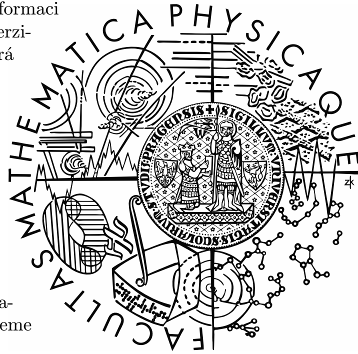
Studium informatiky na MFF UK Praha



Milí řešitelé, se zadáním 1. série úloh jste od nás dostali také přehlednou informaci o tom, jak vypadá studium informatiky na Matematicko-fyzikální fakultě Univerzity Karlovy v Praze. Tentokrát vás chceme seznámit s některými pracovišti, která tuto výuku zajišťují. O dalších informatických pracovištích se dozvíte zase příště. Informatická sekce MFF UK sídlí v budově na Malostranském náměstí 25 v Praze 1. Pokud se tedy rozhodnete pro studium informatiky na naší fakultě, na tomto místě strávíte i vy většinu svého času. Malostranská budova prochází v současné době rozsáhlou rekonstrukcí, při níž se již zmodernizovaly posluchárny, vznikly nové počítačové pracovny a informatické oddělení knihovny. Oproti dřívějšímu stavu sice přibylo posluchárен, dokud ale nebude rekonstrukce dokončena, jejich množství plně nepostačuje pro všechny potřeby výuky. Zejména na výuku matematiky v prvním dvouletí proto jistě budete jezdit i do jiných fakultních budov. Sekce informatiky je tvořena osmi pracovišti. Pojďme je nyní společně postupně navštívit – třeba v tom pořadí, jak je v malostranské budově najdeme cestou po schodech shora dolů. Pro dnešek zůstaneme v nejvyšším 4. patře, kde sídlí:

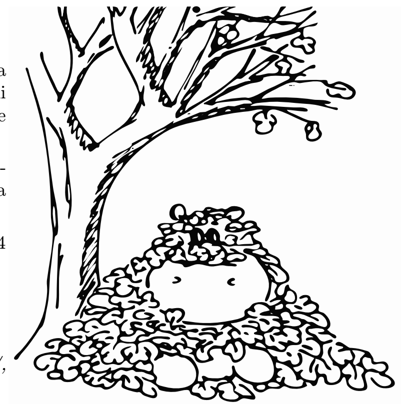
Kabinet software a výuky informatiky (KSVI) je co do počtu zaměstnanců nejmenším informatickým pracovištěm, vzhledem ke svému zaměření má však značný podíl na výuce zajišťované informatickou sekcí. Aktivity KSVI jsou dosti různorodé a lze je přibližně shrnout následujícím výčtem:

- zajišťuje výuku základních kursů programování v 1. ročníku, a to nejen pro posluchače informatiky, ale i pro studenty jiných studijních programů
- zajišťuje odbornou přípravu těch posluchačů fakulty, kteří se připravují na dráhu budoucích učitelů informatiky na středních školách, vede pro ně specializovanou výuku didaktických předmětů, pedagogické praxe apod.
- zajišťuje výuku předmětů zaměřených na pokročilejší práci s běžným aplikačním softwarem
- zajišťuje výuku odborných předmětů z oblasti počítačové geometrie a grafiky a v rámci oboru Softwarové systémy garantuje studijní plán Počítačová grafika
- zajišťuje chod speciální počítačové laboratoře *Carolina*, která slouží k podpoře studia nevidomých a slabozrakých posluchačů univerzity
- organizuje různé propagační a vzdělávací aktivity v oblasti informatiky, které směřují ke studentům i učitelům informatiky na středních školách (programátorské soutěže, přijímací zkoušky, školy učitelů informatiky, apod.).

Středisko informatické sítě a laboratoří (SISAL) je servisním pracovištěm informatické sekce, jehož hlavním úkolem je pečovat o chod a rozvoj síťové infrastruktury v malostranské budově i na celé fakultě a starat se o vybavení a provoz počítačových učeben. SISAL se podílí i na základní výuce pro bakalářské studium, především předmětů blízkých jeho odbornému zaměření (Internet, Unix).

