvočíselnosti
       // vyzkoušíme všechny možné dělitele testovaného čísla
          z intervalu < 2, √cislo >
6      int i = 2;
7      for i ≤ √cislo do
8          if cislo mod i = 0 then
               // pokud je testované číslo beze zbytku dělitelné i,
                  nemůže jít o prvočíslo
9              vysledek = false;
10             break;
11         end
12         i = i + 1;
13     end
       // pokud je testované číslo prvočíslem, vložíme jej do pole
          vygenerovaných prvočísel a inkrementujeme čítač počtu dosud
          vygenerovaných prvočísel
14     if vysledek = true then
15         pole[pocet] = cislo;
16         pocet = pocet + 1;
17     end
18     cislo = cislo + 1;
19 end
```

Eratosthenovo síto

Eratosthenovo síto (viz algoritmus 3.2) je relativně jednoduchý algoritmus pro generování prvočísel. Algoritmus nehledá zadaný počet prvočísel, namísto toho algoritmus hledá všechna prvočísla do zadané maximální hodnoty [18].

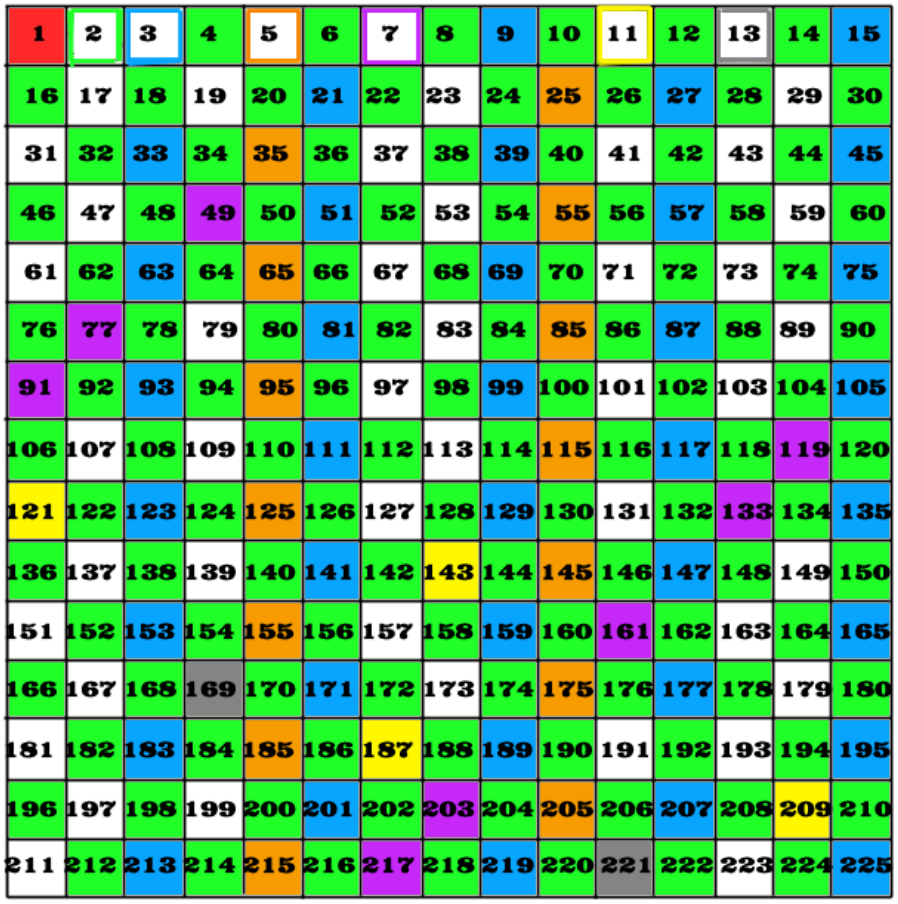
Algoritmus 3.2: Eratosthenovo síto [18]

Data: Proměnná `horniMez` reprezentující horní mez intervalu, v němž jsou prvočísla hledána
Výsledek: Pole logických hodnot, kde hodnota na indexu i značí, zdali je číslo i prvočíslem

```
1  boolean[] pole; // pole s hodnotami prvočíselnosti
2  pole[1] = false; // jedna není prvočíslo
3  int i = 2;
   // naplnění pole hodnotou true (značí, že čísla jsou prvočíselná)
4  for i ≤ horniMez do
5  |  pole[i] = true;
6  |  i = i + 1;
7  end
8  int p = 2;
9  while p ≤ √horniMez do
   |  // označení násobků prvočísla jako neprvočíselných čísel
10 |  int n = 2;
11 |  for p × n ≤ horniMez do
12 |  |  pole[p × n] = false;
13 |  |  n = n + 1;
14 |  end
   |  // nalezení dalšího prvočísla
15 |  int x = p + 1;
16 |  while pole[x] ≠ true do
17 |  |  x = x + 1;
18 |  end
19 |  p = x;
20 end
```

Algoritmus pracuje s polem logických hodnot, kde hodnota na indexu i značí, zdali je číslo i prvočíslem. Na počátku algoritmus alokuje pole logických hodnot o délce n, kde n značí horní mez intervalu, v němž jsou hledána prvočísla. Toto pole je naplněno hodnotami true s výjimkou prvního indexu,

kam je uložena hodnota `FALSE` (číslo jedna není obecně považováno za prvočíslo). Následně algoritmus z pole vybírá nejnižší index, na němž je uložena hodnota `TRUE` (toto číslo je prvočíslem). Algoritmus dále všechny násobky vybraného prvočísla označí hodnotou `FALSE`, neboť tyto násobky nemohou být prvočíselné.



Obrázek 3.1: Princip funkce Eratosthenova síta
Zdroj: `www.murderousmaths.co.uk/books/MMoE/new200sieve.gif` [cit. 2016/11/02]

Princip funkce algoritmu je zřejmý z obrázku 3.1. V prvním kroku je vybráno číslo dva, které je prvočíslem. Dále jsou označeny násobky dvou (v obrázku zeleně) jako čísla, která nejsou prvočísly. V dalším kroku algoritmu je vybráno číslo 3 a jeho násobky jsou označeny jako neprvočíselná čísla (v obrázku modře). Algoritmus takto postupuje do doby, kdy vybrané prvočíslo není větší než odmocnina z horní meze intervalu, v němž jsou prvočísla hledána. Ve chvíli, kdy algoritmus vybere první prvočísla větší než odmocnina z horní meze intervalu, v němž jsou prvočísla hledána, již není

prováděno označování násobků vybraného prvočísla jako neprvočíselných čísel a všechna zbývající čísla jsou prvočísla.

Asymptotická časová složitost algoritmu činí $\mathcal{O}(n \log \log n)$ [18].

Další algoritmy

Dále jsou dostupné další algoritmy založené na principu „prosívání“ seznamu čísel. Patří mezi ně například moderní algoritmus Atkinova síta, který dosahuje lepší asymptotické časové složitosti $\mathcal{O}(\frac{n}{\log \log n})$ [13], ovšem tento algoritmus využívá sofistikovanějších principů a je složitější na pochopení a implementaci. Z tohoto důvodu není vhodný k implementaci začínajícími programátory.

3.1.5 Řešení úlohy

V rámci řešení úlohy bylo v první řadě nezbytné zvolit vhodný algoritmus pro generování prvočísel, vzhledem k lepší asymptotické časové složitosti byl zvolen algoritmus Eratosthenova síta. Pro reprezentaci osob stojících v kruhu byl použit spojový seznam reprezentovaný kolekcí `LinkedList` ze standardní knihovny `java.util` programovacího jazyka Java.

3.1.6 Výsledek validace

Úloha byla úspěšně zvalidována, čas běhu činil 2,270 sekundy (viz obrázek 3.2).

#	Problem	Verdict	Language	Run Time	Submission Date
19209572	10015 Joseph's Cousin	Accepted	JAVA	2.270	2017-04-20 13:26:44

Obrázek 3.2: Zpráva o validaci úlohy Joseph's Cousin

3.2 All Roads Lead Where?

3.2.1 Originální zadání úlohy

10009 - All Roads Lead Where?

There is an ancient saying that „All Roads Lead to Rome“. If this were true, then there is a simple algorithm for finding a path between any two cities. To go from city A to city B, a traveller could take a road from A to Rome, then from Rome to B. Of course, a shorter route may exist.

The network of roads in the Roman Empire had a simple structure: beginning at Rome, a number of roads extended to the nearby cities. From these cities, more roads extended to the next further cities, and so on. Thus, the cities could be thought of as existing in levels around Rome, with cities in the ith level only connected to cities in the $i-1$th and $i+1$th levels (Rome was considered to be at level 0). No loops existed in the road network. Any city in level i was connected to a single city in level $i-1$, but was connected to zero or more cities in level $i+1$. Thus, to get to Rome from a given city in level i, a traveller could simply walk along the single road leading to the connected $i-1$ level city, and repeat this process, with each step getting closer to Rome.

Given a network of roads and cities, your task is to find the shortest route between any two given cities, where distance is measured in the number of intervening cities.

Input

The first line is the number of test cases, followed by a blank line. The first line of each test case of the input contains two numbers in decimal notation separated by a single space. The first number (m) is the number of roads in the road network to be considered. The second number (n) represents the number of queries to follow later in the file. For each test case, in the next m lines in the input each contain the names of a pair of cities separated by a single space. A city name consists of one or more letters, the first of which is in uppercase. No two cities begin with the same letter. The name Rome always appears at least once in this section of input, for each test case; this city is considered to be at level 0, the lowest-numbered level. The pairs of names indicate that a road connects the two named cities. The first city named on a line exists in a lower level than the second named

city. The road structure obeys the rules described above. For each test case, no two lines of input in this section are repeated.
The next n lines, for each test case in the input each contain the names of a pair of cities separated by a single space. City names are as described above. These pairs of cities are the query pairs. Your task for each query pair is to find the shortest route from the first named city to the second. Each of the cities in a query pair is guaranteed to have appeared somewhere in the previous input section, for each test case, describing the road structure.
Each test case will be separated by a single line.

Output

In each test case, for each of the n query pairs, output a sequence of uppercase letters indicating the shortest route between the two query pair cities. The sequence must be output as consecutive letters, without intervening whitespace, on a single line. For each test case, the first output line corresponds to the first query pair, the second output line corresponds to the second query pair, and so on. The letters in each sequence indicate the first letter of the cities on the desired route between the query pair cities, including the query pair cities themselves. A city will never be paired with itself in a query.
Print a blank line between the outputs for two consecutive test cases.

Sample Input

```
 1
7 3
Rome Turin
Turin Venice
Turin Genoa
Rome Pisa
Pisa Florence
Venice Athens
Turin Milan
Turin Pisa
Milan Florence
Athens Genoa
```

Sample Output

```
TRP
MTRPF
AVTG
```

Originální zadání úlohy je dostupné na adrese: `https://uva.onlinejudge.org/external/100/10009.pdf` [cit. 2016/11/10].

3.2.2 Česká verze zadání úlohy

10009 - Kam vedou všechny cesty?

„Všechny cesty vedou do Říma," tedy alespoň to tvrdí jedno starověké rčení. Pokud by to byla pravda, pak lze sestavit jednoduchý algoritmus pro hledání tras mezi jakýmikoliv dvěma městy. Abychom se dostali z města A do města B, stačilo by jít z města A do Říma a poté z Říma do města B. Samozřejmě by mohla existovat kratší cesta.
Síť cest ve starověkém antickém impériu měla jednoduchou strukturu - některé cesty začínaly v Římě a vedly do blízkých měst, z těchto měst vedly další cesty do vzdálenějších měst a podobně. Vzhledem k tomuto lze města rozdělit do úrovní okolo Říma, kde města i-té úrovně byla cestami spojeny jen s městy $i-1$-té úrovně a $i+1$-té úrovně (Řím byl na úrovni 0). V systému cest neexistovaly žádné smyčky. Každé město na i-té úrovni bylo spojeno právě s jedním městem úrovně $i+1$ a s žádným nebo více městy úrovně $i-1$. Vzhledem k tomuto uspořádání měst a meziměstských cest mohl cestovatel, jehož cílem byl Řím, jednoduše jít po cestě k městu o úroveň nižší než mělo město, v němž se právě nacházel, a tento proces opakoval, dokud nedorazil do Říma.
Vaším úkolem je najít nejkratší cestu mezi dvěma zadanými městy, přičemž vzdálenost je rovna počtu navštívených měst.
Vstup
První řádka vstupu obsahuje počet testovacích případů a je následována prázdnou řádkou.
Na první řádce každého testovacího případu najdeme dvě dekadická čísla oddělená mezerou. První číslo m udává počet cest v síti dopravní infrastruktury, druhé číslo n pak udává počet dvojic měst, mezi nimiž máme najít cestu.
Dále jsou uvedeny dvojice měst, které jsou spojeny cestou. Tato města jsou oddělena mezerou. Název města sestává z jednoho a více písmen, první písmeno názvu města je velké. Je zaručeno, že žádná dvě města nebudou mít stejné počáteční písmeno svého názvu. Město `Rome (Řím)` je uvedeno alespoň jednou v seznamu měst a je umístěno na nulté úrovni v hierarchii měst, která byla popsána výše. První město na řádce je vždy město na nižší úrovni než město uvedené druhé. Je zaručeno, že se nebudou dvojice měst ve vstupu opakovat.

Dalších n řádek obsahuje dvojice měst oddělené mezerou, mezi kterými budeme hledat nejkratší cestu. Každé město uvedené v dvojicích měst, mezi nimiž hledáme cestu, bylo alespoň jednou uvedeno při definování cest mezi městy.
Každý testovací případ je oddělen prázdnou řádkou.
Výstup
Pro každý testovací případ vypíšeme několik sekvencí velkých písmen složených z počátečních písmen měst na nalezené nejkratší cestě. Sekvence nesmí obsahovat mezery a musí být vypsána na jedné řádce. Sekvence obsahuje i počáteční písmena počátečního a koncového města. Jednotlivé vypsané sekvence jsou vždy uvedeny na nové řádce.
Výsledné sekvence cest jsou pro každý testovací případ oddělené prázdnou řádkou.
Vzorový vstup