Příště se seznámíme s *Ústavem formální a aplikované lingvistiky* a *Centrem komputační lingvistiky,* které se nacházejí taktéž ve čtvrtém patře malostranské budovy.

Milí řešitelé!



Je tady podzim a s ním druhá série našeho semináře. Tentokrát Vám přišla bez opravené první série. Nicméně dříve, než budete muset tuto druhou sérii odevzdat, dostanete spolu se zadáním třetí série i Vaše opravená řešení série první. Prostě podzim :–)

Pokud chcete posílat svá řešení elektronickou cestou, prosím držte se instrukcí, které můžete najít na *http://ksp.mff.cuni.cz/submit/*. Řešení, která přišla mailem, jsme přijali pouze výjimečně.

Termín odeslání Vašich řešení druhé série jest stanoven na 13. prosince 2004 a naše adresa je stále stejná:

Korespondenční seminář z programování
KSVI MFF UK
Malostranské náměstí 25
118 00 Praha 1

Aktuální informace o KSP můžete nalézt na stránkách *http://ksp.mff.cuni.cz/*, dotazy organizátorům můžete posílat e-mailem na adresu *ksp@mff.cuni.cz*.

Zadání druhé série sedmnáctého ročníku KSP

17-2-1 Prasátko Květ(ák)omil — 10 bodů

Květomil byl úplně normální prasátko. Již v útlém dětství ničím nevynikal mezi svými vrstevníky, své rodiče nepřekvapoval svou předčasnou duševní vyspělostí. Ani později nijak nezastiňoval své přátele a známé v žádné činnosti, kterou prováděl, snad s jedinou výjimkou, a tou bylo jídlo (proto si vysloužil přezdívku Pašík Kvašík). Byl prostě úplně normální obyčejné prasátko.

Když vyrostl, zvolil si úplně normální obyčejné povolání a stal se programátorem u firmy *Ptáček Sáček, práce všeho druhu.* Jeho práce u této firmy (konkurující známému příteli Ferdy Mravence) byla také úplně normální a obyčejná. Proto, když Sáček kontroloval práci svých zaměstnanců, aby zjistil, proč jeho konkurence dodává rychlejší programy, zjistil, že programy Pašíka Kvašíka jsou pomalé až hrůza. Prostě úplně normální a obyčejné.

A tak si Sáček najal Vás, abyste mu pomohli Kvašíkovy programy zrychlit. Kvašíkovy programy jsou posloupnosti přiřazení do proměnných, což jsou řetězce znaků složené z malých písmen, velkých písmen a podtržítek. Na pravé straně přiřazení může být buď proměnná nebo operace „+“ nebo „∗“ aplikovaná na dvě proměnné. Tyto operace jsou komutativní, neboli $a+b=b+a$ a $a*b=b*a$.

Vaším úkolem je napsat program, který dostane Kvašíkův program skládající se z N přiřazení a má říci, jak moc ho lze zrychlit, čili říci, kolik nejméně operací „+“ a „∗“ stačí k tomu, aby nový program přiřadil do všech proměnných stejnou hodnotu jako Kvašíkův. Formálně pro každých i prvních řádků Kvašíkova programu musí v novém programu existovat místo, kdy jsou hodnoty všech proměnných z Kvašíkova programu v obou programech shodné. Můžete využívat toho, že operace „+“ a „∗“ jsou komutativní, ale jejich asociativita a distributivita se neberou v úvahu, čili $a+(b+c) \neq (a+b)+c$ a také $(a+b)*c \neq a*c+b*c$.

Příklad: Vlevo je Kvašíkův program, vpravo náš.

```
                    t = b + c;
a = b + c;          a = t;
d = a + b;          d = a + b;
e = c + b;          e = t;
                    s = a * e;
f = a * e;          f = s;
a = d;              a = d;
g = e * e;          g = s;
```

Zatímco Kvašíkův program potřeboval operací pět, náš si vystačí se třemi, takže výstup programu by měl být „3“.