```
1
7 3
Rome Turin
Turin Venice
Turin Genoa
Rome Pisa
Pisa Florence
Venice Athens
Turin Milan
Turin Pisa
Milan Florence
Athens Genoa
```

Vzorový výstup

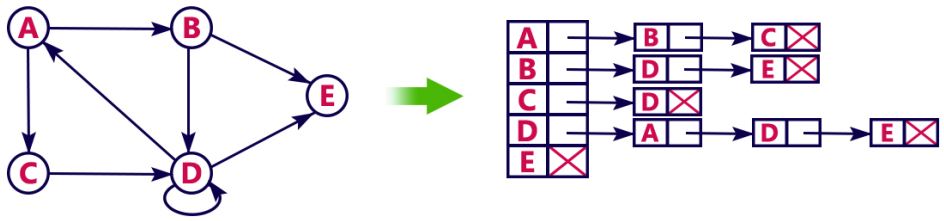
```
TRP
MTRPF
AVTG
```

3.2.3 Analýza úlohy

Je zřejmé, že jde o grafovou úlohu. V první řadě je tedy třeba vybrat vhodnou datovou strukturu pro reprezentaci grafu. Dále bude třeba vybrat vhodný algoritmus pro nalezení nejkratších cest v grafu.

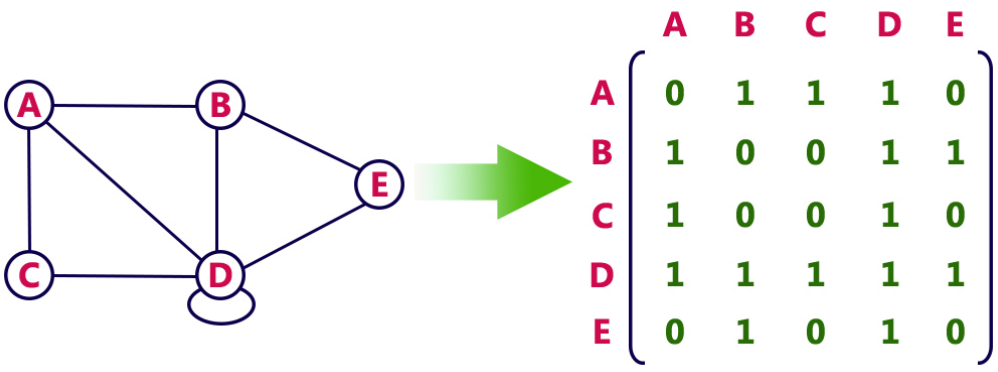
Při výběru datových struktur pro reprezentaci grafu máme na výběr ze dvou možností, kterými jsou reprezentace spojovým seznamem a reprezentace maticí sousednosti. Reprezentace grafu spojovým seznamem (viz obrá-

zek 3.3) je vhodná především pro grafy, kde není příliš mnoho hran mezi vrcholy. V případě reprezentace grafu spojovým seznamem obsahuje datová struktura reprezentující vrchol grafu spojový seznam obsahující vrcholy sousedící s daným vrcholem, nebo jinak řečeno vrcholy, s nimiž daný vrchol sdílí hrany grafu.



Obrázek 3.3: Reprezentace grafu spojovým seznamem
Zdroj: `http://btechsmartclass.com/DS/images/Graph%20Adjacency%20List.jpg` [cit. 2016/11/10]

Reprezentace grafu maticí sousednosti (viz obrázek 3.4) je naopak vhodná pro grafy obsahující vrcholy sousedící s mnoha jinými vrcholy. Pokud bychom použili reprezentaci grafu maticí sousednosti pro graf obsahující jen málo hran, dostali bychom řídkou matici sousednosti (za řídkou matici považujeme takovou matici, jejíž prvky jsou z převážné části nulové), což je v případě uložení matice v „hustém formátu“ paměťově značně neefektivní.



Obrázek 3.4: Reprezentace neorientovaného grafu maticí sousednosti
Zdroj: `http://btechsmartclass.com/DS/images/Graph%20Adjacency%20Matrix%201.jpg` [cit. 2016/11/10]

Matici sousednosti sestavíme poměrně jednoduše - nejprve si očíslujeme vrcholy grafu od jedné vzestupně. Čísla vrcholů grafu budou představo-

vat řádky, resp. sloupce matice sousednosti. Následně matici naplníme daty podle vztahu 3.1, v němž i představuje řádek a j sloupec matice:

$$m_{i,j} = \begin{cases} 0 \ldots & \text{pokud vrcholy } i \text{ a } j \text{ nejsou spojené hranou,} \\ 1 \ldots & \text{pokud vrcholy } i \text{ a } j \text{ jsou spojené hranou.} \end{cases} \tag{3.1}$$

V případě neorientovaných grafů je výsledná matice sousednosti symetrická (viz obrázek 3.4). V opačném případě, kdy uvažujeme orientovaný graf, není matice sousednosti obecně symetrická. Pokud bychom uvažovali ohodnocený graf, v matici bychom namísto pevné hodnoty 1 značící sousedící vrcholy umístili ohodnocení hrany, která dané dva vrcholy spojuje.

Hlavním úkolem je nalezení nejkratší cesty v grafu, kterou po jejím nalezení vypíšeme v požadovaném formátu.

3.2.4 Algoritmy vhodné pro řešení úlohy

Stěžejní částí úlohy je výběr a implementace vhodného algoritmu pro hledání nejkratší cesty v grafu. Ze zadání víme, že graf reprezentující úlohu bude acyklický, neorientovaný a neohodnocený. Tyto atributy jsou důležité pro výběr vhodného algoritmu pro nalezení nejkratší cesty v grafu.

Některé algoritmy pro hledání nejkratší cesty v grafu vyžadují ohodnocený graf. Ohodnocený graf v našem případě získáme poměrně jednoduše. Postačí přiřadit všem hranám v grafu ohodnocení hrany rovno jedné.

Prohledávání do šířky (BFS)

Algoritmus prohledávání grafu do šířky (Breadth-first search, BFS; viz algoritmus 3.3) především popisuje strategii procházení grafu, nicméně jej lze využít i pro hledání nejkratší cesty v neohodnocených grafech [14]. V tomto případě předpokládáme, že všechny hrany grafu mají stejné ohodnocení, které je typicky rovno jedné. V případě, kdy bychom měli ohodnocený graf, lze toto omezení algoritmu překonat „rozsekáním“ hrany s ohodnocením n na n „jednotkových“ hran, mezi něž umístíme pomocné vrcholy [19].

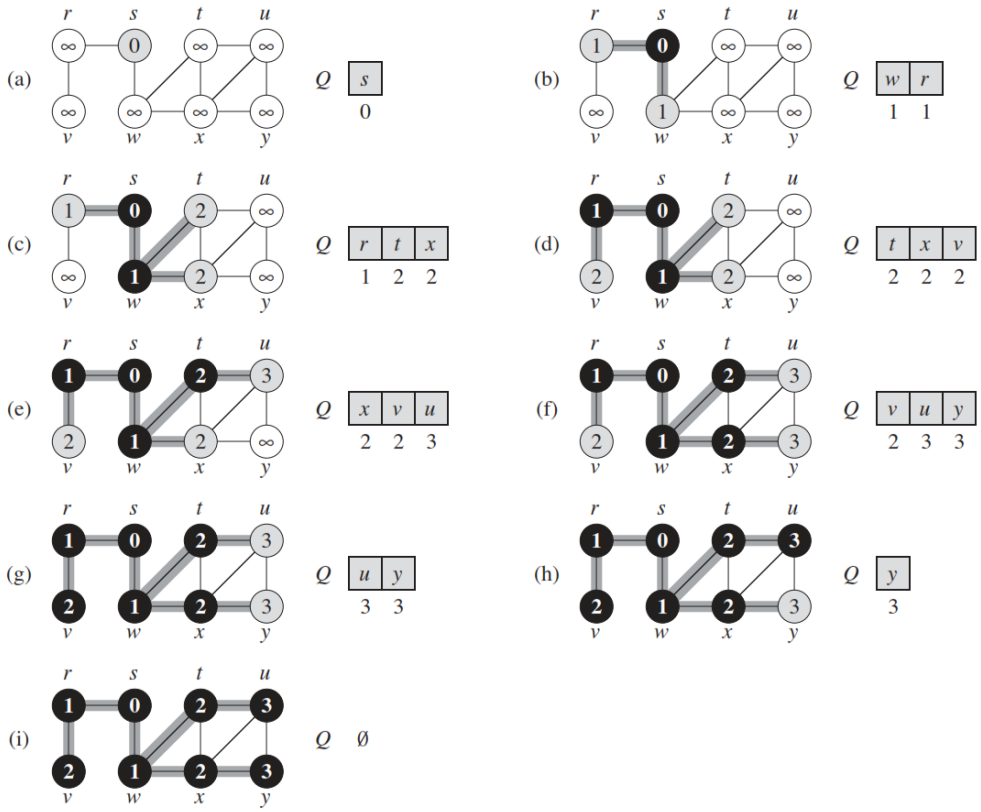
Princip algoritmu (viz obrázek 3.5) při prohledávání grafu je jednoduchý - nejprve navštívíme startovní vrchol s a do FIFO fronty (FIFO - First In First Out; první vrchol, který vstoupil do fronty, ji také jako první opustí) si uložíme sousední vrcholy vrcholu s. V dalším kroku vybíráme z fronty vrcholy, které sousedí s vrcholem s (v obrázku 3.5 jde o vrcholy r a w) a těmto vrcholům nastavíme vzdálenost 1 a předchůdce s. Do fronty si uložíme sousední vrcholy právě navštívených vrcholů. Z těchto vrcholů začneme

Algoritmus 3.3: Prohledávání grafu do šířky (BFS) [14]

Data: Graf G = (V, H), kde V jsou vrcholy a H hrany grafu, a startovní vrchol s

Výsledek: Pro každý vrchol grafu vzdálenost od vrcholu s a předchůdce na nejkratší cestě do vrcholu s

```
// pro každý vrchol grafu nastavíme vzdálenost ∞ a předchůdce NULL
foreach vrchol v ∈ V do
    v.vzdalenost = ∞;
    v.predchudce = NULL;
end
s.vzdalenost = 0;
// Inicializujeme frontu Q a vložíme do ní vrchol s
Fronta Q;
Q.push(s);
while Q není prázdná do
    u = Q.pop();
    foreach vrchol v sousedící su do
        // Pokud jsme vrchol ještě nenavštívili
        if v.vzdalenost ≠ ∞ then
            // Nastavíme vzdálenost
            v.vzdalenost = u.vzdalenost + 1;
            // Nastavíme předchůdce
            v.predchudce = u;
            // Vložíme do fronty
            Q.push(v);
        end
    end
end
```



Obrázek 3.5: Ukázka průběhu algoritmu BFS
Zdroj: `http://4.bp.blogspot.com/-cBfQheDKSqw/UL84AjQU7TI/AAAAAAAABB4/oTq7riy8Zng/s000/bfs.png` [cit. 2016/11/12]

opět navštěvovat sousední vrcholy (v obrázku 3.5 jde o vrcholy t, v, x), kterým nastavíme vzdálenost 2 a předchůdce podle toho, z jakého vrcholu jsme „přišli“. Takto pokračujeme do doby, než navštívíme všechny dostupné vrcholy [14]. V průběhu algoritmu každý vrchol navštívíme nejvýše jednou.

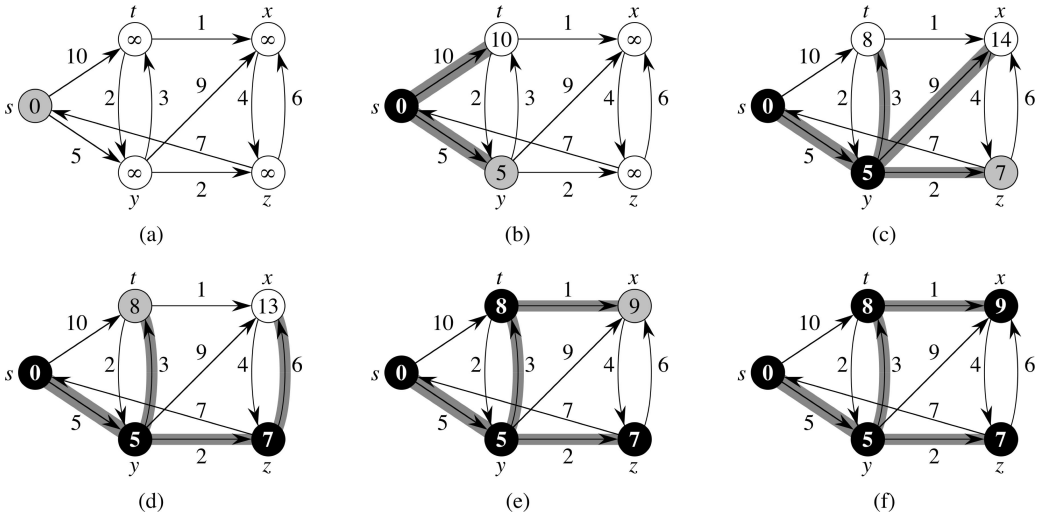
Asymptotická složitost algoritmu BFS činí v případě reprezentace grafu spojovým seznamem $\mathcal{O}(m + n)$, kde m značí počet vrcholů a n počet hran [14].

Dijkstrův algoritmus

Dijkstrův algoritmus (viz algoritmus 3.4) hledá nejkratší cesty v orientovaném ohodnoceném grafu ze zadaného počátečního vrcholu s do všech zbývajících vrcholů v grafu [14]. Předpokladem pro správnou funkčnost algoritmu je nezáporné ohodnocení hran v grafu [14]. Pokud graf obsahuje záporně ohodnocené hrany, je nutné pro hledání nejkratší cesty mezi vrcholy pou-

žít jiný algoritmus. Lze použít například Floyd-Warshallův algoritmus nebo Bellman-Fordův algoritmus, které jsou popsány dále.

Dijkstrův algoritmus lze využít i v neorientovaném grafu [19], neboť neorientovaný graf snadno převedeme na orientovaný - postačí mezi dvěma vrcholy v_1 a v_2, mezi nimiž vedla neorientovaná hrana, položit dvě orientované hrany, z nichž první povede z vrcholu v_1 do vrcholu v_2 a druhá vice versa, tj. z vrcholu v_2 do vrcholu v_1 (tento postup ovšem povede k vytvoření cyklů v grafu, což znemožňuje použití některých algoritmů, které vyžadují pro svou správnou funkcionalitu acyklický graf). V případě použití reprezentace grafu spojovým seznamem nebo maticí sousednosti, jak bylo popsáno výše, máme již grafy uložené v orientované podobě a lze Dijkstrův algoritmus použít rovnou bez nutnosti použití výše popsané úpravy hran grafu z neorientovaných na orientované.



Obrázek 3.6: Ukázka průběhu Dijkstrova algoritmu
Zdroj: `https://www.cs.indiana.edu/~achauhan/Teaching/B403/LectureNotes/images/10-dijkstra.jpg` [cit. 2016/11/12]

Průběh Dijkstrova algoritmu je zobrazen na obrázku 3.6. Vrcholu s nastavíme vzdálenost rovnu nule a ostatním vrcholům vzdálenost od vrcholu s rovnu nekonečnu. V průběhu algoritmu postupně z prioritní fronty odebíráme vrcholy s nejnižším ohodnocením a ty prohlašujeme za trvalé (k těmto vrcholům již známe nejkratší cestu; v obrázku 3.6 zobrazeny černě). Následně zkontrolujeme vzdálenosti sousedních vrcholů. Pokud má některý ze sousedních vrcholů vzdálenost od vrcholu s větší než součet vzdálenosti aktuálního vrcholu a ohodnocení příslušné hrany incidentní s aktuálním i daným sousedním vrcholem, aktualizujeme vzdálenost daného vrcholu (např. ve stavu (c) v obrázku měl vrchol t původní vzdálenost 10, avšak do vrcholu t se

Algoritmus 3.4: Dijkstrův algoritmus [19]

Data: Graf G = (V, H, c), kde V jsou vrcholy, H hrany grafu, c představuje ohodnocení hran; vrchol s

Výsledek: Pro každý vrchol grafu vzdálenost od vrcholu s a předchůdce na nejkratší cestě do vrcholu s

```
// pro každý vrchol grafu nastavíme vzdálenost ∞ a předchůdce NULL
foreach vrchol v ∈ V do
    v.vzdalenost = ∞;
    v.predchudce = NULL;
end
s.vzdalenost = 0;
// Inicializujeme prioritní frontu Q netrvalých bodů a vložíme do ní
   vrcholy grafu
Fronta Q = v ∈ V s prioritami v.vzdalenost;
// Inicializujeme prázdnou množinu P, kde budou uložené trvalé
   vrcholy
Množina P;
while Q není prázdná do
    // Vybereme vrchol s nejnižší prioritou
    u = Q.pop();
    // Vybraný vrchol prohlásíme za trvalý a proto jej vložíme do
       množiny P
    P.add(u);
    foreach vrchol v sousedící s u do
        // Pokud jsme nalezli kratší cestu do vrcholu v
        // c(u, v) značí ohodnocení hrany mezi vrcholy u a v
        if v.vzdalenost > u.vzdalenost + c(u, v) then
            v.vzdalenost = u.vzdalenost + c(u, v);
            v.predchudce = u;
            // Obnovíme správné pořadí ve frontě
            Q.reorder();
        end
    end
end
```

lze dostat z vrcholu y ve vzdálenosti od počátečního vrcholu s 5 po hraně s ohodnocením 3 a proto je vzdálenost vrcholu t rovna $5 + 3 = 8$).

Asymptotická časová složitost Dijkstrova algoritmu činí $\mathcal{O}(n^2)$, v případě implementace s použitím Fibonacciho haldy lze časovou složitost snížit na $\mathcal{O}(n \log n + m)$ [14].

Floyd-Warshallův algoritmus

Floyd-Warshallův algoritmus (viz algoritmus 3.5) slouží k nalezení nejkratších cest mezi každou dvojicí vrcholů grafu. Předpokladem pro správnou funkčnost algoritmu je orientovaný graf, ve kterém nesmí být obsaženy záporné cykly, avšak na rozdíl od Dijkstrova algoritmu může graf obsahovat záporně ohodnocené hrany [14].

Algoritmus 3.5: Floyd-Warshallův algoritmus [19]

Data: Matice sousednosti D^0
Výsledek: Matice D^n obsahující vzdálenosti mezi všemi dvojicemi vrcholů grafu

```
// Proběhne n iterací (n značí počet vrcholů v grafu)
for i = 1, ..., n do
    // Procházíme matici o rozměru n × n
    // u značí číslo řádku matice
    for u = 1, ..., n do
        // v značí číslo sloupce matice
        for v = 1, ..., n do
            // Pokud existuje kratší cesta, tak vložíme její délku do
               matice
            if D_{u,v} > D_{i,v} + D_{u,i} then
                D_{u,v} = D_{i,v} + D_{u,i};
            end
        end
    end
end
```

Na počátku běhu algoritmu očíslujeme n vrcholů čísly $1, \ldots, n$. Ve čtvercové matici D řádu n si budeme ukládat vzdálenosti mezi každou dvojicí vrcholů grafu. Výpočet matice D neproběhne „naráz“, ale proběhne iterativně, kdy v i-té iteraci určíme matici D^i. V matici D^i hodnota $d_{u,v}$, kde číslo u značí řádek, číslo v pak sloupec matice D^i, značí délku nejkratší cesty mezi vrcholy označenými čísly u a v, která prochází vrcholy označenými čísly

$1, \ldots, i$. Matice D^0 je vstupem algoritmu a značí matici sousednosti obsahující informaci o sousedících vrcholech a ohodnocení hran mezi nimi. Výpočet končí po n iteracích, kdy výsledná matice D^n obsahuje nejkratší cesty, které mohou vést přes všech n vrcholů grafu [19].

Pro výpočet prvků $d_{u,v}$ matice D^i v i-té iteraci algoritmu použijeme vztah 3.2

$$d_{u,v} = \min\{D^{i-1}_{u,v}, D^{i-1}_{u,i} + D^{i-1}_{i,v}\} \tag{3.2}$$

Pokud chceme pomocí Floyd-Warshallova algoritmu zjistit i vrcholy ležící na nejkratší cestě, musíme ještě počítat matici předchůdců. V této matici uchováváme pro každou dvojici vrcholů u, v nejvyšší číslo iterace i, v níž došlo ke změně matice D na řádku u a ve sloupci v, resp. jinak řečeno si pamatujeme vrchol, který leží na nejkratší cestě mezi vrcholy u, v. Nejkratší cestu pak můžeme z matice předchůdců snadno rekurzivně zrekonstruovat [19].

Asymptotická složitost Floyd-Warshallova algoritmu činí $\mathcal{O}(n^3)$, kde n značí počet vrcholů grafu. Floyd-Warshallův algoritmus je v reálných podmínkách rychlejší než Dijkstrův algoritmus v případě, kdy bychom Dijkstrův algoritmus spouštěli ze všech vrcholů grafu, abychom nalezli nejkratší cesty mezi všemi dvojicemi vrcholů grafu [14].

Bellman-Ford-Mooreův algoritmus

Bellman-Ford-Mooreův algoritmus (viz algoritmus 3.6), často uváděn jen jako Bellman-Fordův algoritmus, slouží pro nalezení nejkratší cesty mezi dvěma vrcholy v grafu, avšak narozdíl od algoritmů popsaných výše neklade na graf žádné požadavky na acykličnost nebo nezáporné ohodnocení hran. Bellman-Fordův algoritmus je schopen kromě nalezení nejkratší cesty v grafu obsahujícím cykly i záporné ohodnocení hran detekovat cyklus záporné délky [19].

Na počátku algoritmu nastavíme vzdálenost vrcholu s rovnu nule a vzdálenost ostatních vrcholů rovnu nekonečnu. Dále v průběhu algoritmu jsou opakovaně prováděny tzv. aktualizace hran („updaty“), které kontrolují korektnost hran a případně opravují nejvyšší odhad vzdálenosti nejkratší cesty. Hranu uv, vedoucí z vrcholu u do vrcholu v nazveme korektní hranou, pokud platí

$$dist[v] \leq dist[v] + c(uv), \tag{3.3}$$

kde $dist[v]$ značí vzdálenost vrcholu v od počátečního vrcholu s a $c(uv)$ značí ohodnocení hrany vedoucí z vrcholu u do vrcholu v, v opačném případě nazveme hranu uv nekorektní hranou [19].

Při update hrany dochází k výpočtu vzdálenosti vrcholu v podle vztahu 3.4.

$$dist[v] = \min\{dist[v] + c(uv)\} \tag{3.4}$$

Bellman-Fordův algoritmus v $n - 1$ iteracích updatuje všechny hrany grafu. Počet $n - 1$ odpovídá počtu hran na nejdelší možné cestě mezi n vrcholy, kde n značí počet vrcholů v grafu. Při implementaci je vhodné zjistit, zdali došlo ke změně vzdálenosti nějakého vrcholu [19]. Pokud by ke změně nedošlo, můžeme algoritmus ukončit, neboť ani v dalších krocích změna nenastane.

Pokud algoritmus doběhne do konce, tedy proběhne $n - 1$ iterací, je posledním krokem algoritmu kontrola přítomnosti cyklu záporné délky v grafu. Pokud se v grafu nachází cyklus záporné délky, bude některá z hran nekorektní [19]. Postačí tedy toto otestovat a v případě nalezení nekorektní hrany můžeme konstatovat přítomnost cyklu záporné délky v grafu. V opačném případě, kdy graf neobsahuje žádnou nekorektní hranu, nalezl Bellman-Fordův algoritmus nejkratší cestu mezi požadovanou dvojicí vrcholů.

V případě, kdy graf obsahuje cyklus záporné délky, mohli bychom neustále „chodit dokola“ po cyklu záporné délky, čímž bychom snižovali délku nejkratší cesty. Proto je nutné na konci algoritmu zkontrolovat korektnost všech hran. Proto v případě, kdy graf obsahuje cyklus záporné délky, nejkratší cesta v grafu neexistuje.

Asymptotická časová složitost Bellman-Fordova algoritmu činí $\mathcal{O}(nm)$, kde m značí počet vrcholů a n počet hran [14].

3.2.5 Řešení úlohy

Pro řešení úlohy byl zvolen Dijkstrův algoritmus vzhledem ke své příznivé časové asymptotické složitosti $\mathcal{O}(n^2)$. Dalším zvažovaným kandidátem byl Floyd-Warshallův algoritmus, který by musel být v každém testovacím případu spuštěn pouze jednou, neboť určí nejkratší cestu mezi všemi dvojicemi vrcholů v grafu, ovšem za cenu vyšší asymptotické složitosti $\mathcal{O}(n^3)$. Jako prioritní fronta byla zvolena generická kolekce `PriorityQueue` ze standardní knihovny `java.util` programovacího jazyka Java. Graf byl reprezentován spojovým seznamem, ohodnocení hran nebyla s ohledem na zadání úlohy ukládána (u všech hran obsažených v grafu bylo předpokládáno ohodnocení rovno jedné).

Algoritmus 3.6: Bellman-Fordův algoritmus [19]

Data: Graf G = (V, H, c), kde V jsou vrcholy, H hrany grafu, c představuje ohodnocení hran; počáteční vrchol s

Výsledek: Pro každý vrchol grafu vzdálenost od vrcholu s a předchůdce na nejkratší cestě do vrcholu s

```
   // pro každý vrchol grafu nastavíme vzdálenost ∞ a předchůdce NULL
1  foreach vrchol v ∈ V do
2  |  v.vzdalenost = ∞;
3  |  v.predchudce = NULL;
4  end
5  s.vzdalenost = 0;
   // Opakované updaty hran
6  for i = 1, ..., n-1 do
7  |  foreach hrana e ∈ H do
8  |  |  u = e.pocatek; // u je počáteční vrchol hrany
9  |  |  v = e.konec; // v je koncový vrchol hrany
10 |  |  if v.vzdalenost > u.vzdalenost + c(e) then
11 |  |  |  v.vzdalenost = u.vzdalenost + c(e);
12 |  |  |  v.predchudce = u;
13 |  |  end
14 |  end
15 end
   // Kontrola přítomnosti cyklů záporné délky v grafu
16 foreach hrana e ∈ H do
   |  // u je počáteční vrchol hrany
17 |  u = e.pocatek;
   |  // v je koncový vrchol hrany
18 |  v = e.konec;
19 |  if v.vzdalenost > u.vzdalenost + c(e) then
   |  |  // Graf obsahuje cyklus záporné délky
20 |  |  exit;
21 |  end
22 end
```

3.2.6 Výsledek validace

Úloha byla úspěšně zvalidována, čas běhu činil 0,140 sekundy (viz obrázek 3.7).

#	Problem	Verdict	Language	Run Time	Submission Date
19209610	10009 All Roads Lead Where?	Accepted	JAVA	0.140	2017-04-20 13:31:01

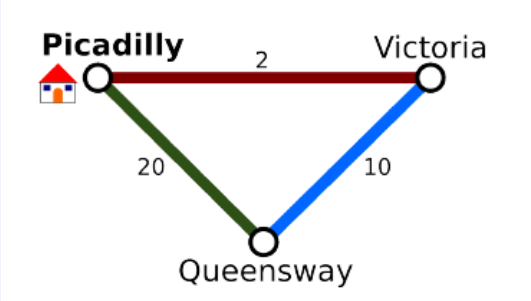
Obrázek 3.7: Zpráva o validaci úlohy All Roads Lead Where?

3.3 Expensive Subway

3.3.1 Originální zadání úlohy

11710 - Expensive Subway

Peter lives in *Expensive City*, one of the most expensive cities in the world. Peter has not got enough money to buy a car, and the buses in *Expensive City* are pretty bad, so he uses the subway to go to work. Up to now, the subway was very cheap: you could travel anywhere with just one $2 ticket. Last month, the managers decided that it was too cheap so they invented the EFS (Expensive Fare System). With this system, users can only buy monthly tickets between adjacent stations, which allows them to move between these stations any number of times. The price of the monthly ticket varies between stations, so the decision of which tickets to buy must be taken carefully.



With the previous subway plan, the cheapest way to travel from Picadilly to Victoria and Queensway was to buy the monthly ticket Picadilly-Victoria and Queensway-Victoria, for a total cost of $12. Peter is a salesperson, so he needs to be able to travel to any part of the city. He wants to spend as little money as possible, and here is where you come into the picture. He has hired you to write a program that, given the list of stations, the fares of the monthly tickets between pairs of stations and the station nearest Peter's home, returns the minimum amount of money Peter has to spend in order to travel to any other station. This program also has to return value if it is not possible to go from Peter's home station to all the rest, because in this case Peter will begin to consider using buses.

Input

The input consists of several test cases. A test case begins with a line containing two integers: $1 \leq s \leq 400$ (the number of stations) and $0 \leq c \leq 79800$ (the number of connections) separated by a single

space. This is followed by s lines, each one containing the name of a subway station. These names will be strings of characters (uppercase or lowercase) without punctuation marks or whitespace characters, and with a maximum length of 10 characters. After the names of the stations there will be c lines showing the connections between stations. A connection allows people to travel from one station to the other in both directions. Each connection is represented as two strings indicating the names of the stations and a positive integer indicating the cost of the monthly ticket, all of which are separated by single spaces. All names of stations appearing in the connections will have previously appeared in the list of s stations. The connections will all be different, and there will not be any connection from a station to itself. The test case will end with a line containing the name of the station from which Peter needs to travel to all the other stations.
The input finishes with the phantom test case `„0 0"`, which must not be processed.

Output

For every test case, the output will be a line containing an integer, the minimum monthly price that Peter has pay to travel from the given station to all the others, or `Impossible` if it is not possible to travel to all the stations.

Sample Input