17-2-2 Bobr Béďa — 10 bodů

Béďa byl hodný bobr, který poslouchal svou maminku. A ta ho, jako každá jiná maminka, naučila čistit si zoubky. Když Béďa vyrostl, začal používat zubní pastu *Bělosup*, po které, jak bylo na jejím obalu napsáno, „zuby nádherně vypadají.“

A byla to pravda. Hodnému Béďovi vypadaly všechny zuby. Dostal sice samozřejmě umělé, ale protože bobři vyrábějí vše ze dřeva, byly celé dřevěné. Leč s dřevěnými zuby Béďa nemohl chodit do normální bobří práce, protože by sotva přehryzal dřevěný strom. A tak začal pracovat u firmy Ptáčka Sáčka.

Přestože byly Béďovy zuby dřevěné, stále dokázal skvěle pracovat se dřevem, a tak byl zaměstnán jako výrobce integrovaných odvodů. Integrovaný odvod je součástka na rozvod vody. Představit si ji můžete jako dřevěnou desku, na které je pravidelná čtvercová síť bodů, některé sousední body jsou spojeny vydlabanou spojnicí. Za sousední body se považují takové, které se liší v jedné souřadnici o jedničku (čili vnitřní body mají každý čtyři sousedy). Béďův úkol je dodělat na odvod nějaké spojnice sousedních bodů, aby bylo možné dostat se z každého bodu do jiného.

Protože je integrovaný odvod ze dřeva, dokáže Béďa vytvořit svislé spojnice rychleji než vodorovné (jdou „po letech“). Změřil si, že udělat jednu vodorovnou nebo dvě svislé spojnice mu zabere stejně času. A protože je Béďa hodný, chce mít každý odvod co nejrychleji hotový.

Napište Béďovi program, který dostane na vstupu N a M (rozměry mřížky bodů na integrovaném odvodu), S (počet již hotových spojnic), a popis jednotlivých spojnic (souřadnice dvou bodů, které spojuje). Výstupem by měl být seznam spojnic takových, že po jejich přidání do integrovaného odvodu se půjde dostat z každého bodu do každého a navíc doba na vytvoření těchto spojnic bude co nejmenší (čili neexistuje jiná množina spojnic, která by se dala vyrobit v kratším čase a přitom by splnila popsanou podmínku).

Příklad: Pro $N=4, M=4$ a odvod

je pro Béďu nejlepší vytvořit spojnice nakreslené dvojitě.

17-2-3 Krkavec Kryšpín — 8 bodů

Krkavec Kryšpín byl velmi známý a uznávaný básník, snad každý se obdivoval jeho poezii. Což ale znamená, že to byl básník velmi zaneprázdněný, protože každé zvířátko po něm chtělo jinou básničku. Jako každý básník i Kryšpín potřeboval inspiraci – zpěv ptáků. Ovšem po čase se mu všichni ptáci začali vyhýbat, nebavilo je věčně stát před krkavcem, který je neustále napomínal, ať nekrákají.

To se Kryšpínovi ani trochu nelíbilo a usmyslel si, že zpěvavé ptáky nějak naláká. Rozhodl se, že jim postaví fontánu roztodivného tvaru – ptáci budou obdivovat fontánu (hned ji překřtil na Fontárnu, když u ní bude psát básně), on ptačí zpěv a ostatní jeho poezii.

Hned vyrobil zkušební Fontárničku, ale zjistil, že potřebuje najít její těžiště, aby mu nepadala ze stojánku. Vypravil se za Ptáčkem Sáčkem, zda by mu jeho firma mohla pomoci, ale Sáček se mu jenom vysmál, protože nechtěl zradit své ptačí přátele. Dokážete Kryšpínovi poradit Vy?

Na vstupu dostanete N bodů zadaných svými souřadnicemi, které představují Fontárnu. Tu si můžete představit jako mnohoúhelník, který vznikne, spojíme-li vždy dva po sobě jdoucí zadané body (a ještě první s posledním). Tento mnohoúhelník je navíc konvexní, což znamená, že všechny jeho vnitřní úhly jsou menší než 180°. Vaším úkolem je najít *těžiště* zadaného mnohoúhelníka.

Příklad: Pro $N = 4$ a body $[0,0], [12,6], [12,12], [0,18]$ by měl Váš program odpovědět, že těžiště se nachází na souřadnicích $[5,9]$.

Pro náročnější: Pokud bude Váš program fungovat i pro nekonvexní mnohoúhelníky, můžete dostat další 3 body.