```
3 3 Picadilly
Victoria
Queensway
Picadilly Victoria 2
Queensway Victoria 10
Queensway Picadilly 20
Picadilly
4 2
Picadilly
Victoria
Queensway
Temple
Picadilly Victoria 2
Temple Queensway 100
Temple
0 0
```

Sample Output

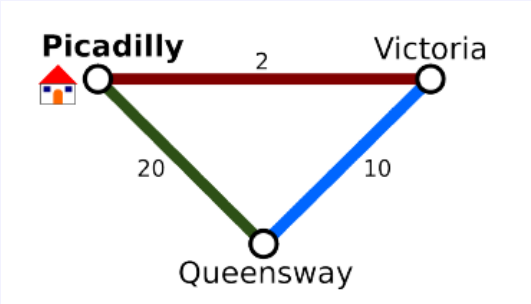
```
12
Impossible
```

Originální zadání úlohy je dostupné na adrese `https://uva.onlinejudge.org/external/117/11710.pdf` [cit. 2016/11/12]. Obrázek byl převzat z originálního zadání.

3.3.2 Česká verze zadání úlohy

11710 - Drahé metro

Petr bydlí v *Expensive City*, v jednom z nejdražších měst na světě. Petr nemá dostatek peněz, aby si koupil automobil, autobusy jsou v *Expensive City* poměrně špatné, a proto využívá služeb metra pro cestu do práce. Dosud bylo metro velmi levné - každý mohl cestovat kamkoliv se mu zachtělo s levnou jízdenkou za dva dolary. Minulý měsíc se manažeři rozhodli, že tato jízdenka je levná až příliš a vymysleli systém EFS (Drahý systém jízdného, Expensive Fare System). Uživatelé systému EFS si mohou zakoupit pouze jízdenky mezi sousedními stanicemi metra. Jízdenky dovolují uživatelům mezi příslušnými dvěma stanicemi cestovat bez omezení. Cena měsíční jízdenky se různí podle stanic, mezi kterými platí, proto Petr musí vhodnou jízdenku vybírat pečlivě.



V novém systému EFS je pro Petra nejlevnější si zakoupit jízdenky mezi stanicemi Picadilly-Victoria a Victoria-Queensway v celkové ceně 12 dolarů, ačkoliv existuje přímé spojení mezi stanicemi Picadilly a Queensway, jízdenka mezi těmito dvěma stanicemi stojí 20 dolarů.
Petr je obchodník a proto se potřebuje dostat do všech městských částí. Za jízdenky chce utratit co nejméně peněz a zde přichází řada na vás. Petr vás najal, abyste pro něj napsali program, který po zadání

seznamu stanic, cen jízdného mezi dvojicemi stanic a stanice nejblíže Petrova domova určí minimální cenu jízdenek tak, aby Petr utratil co nejméně peněz a zároveň mohl cestovat do všech stanic metra. Program musí v případě, kdy se nelze metrem dostat do všech stanic toto oznámit a Petr v tomto případě uváží cestování autobusem.

Vstup

Vstup se skládá z několika testovacích případů. Testovací případ začíná řádkou obsahující dvě celá čísla $1 \leq s \leq 400$ (počet stanic metra) a $0 \leq c \leq 79800$ (počet spojení) oddělená mezerou. Následuje s řádek, z nichž každá obsahuje název jedné ze stanic metra. Názvy stanic jsou řetězce (mohou obsahovat velká i malá písmena) bez interpunkce a mezer o maximální možné délce 10 znaků. Po části vstupu s názvy stanic je uvedeno c řádek s jednotlivými spoji mezi stanicemi. Každý umožňuje cestování pasažérů v obou směrech. Každé spojení je reprezentováno dvěma řetězci značícími stanice, mezi nimiž leží dané spojení, a cenou měsíční jízdenky, všechny údaje na řádce jsou odděleny mezerou. Všechna jména stanic, která se objeví v části popisu spojení, již byla uvedena v seznamu stanic. Žádné spojení nebude uvedeno dvakrát, a rovněž nebude uvedeno žádné spojení stanice se sebou samou. Testovací případ končí řádkou s názvem stanice, ze které Petr potřebuje cestovat do všech ostatních stanic.

Vstup končí dvěma nulami `„0 0"`, které nesmějí být zpracovány.

Výstup

Pro každý testovací případ bude výstupem řádka obsahující přirozené číslo značící minimální cenu všech jízdenek, které si Petr musí koupit, aby se mohl dopravit do libovolné stanice metra. Pokud by toto nebylo možné, bude výstupem `Impossible`.

Vzorový vstup

```
3 3
Picadilly
Victoria
Queensway
Picadilly Victoria 2
Queensway Victoria 10
Queensway Picadilly 20
Picadilly
4 2
Picadilly
Victoria
```

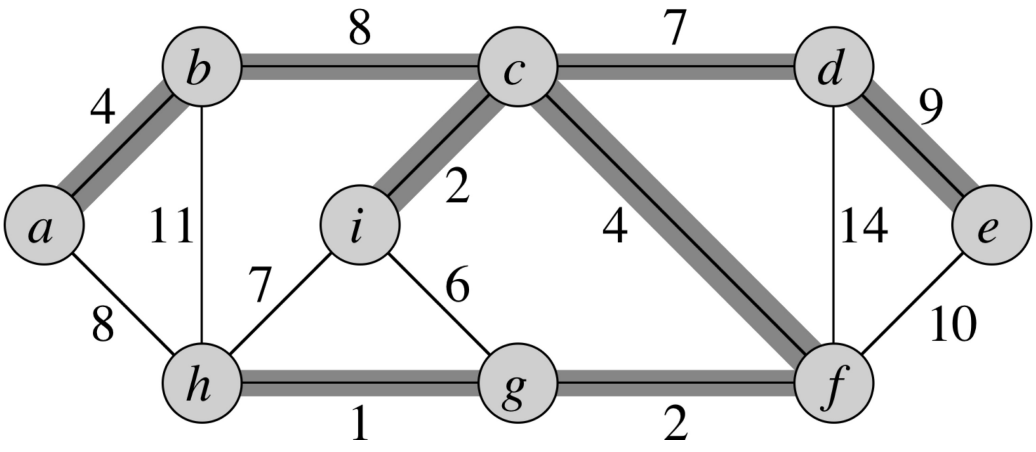
```
Queensway
Temple
Picadilly Victoria 2
Temple Queensway 100
Temple
0 0
Vzorový výstup
12
Impossible
```

Obrázek byl převzat z originálního zadání, které je dostupné na adrese `https://uva.onlinejudge.org/external/117/11710.pdf` [cit. 2016/11/12].

3.3.3 Analýza úlohy

Ze zadání úlohy je zřejmé, že se jedná o grafovou úlohu. Prvním krokem při řešení této úlohy je výběr vhodných datových struktur pro reprezentaci grafu. Struktury vhodné pro reprezentaci grafu byly popsány v kapitole 3.2.3.

Naším stěžejním úkolem je vybrat podgraf ze zadaného grafu tak, aby součet ohodnocení hran v podgrafu byl co nejmenší, a zároveň musí být v podgrafu obsaženy všechny vrcholy původního grafu. Takový podgraf nazýváme *minimální kostra grafu* (viz obrázek 3.8). Z obrázku jsou zřejmé vlastnosti minimální kostry grafu - neobsahuje cykly, obsahuje všechny vrcholy původního grafu a součet ohodnocení obsažených hran je nejmenší možný.



Obrázek 3.8: Minimální kostra grafu

Zdroj: `https://www.cs.indiana.edu/~achauhan/Teaching/B403/LectureNotes/images/09-mst-intro.jpg` [cit. 2016/11/12]

V minimální kostře grafu pak sečteme ohodnocení všech hran, čímž zjistíme minimální cenu jízdného, kterou musí Petr zaplatit za jízdenky, aby mohl cestovat mezi všemi stanicemi metra.

3.3.4 Algoritmy vhodné pro řešení úlohy

Stěžejní částí řešení úlohy je implementace algoritmu pro nalezení minimální kostry grafu. Ze zadání víme pouze, že zadaný graf bude mít kladné ohodnocení hran. Zadání nám negarantuje žádné další speciální vlastnosti grafu, a to včetně souvislosti (souvislým grafem rozumíme graf, v němž existuje cesta pro každou dvojici vrcholů v grafu). V případě nesouvislého grafu by výsledkem řešení úlohy bylo, že nelze určit minimální cenu jízdenek tak, aby byly pokryty všechny stanice metra.

Kruskalův algoritmus

Kruskalův algoritmus (viz algoritmus 3.7) lze použít pro nalezení minimální kostry grafu [19]. Algoritmus nejprve seřadí hrany grafu dle jejich ohodnocení. Následně tyto hrany zkouší přidat do kostry tak, aby po přidání hrany nevznikl v dosavadní kostře cyklus. Pokud cyklus po přidání hrany do kostry nevznikne, je přidaná hrana součástí minimální kostry grafu. V opačném případě, kdy by po přidání hrany do kostry vznikl cyklus a tím by došlo k porušení vlastností kostry, hrana není přidána. Při přidávání hrany nemusíme testovat přítomnost cyklu, postačí zjistit, zdali přidávaná hrana spojuje různé komponenty souvislosti [19].

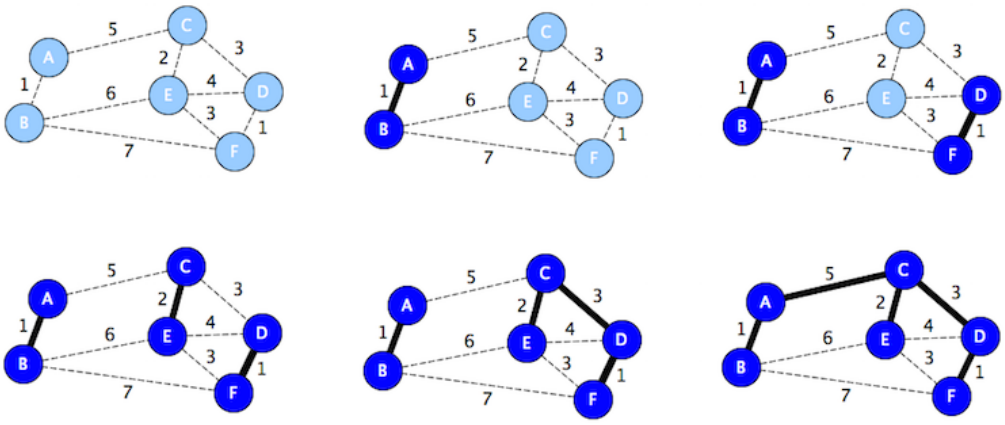
Průběh algoritmu je zobrazen na obrázku 3.9. Na počátku má algoritmus k dispozice hrany grafu, pro který hledáme minimální kostru. V prvním kroku algoritmus seřadí hrany podle jejich ohodnocení. Následuje postupné přidávání hran do kostry grafu. Nejprve algoritmus zkusí přidat hranu mezi vrcholy A a B. Protože tato hrana spojuje dvě komponenty souvislosti, je přidána do výsledné minimální kostry grafu. Analogicky algoritmus postupuje pro hrany mezi vrcholy D a F, C a E, C a D. Následně algoritmus zkusí do kostry vložit hranu mezi vrcholy E a F, ovšem tato hrana, podobně jako následující testovaná hrana mezi vrcholy D a E, nespojuje dvě komponenty souvislosti, a proto není tato hrana do výsledné minimální kostry přidána. Poslední testovanou hranou je hrana mezi vrcholy A a C, která je přidána do kostry. V tomto okamžiku je výsledná kostra grafu složena jen z jedné komponenty souvislosti a proto můžeme algoritmus ukončit.

Algoritmus 3.7: Kruskalův algoritmus [16]

Data: Graf G = (V, H, c), kde V jsou vrcholy, H hrany grafu, c představuje ohodnocení hrany; počáteční vrchol s

Výsledek: A obsahující hrany minimální kostry grafu G

```
   // Na počátku prázdná množina A obsahující hrany v minimální kostře
1  Množina A = ∅;
   // Pro každý vrchol vytvoříme samotnou komponentu souvislosti
2  foreach vrchol v ∈ V do
3  |  makeSet(v);
4  end
   // Seřadíme hrany vzestupně dle jejich ohodnocení
5  sort(H);
6  foreach hrana h ∈ H do
   |  // Vrcholy u a v jsou vrcholy na koncích hrany h
7  |  vrchol u = h.pocatek;
8  |  vrchol v = h.konec;
   |  // Pokud jsou vrcholy u a v v různých komponentách souvislosti
9  |  if FIND(u) ≠ FIND(v) then
   |  |  // Hranu h přidáme do fronty
10 |  |  A.add(h);
   |  |  // Sjednotíme komponenty, v nichž ležely vrcholy u a v
11 |  |  UNION(u, v);
12 |  end
13 end
```



Obrázek 3.9: Hledání minimální kostry grafu pomocí Kruskalova algoritmu **Zdroj:** `http://cs.lmu.edu/~ray/images/kruskal.png` [cit. 2016/11/13]

V průběhu algoritmu testujeme hrany, zdali s nimi incidentní vrcholy patří do jedné či dvou komponent souvislosti. Tento problém se nazývá Union&Find problém. Union&Find problém je realizován dvěma operacemi, kterými jsou `UNION`, která sjednocuje dvě komponenty souvislosti, a `FIND`, která vrátí reprezentanta komponenty souvislosti pro daný vrchol grafu. Reprezentanta vybereme pro každou komponentu souvislosti pouze jednoho, tohoto reprezentanta sdílejí všechny vrcholy dané komponenty souvislosti [19].

Problém Union&Find může být řešen několika způsoby. Nejjednodušším způsobem je řešení pomocí pole. Nejdříve očíslujeme vrcholy grafu čísly $1, \ldots, n$. V poli $R[1 \ldots n]$ budeme mít uloženo číslo reprezentanta komponenty souvislosti pro každý vrchol grafu. Operace `FIND` pouze vrátí pro vrchol i hodnotu $R[i]$, z čehož vyplývá, že její časová asymptotická složitost je rovna $\mathcal{O}(1)$ [19]. Operace `UNION` nejdříve najde pro dva zadané vrcholy i a j reprezentanty $a = R[i]$ a $b = R[j]$. Pokud nejsou získaní reprezentanti a, b shodní, operace `UNION` v poli přepíše všechny výskyty reprezentanta a na reprezentanta b, což značí sjednocení daných komponent souvislostí. Asymptotická časová složitost operace `UNION` činí $\mathcal{O}(n)$ [19].

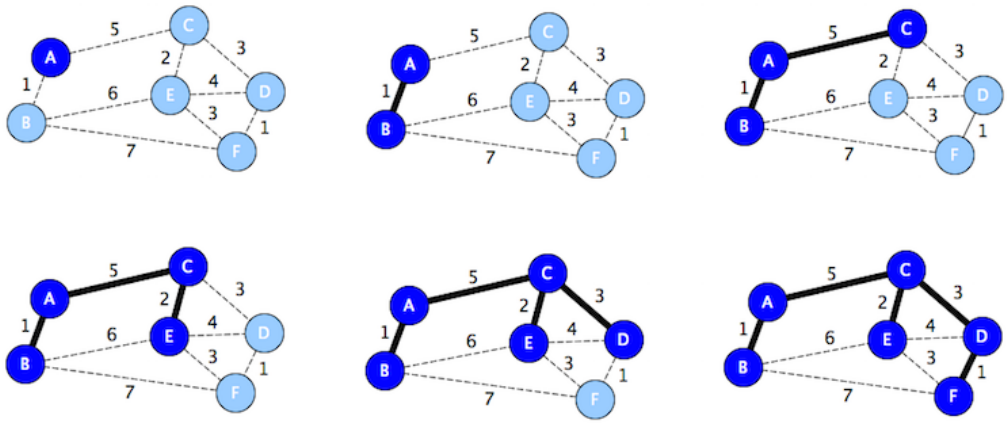
Problém Union&Find může být implementován i dalšími způsoby, kdy můžeme získat lepší časovou složitost operace `UNION`, ovšem za cenu zhoršení složitosti operace `FIND`. Asymptotická časová složitost obou operací pak v případě použití implementace nazývané *union by rank* činí $\mathcal{O}(\log n)$ [19].

Časová asymptotická složitost Kruskalova algoritmu činí $\mathcal{O}(m \log n)$ při implementace problému Union&Find nazývané *union by rank* [19].

Prim-Jarnikův algoritmus

Prim-Jarnikův algoritmus (viz algoritmus 3.8) si při svém běhu udržuje pouze jednu komponentu souvislosti, kterou postupně rozšiřuje. Do vytvářené komponenty, která na konci algoritmu odpovídá minimální kostře grafu, jsou postupně přidávány hrany s nejmenším ohodnocením, které vedou z této komponenty do jiné [19]. Tímto se konstruovaná minimální kostra postupně „rozrůstá“ do doby, kdy jsou v konstruované minimální kostře obsaženy všechny vrcholy zadaného grafu, pro který hledáme minimální kostru (viz obrázek 3.10).

V průběhu algoritmu si pamatujeme pro každý vrchol v, který zatím není obsažen v množině hran T tvořících minimální kostru, ohodnocení hrany $d[v]$ s nejmenším ohodnocením a odkud tato hrana vede $odkud[v]$. Hrana, kterou si pamatujeme, musí vést „ven“ z komponenty souvislosti, tedy musí být incidentní s některým z vrcholů tvořících konstruovanou minimální kostru grafu. Nyní postačí projít hodnoty $d[v]$ všech vrcholů grafu, které dosud nejsou součástí konstruované minimální kostry a z těchto hodnot vybrat nejnižší hodnotu a k ní příslušnou hranu, kterou přidáme do konstruované minimální kostry grafu. Po každém přidání vrcholu do konstruované minimální kostry je nutné zkontrolovat u všech vrcholů, které dosud nejsou obsaženy v minimální kostře grafu, zdali si pamatujeme správnou hranu, tedy hranu s nejnižším ohodnocením vedoucí do některého z vrcholů, které jsou dosud obsaženy v konstruované kostře grafu [19].



Obrázek 3.10: Hledání minimální kostry grafu Prim-Jarnikovým algoritmem

Zdroj: `http://cs.lmu.edu/~ray/images/prim.png` [cit. 2016/11/13]

Algoritmus 3.8: Prim-Jarnikův algoritmus [19]

Data: Graf G = (V, H), kde V jsou vrcholy, H hrany grafu
Výsledek: Množina A obsahující hrany minimální kostry grafu G

```
   // Na počátku prázdná množina A obsahující hrany v minimální kostře
1  Množina A = ∅;
   // Zvolíme startovní bod r a nastavíme pole d[v] obsahující
      ohodnocení hran vedoucích „ven" z komponenty souvislosti pro
      všechny vrcholy
2  d[r] = 0;
3  odkud[r] = NULL;
4  foreach Vrchol v ∈ V \{r} do
5  |  d[v] = ∞;
6  |  odkud[v] = NULL;
7  end
   // Inicializujeme prioritní frontu vrcholů grafu
8  Fronta Q = v ∈ V s prioritami d[v];
   // Hrany postupně přidáváme do kostry
9  while Q není prázdná do
10 |  Vrchol v = Q.pop();
11 |  A.add(hrana(v, odkud[v]));
12 end
   // Zkontrolujeme správnost d[v] a odkud[v]
13 foreach Hrana h ∈ H do
   |  // Vrcholy u a v jsou vrcholy na koncích hrany h
14 |  Vrchol u = h.pocatek;
15 |  Vrchol v = h.konec;
16 |  if d[v] > h.ohodnoceni then
17 |  |  d[v] = h.ohodnoceni;
18 |  |  odkud[v] = u;
19 |  end
20 end
```

Asymptotická časová složitost Prim-Jarnikova algoritmu činí v případě realizace prioritní fronty polem $\mathcal{O}(n^2 + m)$, při realizaci prioritní fronty binární haldou pak časová složitost činí $\mathcal{O}(n + m) \log n$.

Borůvkův algoritmus

Borůvkův algoritmus (viz algoritmus 3.9) je dalším zástupcem algoritmů pro hledání minimální kostry grafu. Borůvkův algoritmus vyžaduje, aby minimální kostra grafu byla určena jednoznačně [19]. Jednoznačné určení minimální kostry v grafu je zaručeno v grafech, kde mají všechny hrany vzájemně různá ohodnocení [19]. Borůvkův algoritmus lze dobře paralelizovat [19].

Algoritmus 3.9: Borůvkův algoritmus [19]

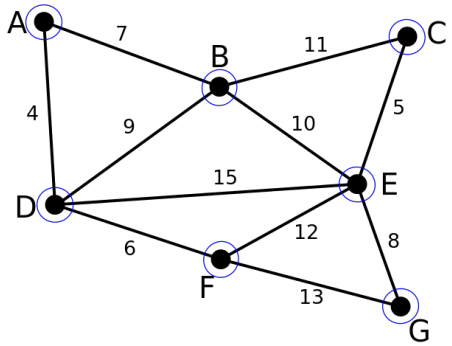
Data: Graf G = (V, H), kde V jsou vrcholy, H hrany grafu
Výsledek: Graf T představující minimální kostru grafu G

```
1  Graf T = (V, ∅);
2  while graf T není souvislý do
       // Pomocný graf G_pom
3      Graf GPom = (reprezentanti komponent souvislosti T, ∅);
4      foreach hrana h ∈ H do
           /* u je reprezentant komponenty, v níž leží vrchol na začátku
              hrany; podobně v                                          */
5          Vrchol u = reprezentant(h.pocatek);
6          Vrchol v = reprezentant(h.konec);
7          GPom.pridejHranu(u, v);
           // Odstraníme smyčky v grafu G_pom
8          GPom.odstranSmycky();
           /* Odstraníme násobné hrany, ponecháme hranu s nejnižším
              ohodnocením, kterou přidáme do kostry grafu T             */
9          foreach dvojice vrcholů [uv] ∈ GPom do
10             Mnozina hrany = najdiVsechnyHrany([uv]);
11             Hrana min = urciNejlevnejsiHranu(hrany);
12             T.pridejHranu(min);
13         end
14     end
15 end
```

Na počátku běhu algoritmu máme k dispozici graf obsahující vrcholy a ohodnocené hrany (viz obrázek 3.11). Jak již bylo řečeno, pro správnou

funkci algoritmu je nutné, aby minimální kostra grafu byla určena jednoznačně, tedy aby neexistovali dvě různé minimální kostry se shodným součtem ohodnocení hran obsažených v daných minimálních kostrách. Pokud není zaručena jednoznačnost kostry, lze použít Kruskalův nebo Prim-Jarnikův algoritmus, které byly popsány výše.

Algoritmus v první iteraci (viz obrázek 3.12) vybere pro každý vrchol hranu s nejnižším ohodnocením, která je s tímto vrcholem incidentní. Pro vrcholy A a D jde o hranu AD, pro vrchol B byla vybrána hrana AB, pro vrcholy C a E hrana CE, pro vrchol F hrana DF a pro vrchol G hrana EG. Tímto v grafu vznikly dvě komponenty souvislosti, z nichž první je tvořena vrcholy $\{A, B, D, F\}$, druhá komponenta souvislosti je pak tvořena vrcholy $\{C, E, G\}$. Ve druhé iteraci algoritmu (viz obrázek 3.13) je pak vybrána hrana komponenty $\{A, B, D, F\}$ s nejnižším ohodnocením, která vede ven z této komponenty. Protože získaná minimální kostra nyní pokrývá všechny vrcholy grafu, můžeme algoritmus ukončit.

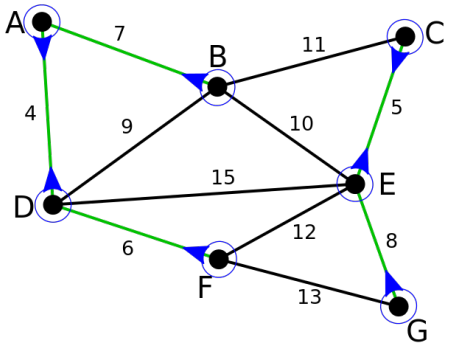


Obrázek 3.11: Počáteční stav při běhu Borůvkova algoritmu
Zdroj:
`https://en.wikipedia.org/wiki/Borůvka's_algorithm#/media/File:Borůvka_Algorithm_1.svg` [cit. 2016/11/14]

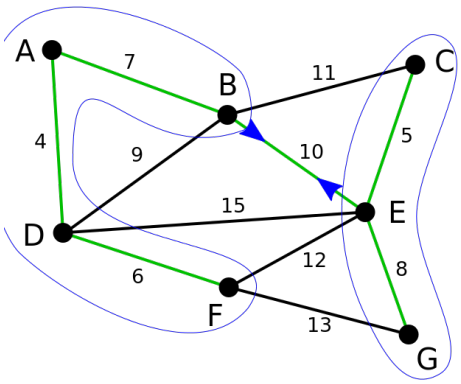
Při implementaci pro každou komponentu souvislosti zvolíme reprezentanta této komponenty a pro každou komponentu souvislosti hledáme hranu s nejnižším ohodnocením, která vede z této komponenty „ven“. V každé iteraci si vytvoříme pomocný graf G_{pom}, jehož vrcholy jsou jednotlivé komponenty souvislosti. Do pomocného grafu přidáváme všechny hrany z původního grafu (vrcholy původního grafu jsou nahrazeny reprezentanty komponent souvislosti). Při přidávání hran do pomocného grafu mohou vznikat smyčky, které rovnou zahazujeme, a násobné hrany mezi vrcholy pomocného grafu, z nichž ponecháme jen tu s nejnižším ohodnocením. Před další iterací Borůvkova algoritmu musíme sjednotit nyní propojené komponenty

souvislosti. Protože sjednocujeme více komponent souvislosti, není vhodné postupně sjednocovat jednotlivé dvojice komponent souvislosti, neboť by sjednocování tímto způsobem mohlo trvat příliš dlouho [19]. Je vhodnější sjednotit komponenty souvislosti tím, že pomocí prohledávání do hloubky (Depth-first search, DFS) projdeme všechny vrcholy ve sjednocovaných komponentách souvislosti (které jsou již nyní propojené) a všem nalezeným vrcholům nastavíme stejného reprezentanta.

Asymptotická časová složitost Borůvkova algoritmu činí $\mathcal{O}(m \log n)$, kde m značí počet hran a n počet vrcholů grafu.



Obrázek 3.12: První iterace běhu Borůvkova algoritmu
Zdroj:
`https://en.wikipedia.org/wiki/Borůvka's_algorithm#/media/File:Borůvka_Algorithm_2.svg` [cit. 2016/11/14]



Obrázek 3.13: Druhá (poslední) iterace běhu Borůvkova algoritmu
Zdroj:
`https://en.wikipedia.org/wiki/Borůvka's_algorithm#/media/File:Borůvka_Algorithm_3.svg` [cit. 2016/11/14]

3.3.5 Řešení úlohy

Při řešení úlohy byl použit Kruskalův algoritmus vzhledem k příznivé hodnotě asymptotické časové složitosti, která činí $\mathcal{O}(m \log n)$. Při implementaci této úlohy není nutné ukládat celý graf, postačí si jen uložit hrany. Namísto názvů vrcholů si v hranách pamatujeme jejich číselná označení. Čísla vrcholů pak můžeme přímo používat v části implementace Union&Find, aniž bychom si museli pamatovat názvy vrcholů grafu.

Při implementaci problému Union&Find bylo použito mírně vylepšené řešení, kdy si u operace `UNION` pamatujeme ještě vrcholy ležící v dané komponentě a jejich počet. Poté přepisujeme reprezentanty jen u vrcholu ležící v komponentě s menším počtem vrcholů. Tímto vylepšením získáme amortizovanou asymptotickou časovou složitost operace `UNION` rovnu $\mathcal{O}(\log n)$ [19].

3.3.6 Výsledek validace

Úloha byla úspěšně zvalidována, čas běhu činil 1,880 sekundy (viz obrázek 3.14).

#	Problem	Verdict	Language	Run Time	Submission Date
19209634	11710 Expensive subway	Accepted	JAVA	1.880	2017-04-20 13:34:04

Obrázek 3.14: Zpráva o validaci úlohy Expensive Subway

3.4 Where's Waldorf?

3.4.1 Originální zadání úlohy

10010 - Where's Waldorf?

Given a m by n grid of letters, $(1 \leq m, n \leq 50)$, and a list of words, find the location in the grid at which the word can be found.
A word matches a straight, uninterrupted line of letters in the grid. A word can match the letters in the grid regardless of case (i.e. upper and lower case letters are to be treated as the same). The matching can be done in any of the eight directions either horizontally, vertically or diagonally through the grid.

Input

The input begins with a single positive integer on a line by itself indicating the number of the cases following, each of them as described below. This line is followed by a blank line, and there is also a blank line between two consecutive inputs.

The input begins with a pair of integers, m followed by n, $1 \leq m, n \leq 50$ in decimal notation on a single line. The next m lines contain n letters each; this is the grid of letters in which the words of the list must be found. The letters in the grid may be in upper or lower case. Following the grid of letters, another integer k appears on a line by itself $(1 \leq k \leq 20)$. The next k lines of input contain the list of words to search for, one word per line. These words may contain upper and lower case letters only (no spaces, hyphens or other non-alphabetic characters).

Output

For each test case, the output must follow the description below. The outputs of two consecutive cases will be separated by a blank line.

For each word in the word list, a pair of integers representing the location of the corresponding word in the grid must be output. The integers must be separated by a single space. The first integer is the line in the grid where the first letter of the given word can be found (1 represents the topmost line in the grid, and m represents the bottom-most line). The second integer is the column in the grid where the first letter of the given word can be found (1 represents the leftmost column in the grid, and n represents the rightmost column in the grid). If a

word can be found more than once in the grid, then the location which is output should correspond to the uppermost occurence of the word (i.e. the occurence which places the first letter of the word closest to the top of the grid). If two or more words are uppermost, the output should correspond to the leftmost of these occurences. All words can be found at least once in the grid.

Sample Input

```
1

8 11
abcDEFGhigg
hEbkWalDork
FtyAwaldORm
FtsimrLqsrc
byoArBeDeyv
Klcbqwikomk
strEBGadhrb
yUiqlxcnBjf
4
Waldorf
Bambi
Betty
Dagbert
```

Sample Output

```
2 5
2 3
1 2
7 8
```

Originální zadání úlohy je dostupné na adrese `https://uva.onlinejudge.org/external/100/10010.pdf` [cit. 2017/03/05].

3.4.2 Česká verze zadání úlohy

10010 - Kde je Waldorf?

Vaším úkolem je najít v zadané mřížce písmen o rozměrech $m \times n$ pozice, na kterých lze najít hledaná slova.
Slova musí být ve mřížce nalezena jako nepřerušená posloupnost písmen s dodrženým pořadím písmen, u jednotlivých písmen nezáleží na

jejich velikosti (jednotlivá písmena slov tedy mohou být malá i velká). Hledaná slova se v mřížce mohou nacházet v horizontálně, vertikálně i diagonálně ve všech osmi směrech.

Vstup

Vstupní data začínají celým kladným číslem na jedné řádce, které udává celkový počet testovaných případů. Tato řádka je následována prázdnou řádkou. Dále se prázdná řádka nachází mezi dvěma po sobě jdoucími případy.

Každý testovaný případ začíná dvojicí celých čísel uvedených v desítkové soustavě, z nichž první je číslo m následované číslem n ($1 \leq m, n \leq 50$), na samostatné řádce. Každá z následujících m řádek obsahuje n písmen, která představují jednotlivé řádky mřížky písmen, v nichž budou hledána zadaná slova. Písmena mřížky mohou být velká i malá. Po řádkách, které popisují mřížku písmen, následuje řádka obsahující pouze jedno celé číslo k ($1 \leq k \leq 20$), které udává počet řádek s hledanými slovy, která následují na dalších k řádkách. Hledaná slova mohou obsahovat jen malá a velká písmena, nebudou obsahovat mezery, pomlčky a jiné nealfabetické znaky.

Výstup

Pro každý testovací případ musí výstup přesně odpovídat níže uvedené specifikaci výstupu. Výstupní data dvou po sobě jdoucích testovacích případů budou oddělena prázdnou řádkou.

Pro každé hledané slovo vypište dvojici celých čísel oddělených mezerou, která představuje pozici hledaného slova v mřížce písmen. První z čísel udává řádku mřížky, ve které se nachází první písmeno hledaného slova (první horní řádka mřížky je označena číslem 1, číslo m naopak označuje spodní řádku mřížky). Druhé číslo udává sloupec mřížky, ve kterém se nachází první písmeno hledaného slova (sloupec mřížky nacházející se nejvíce vlevo je označen číslem 1, číslo n naopak označuje sloupec mřížky nejvíce vpravo).

Pokud se slovo nachází v mřížce vícekrát, je očekávaným výsledkem výskyt slova v co nejvyšším řádku mřížky. Pokud se první písmeno slova nachází vícekrát ve stejném řádku, potom je požadovaným výsledkem výskyt slova ve sloupci co nejvíce vlevo mřížky. Všechna slova se v mřížce nacházejí alespoň jednou.

Vzorový vstup

```
1
```

```
8 11
abcDEFGhigg
hEbkWalDork
FtyAwaldORm
FtsimrLqsrc
byoArBeDeyv
Klcbqwikomk
strEBGadhrb
yUiqlxcnBjf
4
Waldorf
Bambi
Betty
Dagbert
```
Vzorový výstup
```
2 5
2 3
1 2
7 8
```

3.4.3 Analýza úlohy

Ze zadání je zřejmé, že úloha je jednou ze známých luštitelských her, v tomto případě jde o klasickou osmisměrku. Stěžejním řešeným úkolem v rámci úlohy je vyhledávání podřetězce v řetězci. Vstupní data úlohy mají formu dvourozměrného pole znaků, v němž hledáme zadané řetězce v osmi směrech (horizontálně zleva doprava i zprava doleva, vertikálně shora dolů i zdola nahoru, diagonálně pak ve čtyřech směrech - doleva a nahoru, doleva a dolů, doprava a nahoru, doprava a dolů).

Ve chvíli, kdy nalezneme hledané slovo, vypíšeme informaci o pozici, na které se dané slovo nachází. Zde je třeba dát pozor na správné číslování řádků i sloupců, které dle zadání úlohy začíná číslem jedna.

3.4.4 Algoritmy vhodné pro řešení úlohy

Známé algoritmy pro vyhledávání podřetězců v řetězci jsou určené pro vyhledávání podřetězců v řetězci (tedy v jednorozměrném poli znaků), ovšem v případě této úlohy je nutné podřetězce hledat v dvourozměrném poli znaků a ve více směrech.

V případě této úlohy lze snadno získat řetězce jakožto jednorozměrná pole znaků, v nichž budeme hledat hledané podřetězce, kdy postačí hledat slova nejprve v jednotlivých řádcích, poté v jednotlivých sloupcích a nakonec v řetězcích uložených v mřížce diagonálně. Abychom hledali podřetězce ve všech osmi směrech, je nutné, v případě, kdy jsme dosud nenalezli hledaný podřetězec, provést reverzi řetězců, v nichž zadané podřetězce hledáme, tedy např. v případě řádek jednou slovo hledáme zleva doprava a po reverzi jej hledáme zprava doleva, analogicky pak postupujeme i pro sloupce a diagonály.

Triviální algoritmus

Triviální algoritmus (viz algoritmus 3.10), jak již jeho název napovídá, je nejjednodušším algoritmem pro vyhledávání podřetězců v řetězci.

Algoritmus 3.10: Triviální algoritmus [15]

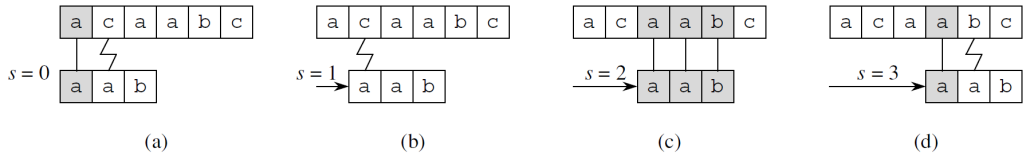
Data: Prohledávaný řetězec T, hledaný podřetězec P
Výsledek: Pozice x, na které začíná v prohledávaném řetězci T hledaný podřetězec P

```
1 n = T.délka;
2 m = P.délka;
  // Hledáme na každé pozici v prohledávaném řetězci
3 s = 0;
4 for s ≤ n − m do
5 |   if P[0...m] == T[s...(s + m − 1)] then
  |   |   // Podřetězec nalezen na pozici s
6 |   |   print Podřetězec se v řetězci nachází na pozici s;
7 |   end
8 |   s = s + 1;
9 end
```

Princip algoritmu je velmi jednoduchý (viz obrázek 3.15). Algoritmus pro každou pozici prohledávaného řetězce testuje, zdali se na této pozici nevyskytuje hledaný podřetězec [15].

Například pro prohledávaný řetězec `acaabc`, v němž hledáme podřetězec `aab` (viz obrázek 3.15), nejprve otestujeme, zdali se hledaný podřetězec nenachází hned na první pozici ($s = 0$) prohledávaného řetězce. Protože na první pozici se hledaný podřetězec nenachází, přesuneme se o jednu pozici doprava, kde se hledaný podřetězec opět nenachází. V další iteraci algoritmus se opět posuneme o jednu pozici doprava (tedy na třetí pozici), kde se již hledaný

podřetězec nachází. Dle našich požadavků můžeme v algoritmu pokračovat dále, abychom našli všechna umístění hledaného podřetězce v řetězci, nebo v případě, kdy je naším cílem nalezení jen jednoho výskytu podřetězce v řetězci, můžeme algoritmus ukončit.



Obrázek 3.15: Průběh triviálního algoritmu pro hledání podřetězců v řetězci pro podřetězec `aab` hledaný v řetězci `acaabc`
Zdroj: převzato z [15].

Asymptotická časová složitost triviálního algoritmu pro vyhledávání podřetězců v řetězci činí $\mathcal{O}((n-m+1)m)$, kde n je délka prohledávaného řetězce a m je délka hledaného podřetězce [15].

Rabin-Karpův algoritmus

Rabin-Karpův algoritmus (viz algoritmus 3.11) slouží pro vyhledávání podřetězců v zadaném řetězci a je vhodný především pro případy, kdy hledáme více podřetězců v jednom řetězci. Algoritmus pracuje ve dvou krocích, prvním z nich je předzpracování, které probíhá s asymptotickou časovou složitostí $\mathcal{O}(m)$ [15], kde m označuje délku hledaného podřetězce. Druhou částí pak je vlastní vyhledávání podřetězce, jehož asymptotická časová složitost v nejhorším případě činí $\mathcal{O}((n-m+1)m)$, kde n je délka prohledávaného řetězce [15], tedy asymptotická časová složitost je v nejhorším případě shodná s asymptotickou časovou složitostí triviálního algoritmu, který byl popsán výše v kapitole 3.4.4. Asymptotická časová složitost Rabin-Karpova algoritmu v průměrném případě dosahuje příznivějších hodnot [15].

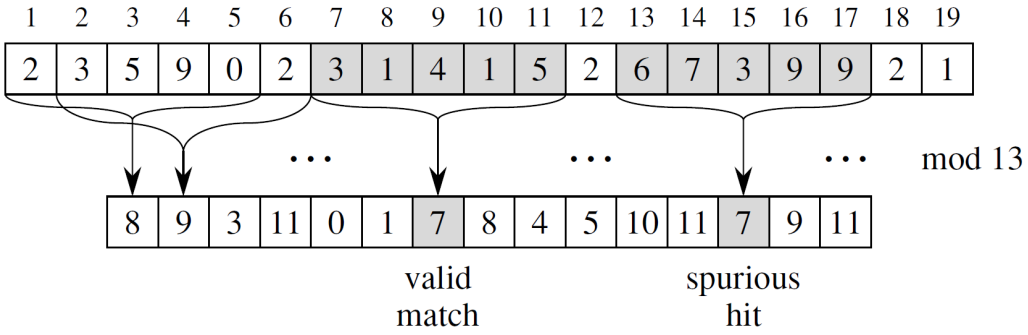
Algoritmus pracuje obdobně jako triviální algoritmus, ovšem namísto porovnávání znaků hledaného podřetězce a části prohledávaného řetězce porovnává otisky (hashe) hledaného podřetězce a prohledávaného řetězce. Pokud se otisky části prohledávaného řetězce a hledaného podřetězce shodují, neznamená to ještě, že jsme nalezli hledaný podřetězec, neboť dva rozdílné řetězce mohou mít shodný otisk. Proto je při shodě otisků nutné porovnat jednotlivé znaky podřetězce a části řetězce, na které testujeme výskyt hledaného podřetězce.

Algoritmus 3.11: Rabin-Karpův algoritmus (převzato z [15])

Data: Prohledávaný řetězec T, hledaný podřetězec P, základ d (základ se obvykle volí jako počet znaků abecedy hledaného podřetězce a prohledávaného řetězce), prvočíslo q pro rozptylovou (hashovací) funkci

Výsledek: Pozice x, na které začíná v prohledávaném řetězci T hledaný podřetězec P

```
1  n = T.délka;
2  m = P.délka;
3  h = d^(m-1) mod q;
4  p = 0;
5  t_0 = 0;
   // Předzpracování dat - výpočet otisků pro hledané podřetězce
6  i = 0;
7  for i < m do
8  |  p = (dp + P[i]) mod q);
9  |  t_0 = (dt_0 + P[i]) mod q);
10 end
   // Hledáme na každé pozici v prohledávaném řetězci
11 s = 0;
12 for s < n − m do
13 |  if p == t_s then
14 |  |  if P[0...m] == T[s...(s + m − 1)] then
   |  |  |  // Podřetězec nalezen na pozici s
15 |  |  |  print Podřetězec se v řetězci nachází na pozici s
16 |  |  end
17 |  end
18 |  ;
19 |  if s < n - m then
   |  |  /* Přepočítáme otisk pro další testovaný úsek prohledávaného
   |  |     řetězce                                                  */
20 |  |  t_{s+1} = (d(t_s − T[s]h) + T[s + m]) mod q;
21 |  end
22 |  s = s + 1;
23 end
```



Obrázek 3.16: Průběh Rabin-Karpova algoritmu při vyhledávání řetězce `31415` v řetězci `2359023141526739921`
Zdroj: převzato z [15].

Princip algoritmu lze demonstrovat na obrázku 3.16, kde v dolní části obrázku jsou znázorněny počítané otisky pro jednotlivé části prohledávaného řetězce. V horní části je pak uveden prohledávaný řetězec `235902314152673-9921`. Hledaným podřetězcem je v tomto konkrétním případě `31415`. Algoritmus postupuje od první pozice prohledávaného řetězce a zkontroluje shodu na otisk pěti následujících znaků s otiskem hledaného podřetězce. Pokud se otisky neshodují, algoritmus postupuje o jednu pozici vpravo a opět zkontroluje shodu otisku části prohledávaného řetězce a hledaného podřetězce. Pokud se otisky shodují, algoritmus zkontroluje shodu části prohledávaného řetězce s hledaným podřetězcem znak po znaku stejně jako triviální algoritmus. Pokud se testovaná část prohledávaného řetězce neshoduje s hledaným podřetězcem, algoritmus postoupí, stejně jako v případě neshody otisků, o jednu pozici vpravo. V opačném případě algoritmus nalezl hledaný podřetězec.

Knuth-Morris-Prattův algoritmus

Knuth-Morris-Prattův algoritmus (viz algoritmus 3.12) vylepšuje triviální algoritmus (viz kapitola 3.4.4) tím, že v případě částečné nalezené shody prohledávaného řetězce a hledaného podřetězce se neposune pouze o jeden znak doprava, ale posune se o více znaků, které nemohou být začátkem hledaného podřetězce. Průběh Knuth-Morris-Prattova algoritmu je rozdělen do dvou částí - předzpracování dat a vlastní vyhledávání hledaného podřetězce.

V rámci předzpracování dat (viz algoritmus 3.13) algoritmus pro hledaný podřetězec vytvoří tzv. chybovou funkci (anglicky „failure function“), která se používá pro určení počtu znaků, o které se může algoritmus v průběhu vyhledávání hledaného podřetězce v zadaném řetězci posunout, aniž by zbytečně kontroloval pozice, na nichž se hledaný podřetězec nemůže nacházet.

Časová asymptotická složitost předzpracování dat činí $\mathcal{O}(m)$, vlastní vyhledávání podřetězce v zadaném řetězci pak dosahuje asymptotické složitosti $\mathcal{O}(n)$, kde n značí délku prohledávaného řetězce a m pak značí délku hledaného podřetězce [15]. Celková asymptotická složitost Knuth-Morris-Prattova algoritmu činí $\mathcal{O}(m+n)$.

Algoritmus 3.12: Knuth-Morris-Prattův algoritmus (převzato z [15])

Data: Prohledávaný řetězec T, hledaný podřetězec P
Výsledek: Pozice x, na které začíná v prohledávaném řetězci T hledaný podřetězec P

```
1  n = T.délka;
2  m = P.délka;
3  π = VYPOČTI-CHYBOVOU-FUNKCI(P);
4  q = 0; // počet shodných znaků
5  i = 0;
6  for i < n do
7  |  while q > 0 ∧ P[q + 1] ≠ T[i] do
8  |  |  q = π[q]; // další znak se neshoduje
9  |  end
10 |  if P[q + 1] == T[i] then
11 |  |  q = q + 1 // další znak se shoduje
12 |  end
13 |  if q == m then
   |  |  // Už jsme nalezli všechny znaky
14 |  |  print Podřetězec se v řetězci nachází na pozici i - m;
15 |  end
16 |  i = i + 1;
17 end
```

3.4.5 Řešení úlohy

Pro řešení úlohy byl použit triviální algoritmus vzhledem ke své snadné implementaci, kdy lze snadno procházet znaky do všech osmi směrů ze zadané pozice. Dalším důvodem pro výběr triviálního algoritmu byl malý rozsah vstupních dat, kdy triviální algoritmus je dostatečně rychlý pro vyřešení úlohy v zadaném časovém intervalu.

Algoritmus 3.13: Knuth-Morris-Prattův algoritmus - chybová funkce (převzato z [15])

Data: Hledaný podřetězec P
Výsledek: Hodnoty chybové funkce π

```
1  m = P.délka;
2  π[1...m];
3  π[1] = 0;
4  q = 2;
5  k  = 0;
6  for q ≤ m do
7  |  while k  > 0 ∧ P[k + 1] ≠ P[q] do
8  |  |  k  = π[k];
9  |  end
10 |  if P[k + 1] == P[q] then
11 |  |  k  = k  + 1;
12 |  end
13 |  π[q] = k ;
14 |  i = i + 1;
15 end
16 return π;
```

3.4.6 Výsledek validace

Úloha byla úspěšně zvalidována, čas běhu činil 0,100 sekundy (viz obrázek 3.17).

#	Problem	Verdict	Language	Run Time	Submission Date
19209658	10010 Where's Waldorf?	Accepted	JAVA	0.100	2017-04-20 13:36:25

Obrázek 3.17: Zpráva o validaci úlohy Where's Waldorf?

4 Závěr

První část práce představuje a popisuje některé známé online systémy pro nácvik programování, konkrétně je v první části práce popsáno celkem 7 systémů, kterými jsou systémy CodeChef, CodinGame, HackerRank, Sphere Online Judge, TopCoder, UVa Online Judge a Timus Online Judge. Popis systémů se zaměřuje především na zaměření těchto systémů a způsob práce s nimi, počet podporovaných programovacích jazyků a počet úloh dostupných uživateli pro nácvik programování.

Dalším cílem bakalářské práce byl výběr sady úloh pro výuku programování na počátku bakalářského studia na Fakultě aplikovaných věd Západočeské univerzity v Plzni. V druhé části práce je vybraná sada úloh popsána. Úlohy byly vybrány na serveru UVa Online Judge vzhledem k vysokému počtu dostupných úloh a snadnému použití daného systému. Vzhledem k tomu, že je systém UVa Online Judge využíván v několika předmětech vyučovaných na Fakultě aplikovaných věd, lze očekávat, že alespoň někteří studenti již budou mít zkušenosti s tímto systémem.

Sada vybraných úloh obsahuje 20 úloh, které jsou různě obtížné. Úlohy byly vybrány tak, aby pokryly hlavní témata vyučovaná na počátku bakalářského studia. Většina úloh proto představuje grafové úlohy, v nichž je úkolem například najít nejkratší cestu mezi dvěma vrcholy nebo zkonstruovat minimální kostru grafu. Zastoupeny však jsou i jednodušší úlohy vhodné pro začátky programování.

V praktické části bakalářské práce jsou jednotlivé úlohy analyzovány. Ve vlastním textu bakalářské práce jsou analyzovány čtyři úlohy - Joseph's Cousin, All Roads Lead Where?, Expensive Subway a Where's Waldorf?; analýza dvou úloh (The Tourist Guide a Shortest Names) je součástí příloh práce, analýza zbývajících úloh byla provedena stejným způsobem jako analýza úloh The Tourist Guide a Shortest Names. Analýza zbývajících úloh je uložena na přiloženém CD. U každé úlohy je uvedeno originální zadání v angličtině, české přeložené zadání, analýza vhodných algoritmů pro řešení dané úlohy a řešení úlohy včetně zdrojových kódů programového řešení v programovacím jazyce Java. Programová řešení všech úloh byla úspěšně zvalidována na serveru UVa Online Judge.

Bakalářská práce může být v budoucnu rozšířena o popis dalších zajímavých úloh vhodných pro výuku programování nebo dalších systémů pro nácvik programování.

Literatura

[1] *Programming Competition,Programming Contest,Online Computer Programming* [online]. Directi Group, 2009. [cit. 2016/10/17]. Dostupné z: www.codechef.com/.

[2] *FAQ | CodeChef* [online]. Directi Group, 2009. [cit. 2016/10/17]. Dostupné z: www.codechef.com/wiki/faq.

[3] *Terms and conditions - CodinGame* [online]. CodinGame, 2012. [cit. 2016/10/19]. Dostupné z: www.codingame.com/home.

[4] *about us | technical recruiting | hiring the best engineers* [online]. HackerRank, 2016. [cit. 2016/10/20]. Dostupné z: www.hackerrank.com/aboutus.

[5] *Scoring | HackerRank* [online]. HackerRank, 2016. [cit. 2016/10/20]. Dostupné z: www.hackerrank.com/scoring.

[6] *Sphere Online Judge (SPOJ) - Info* [online]. Sphere Research Labs, 2016. [cit. 2016/10/20]. Dostupné z: www.spoj.com/info/.

[7] *Sphere Online Judge (SPOJ) - Contests* [online]. Sphere Research Labs, 2016. [cit. 2016/10/20]. Dostupné z: www.spoj.com/contests/.

[8] *Sphere Online Judge (SPOJ) - User tutorial* [online]. Sphere Research Labs, 2016. [cit. 2016/10/20]. Dostupné z: www.spoj.com/tutorials/USERS/.

[9] *Topcoder about TopCoder* [online]. TopCoder, 2016. [cit. 2016/10/20]. Dostupné z: www.topcoder.com/about-topcoder/.

[10] *Topcoder - Help* [online]. TopCoder, 2016. [cit. 2016/10/22]. Dostupné z: community.topcoder.com/tc?module=Static&d1=help&d2=pracArena.

[11] *Guide to using the Problem set @ Timus Online Judge* [online]. Ural Federal University, 2016. [cit. 2016/10/28]. Dostupné z: acm.timus.ru/help.aspx?topic=judge.

[12] *UVa Online Judge - Verdict information* [online]. Universidad de Valladolid, 2016. [cit. 2016/10/22]. Dostupné z: uva.onlinejudge.org/index.php?option=com_content&task=view&id=16&Itemid=31.

[13] Atkin, A. O. L. – Bernstein, D. J. Prime Sieves Using Binary Quadratic Forms. *Mathematics of Computation*. December 2003, 73, 246, s. 1023–1030. ISSN 0025-5718. doi: 10.1090/S0025-5718-03-01501-1.

[14] Bang-Jensen, J. – Gutin, G. *Digraphs: theory, algorithms, and applications*. Springer, 2009. ISBN 978-0857290410.

[15] Cormen, T. H. et al. *Introduction to Algorithms*. The MIT Press, 3 edition, 2014. ISBN 978-0-262-53305-8.

[16] Kolář, J. *Teoretická informatika*. Česká informatická společnost, 2000. ISBN 80-900853-8-5.

[17] Skiena, S. S. – Revilla, M. A. *Programming challenges: the programming contest training manual*. Springer, 2003. ISBN 0-387-00163-8.

[18] Sorenson, J. An Introduction to Prime Number Sieves. *Computer Sciences Technical Report 909*. January 1990. Dostupné z: `http://research.cs.wisc.edu/techreports/1990/TR909.pdf`.

[19] Černý, J. *Základní grafové algoritmy*. České vysoké učení technické, 2013. ISBN 978-80-01-05258-7.

Seznam použitých zkratek

ACM Association for Computing Machinery - sdružení počítačových profesionálů

Java Objektově orientovaný programovací jazyk vyvinutý firmou Sun Micro-Systems

ANSI C Imperativní nízkoúrovňový programovací jazyk

C++ Objektové rozšíření jazyka C

C++11 Standard jazyka C++ z roku 2011

Python Vysokoúrovňový skriptovací programovací jazyk

LISP List processing - rodina multiparadigmatických programovacích jazyků

Nice Objektově orientovaný programovací jazyk

LUA Imperativní programovací jazyk navržený jako skriptovací jazyk

F Sharp Multiparadigmatický programovací jazyk pro .NET

PHP PHP: Hypertext Preprocessor - skriptovací programovací jazyk určený především pro programování dynamických webových stránek

Clojure Dialekt jazyka LISP

Dart Strukturovaný programovací jazyk vyvíjený společností Google

Swift Multiparadigmatický programovací jazyk vyvinutý společností Apple

ELO Statistické ohodnocení hráče dle jeho herních výsledků

Ruby Interpretovaný skriptovací programovací jazyk

SQL Standard Query Language - dotazovací jazyk pro práci s relačními databázemi

Nim Imperativní programovací jazyk

Julia Vysokoúrovňový programovací jazyk pro numerické výpočty

LOLCODE Ezoterický programovací jazyk používající zkomolené názvy příkazů a klíčových slov

Whitespace Ezoterický programovací jazyk používající netisknutelné znaky jako příkazy

C Sharp Výsokoúrovňový programovací jazyk vyvinutý firmou Microsoft

VB.NET Objektově orientovaný programovací jazyk pro platformu .NET

Go Kompilovaný multiparadigmatický jazyk vytvoří společností Google

Haskell Funkcionální programovací jazyk používající odložené vyhodnocování

Scala Multiparadigmatický programovací jazyk integrující rysy funkcionálního a objektově orientovaného programování

Rust Multiparadigmatický kompilovaný jazyk vyvinutý organizací Mozilla

BFS Breadth-first search, algoritmus prohledávání grafu do šířky

DFS Depth-first search, algoritmus prohledávání grafu do hloubky

FIFO First In, First Out - princip, kdy je první prvek zpracován nejdříve

Seznam obrázků

Seznam příloh

Příloha 1

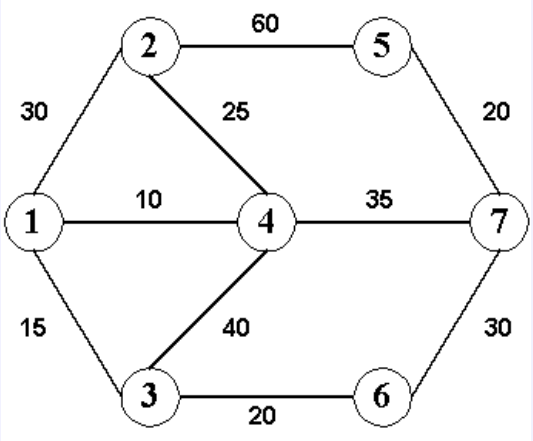
Úloha 10099 - The Tourist Guide

Úloha 10099 - The Tourist Guide

Originální zadání úlohy

10099 - The Tourist Guide

Mr. G. works as a tourist guide. His current assignment is to take some tourists from one city to another. Some two-way roads connect the cities. For each pair of neighboring cities there is a bus service that runs only between those two cities and uses the road that directly connects them. Each bus service has a limit on the maximum number of passengers it can carry. Mr. G. has a map showing the cities and the roads connecting them. He also has the information regarding each bus service. He understands that it may not always be possible for him to take all the tourists to the destination city in a single trip. For example, consider the following road map of 7 cities. The edges connecting the cities represent the roads and the number written on each edge indicates the passenger limit of the bus service that runs on that road.



Now, if he wants to take 99 tourists from city 1 to city 7, he will require at least 5 trips and the route he should take is: 1 - 2 - 4 - 7. But, Mr. G. finds it difficult to find the best route all by himself so that he may be able to take all the tourists to the destination city in minimum number of trips. So, he seeks your help.

Input

The input will contain one or more test cases. The first line of each test case will contain two integers: $N(N \leq 100)$ and R representing respectively the number of cities and the number of road segments. Then R lines will follow each containing three integers: C_1, C_2 and P. C_1 and C_2 are the city numbers and $P(P > 1)$ is the limit on the maximum number of passengers to be carried by the bus service between the two cities. City numbers are positive integers ranging from 1 to N. The $(R+1)$-th line will contain three integers: S , D and T representing respectively the starting city, the destination city and the number of tourists to be guided.

The input will end with two zeroes for N and R.

Output

For each test case in the input first output the scenario number. Then output the minimum number of trips required for this case on a separate line. Print a blank line after the output of each test case.

Sample Input

```
7 10
1 2 30
1 3 15
1 4 10
2 4 25
2 5 60
3 4 40
3 6 20
4 7 35
5 7 20
6 7 30
1 7 99
0 0
```

Sample Output

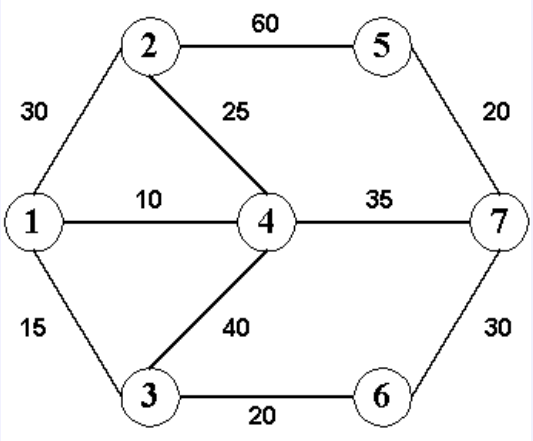
```
Scenario #1
Minimum Number of Trips = 5
```

Z originálního zadání úlohy byl převzat obrázek cest uvedený v zadání. Originální zadání úlohy je dostupné na adrese `https://uva.onlinejudge.org/external/100/10099.pdf` [cit. 2017/04/07].

Česká verze zadání úlohy

10099 - Průvodce turistů

Pan G. pracuje jako průvodce turistů. Jeho nynějším úkolem je dopravit několik turistů z jednoho města do druhého. Města jsou spojena dvousměrnými silnicemi. Mezi každou dvojicí silnicí spojených měst jezdí autobus. Každý autobus má omezení, kolik pasažérů může převézt. Pan G. má k dispozici mapu s městy a silnicemi, která je spojují. Dále také ví, kolik pasažérů uvezou jednotlivé autobusy. Ví, že ne vždy je možné dopravit turisty mezi dvěma městy naráz. Uvažujme následující silniční mapu se sedmi městy. Hrany spojující města představují silnice, ohodnocení hran představuje limit počtu pasažérů, které mohou autobusy na příslušných cestách přepravit.



Chce-li pan G. dopravit 99 turistů z města 1 do města 7, bude muset jet nejméně pětkrát po cestě 1 - 2 - 4 - 7. Pan G. považuje hledání nejlepší cesty za příliš obtížné, proto vás žádá o pomoc s určením nejlepší cesty, aby mohl přepravit turisty s minimálním počtem jízd.

Vstup

Vstupní data sestávají z jednoho anebo více testovacích případů. První řádka každého testovacího případu obsahuje dvě celá čísla - N ($N \leq 100$), které označuje počet měst, a R představující počet silnic mezi městy. Na následujících R řádkách nalezneme vždy tři celá čísla C_1, C_2 a P. C_1 a C_2 jsou čísla měst, která jsou spojena silnicí, a P ($P > 1$) představuje omezení počtu pasažérů v autobusu, který jezdí mezi zadanou dvojicí měst. Města jsou číslována přirozenými čísly od 1 do N. $(R + 1)$-tá řádka obsahuje tři celá čísla S, D a T, která značí počáteční město, cílové město a počet pasažéru k přepravení.

Vstupní data končí dvojicí nul.

Výstup

Pro každý testovací případ vypište pořadí testovacího případu a minimální potřebný počet cest s turisty. Oddělte výstupní data po sobě jdoucích testovacích případů prázdnou řádkou.

Vzorový vstup

```
7 10
1 2 30
1 3 15
1 4 10
2 4 25
2 5 60
3 4 40
3 6 20
4 7 35
5 7 20
6 7 30
1 7 99
0 0
```

Vzorový výstup

```
Scenario #1
Minimum Number of Trips = 5
```

Řešení úlohy

Pro řešení úlohy lze použít dynamické programování. Nejprve si vytvoříme tabulku o rozměrech $N \times N$, kde N značí počet měst, kterou inicializujeme nulami. Následně do tabulky doplníme ohodnocení hran mezi jednotlivými dvojicemi silnicí spojených měst (tedy vytvoříme matici sousednosti).

Následně postupujeme podobně jako při Floyd-Warshallově algoritmu, pouze použijeme jiný vztah pro výpočet matic v jednotlivých iteracích algoritmu. Pro výpočet prvků $d_{u,v}$ matice D^i v i-té iteraci algoritmu použijeme vztah

$$d_{u,v} = \max\{D_{u,v}^{i-1}, \min\{D_{u,i}^{i-1} + D_{i,v}^{i-1}\}\} \tag{4.1}$$

Na konci výpočtu obsahuje matice maximální možný počet osob, které lze přepravit mezi každou dvojicí měst. Následně postačí vydělit počet tu-

ristů maximálním možným počtem osob, které lze na zadané trase přepravit. Nesmíme zapomenout na průvodce, který musí jet v autobuse spolu s turisty.

Výsledek validace

Úloha byla úspěšně zvalidována, doba běhu činila 0,400 sekundy (viz obrázek 4.1).

#		Problem	Verdict	Language	Run Time	Submission Date
19209673	10099	The Tourist Guide	Accepted	JAVA	0.400	2017-04-20 13:38:24

Obrázek 4.1: Zpráva o validaci úlohy The Tourist Guide

Příloha 2

Úloha 12506 - Shortest Names

Úloha 12506 - Shortest Names

Originální zadání úlohy

12506 - Shortest Names

In a strange village, people have very long names. For example: *aaaaa*, *bbb* and *abababab*.

You see, it's very inconvenient to call a person, so people invented a good way: just call a prefix of the names. For example, if you want to call „*aaaaa*“, you can call „*aaa*“, because no other names start with „*aaa*“. However, you can't call „*a*“, because two people's names start with „*a*“. The people in the village are smart enough to always call the shortest possible prefix. It is guaranteed that no name is a prefix of another name (as a result, no two names can be equal).

If someone in the village wants to call every person (including himself/herself) in the village exactly once, how many characters will he/she use?

Input

The first line contains T ($T \leq 10$), the number of test cases. Each test case begins with a line of one integer n ($1 \leq n \leq 1000$), the number of people in the village. Each of the following n lines contains a string consisting of lowercase letters, representing the name of a person. The sum of lengths of all the names in a test case does not exceed 1 000 000.

Output

For each test case, print the total number of characters needed.

Sample Input

```
1
3
aaaaa
bbb
abababab
```

Sample Output

```
5
```

Originální zadání úlohy je dostupné na adrese `https://uva.onlinejudge.org/external/125/12506.pdf` [cit. 2017/04/01].

Česká verze zadání úlohy

12506 - Nejkratší jména

V jedné prapodivné vesnici měli lidé velmi dlouhá jména, například *aaaaa*, *bbb* a *abababab*.
Jak můžete vidět, je velmi nepohodlné říkat osobě celým jménem, proto lidé začali jména zkracovat - lidé se oslovovali jen začátky jmen tak, aby nikdo jiný ve vesnici neměl stejný začátek jména. Například, pokud byste chtěli oslovit osobu „*aaaaa*“, postačí ji oslovit „*aaa*“, protože žádné jiné jméno nezačíná „*aaa*“. Avšak nemůžete tuto osobu oslovit jen „*a*“, protože dvě osoby mají stejný začátek jména. Lidé ve vesnici jsou dost chytří na to, aby se oslovovali nejkratším možným prefixem jména. Je zaručeno, že žádné jméno není prefixem jiného jména (tedy žádná dvě jména nemohou být shodná).
Pokud chce nějaký vesničan oslovit všechny osoby (včetně sebe), kolik potřebuje písmen?

Vstup

První řádka vstupních dat obsahuje číslo T ($T \leq 10$), které udává počet testovacích případů. Každý testovací případ začíná číslem n ($1 \leq n \leq 1000$), které označuje počet osob ve vesnici. Dalších n řádek obsahuje řetězce reprezentující jména jednotlivých vesničanů (všechna písmena jsou malá). Součet délky všech jmen nepřekročí 1 000 000.

Výstup

Pro každý testovací případ vypište počet potřebných písmen pro oslovení všech vesničanů.

Vzorový vstup

```
1
3
aaaaa
bbb
abababab
```

Vzorový výstup

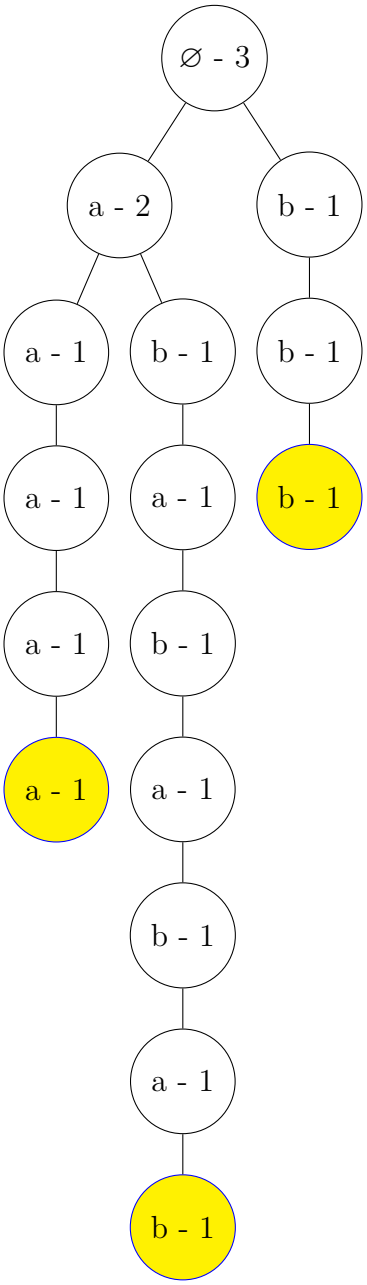
```
5
```

Řešení úlohy

Pro řešení úlohy použijeme datovou strukturu trie. Každému vrcholu trie přiřadíme hodnotu udávající, kolikrát je dané písmeno použito ve jménu nějaké osoby. Pokud je vrchol listem, bude mít hodnotu jedna, pokud nejde

o list, bude hodnota uzlu rovna součtu hodnot potomků. Dále již postačí sečíst všechny hodnoty vrcholů trie, které jsou větší než jedna. Tím dostaneme potřebný počet písmen pro oslovení všech osob ve vesnici.

Pro vzorová vstupní data dostaneme trii zobrazenou na obrázku 4.2. Číslo za pomlčkou značí hodnotu, kolikrát bylo dané písmeno použito ve jménu nějaké osoby. Listy trie jsou označeny žlutě.



Obrázek 4.2: Datová struktura trie pro vzorová vstupní data

Výsledek validace

Úloha byla úspěšně zvalidována, doba běhu činila 0,260 sekundy (viz obrázek 4.3).

#		Problem	Verdict	Language	Run Time	Submission Date
19209688	12506	Shortest Names	Accepted	JAVA	0.260	2017-04-20 13:40:05

Obrázek 4.3: Zpráva o validaci úlohy Shortest Names