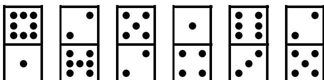
17-2-4 Mravenec Ferda — 11 bodů

Ferdova tajná láska, Beruška, přišla jednou za ním a jeho přítelem Pytlíkem na návštěvu. K Ferdově veliké lítosti ale odmítla jeho návrh najít stonožku Šmajdulu a svázat jí nohy dohromady, a místo toho se podivovala Pytlíkově kvetoucí firmě. Zklamaný Ferda se rozhodl Berušce předvést, že co dokáže jeho přítel, dokáže taky. Aby ale nemusel začínat s prázdnými tykadly, rozhodl se, že se stane vedoucím firmy *Ptáček Sáček, práce všeho druhu.*

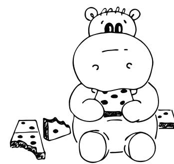
Příští den zašel do Sáčkovy firmy a spustil: „Podívejte se na něj, na ptáčka. Zaměstnává naprosto neschopné programátory, chudáčkovi Bobrovi vyrazil zuby, aby pro něj vyráběl odvody a je to takový trumbera, že ani nedokáže spočítat těžiště. A takové zvíře Vám má šéfovat?“ Zvířátka uznala, že na tom je něco pravdy, a rozhodla se dát Ferdovi šanci. Když vyřeší jejich úlohu, stane se jejich šéfem.

Zvířátka položila před Ferdu N kostiček domina, na každém jsou nahoře a dole dvě celá čísla od 1 do K. Každou kostičku domina může Ferda obrátit, čili její horní číslo se dostane dolů a naopak. Jeho úkolem je dosáhnout toho, aby rozdíl součtu horní a dolní řady čísel na kostičkách byl co nejmenší. A navíc toho má dosáhnout přehozením co nejmenšího počtu kostiček. Ferda ale zjistil, že je to i nad jeho mravenčí síly. Pomůžete mu?

Příklad: Pro $K = 8$ a $N = 6$ a dominové kostky



musí Ferda otočit druhou, třetí a pátou dominovou kostku.



Pozor: I sám Ferda si po chvíli přemýšlení uvědomil, že nestačí najít si kostičku s největším rozdílem čísel, která by mu pomohla snížit celkový rozdíl, otočit ji, a podle stejného postupu pokračovat dál (to nezabere ani pro uvedený příklad). Kdyby to bylo takhle jednoduché, určitě by Vás o pomoc nepožádal.

17-2-5 Jazykozpytcova pomsta — 10 bodů

V druhém dílu seriálu o formálních jazycích se ještě stále budeme věnovat nejjednodušší jazykové rodině, regulárním jazykům. Minule jsme si řekli, co jsou to gramatiky, konečné automaty a regulární jazyky, a nyní si zavedeme věc, která se může na první pohled jevit jako naprostý nesmysl – *nedeterminismus.* Pokud pracuje nějaký proces (stroj, algoritmus, …) deterministicky, pak pokud známe jeho vstupní data, jsme schopni dopředu předpovědět, jak se bude chovat. Ovšem u nedeterministického procesu nic takového určit nemůžeme.

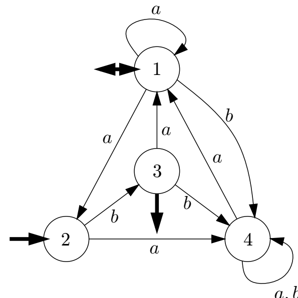
Pokud je konečný automat ve stavu q a načte nové písmeno p, je pomocí přechodové funkce přesně definováno, jaký bude nový stav, do kterého stroj přejde. Ovšem nedeterministický konečný automat má k jedné kombinaci stavu a písmena na výběr hned několik možných stavů, do kterých může přejít, a z těch si jeden naprosto libovolně vybere.

Formálně je nedeterministický konečný automat (NKA) pětice (Q, A, S, δ, F), kde

- Q je konečná množina stavů,
- A je konečná abeceda, nad kterou automat pracuje,
- S je množina počátečních stavů,
- $\delta : Q \times A \to P(Q)$ je přechodová funkce, která ke každé kombinaci aktuálního stavu a nově načteného písmenka vrací neprázdnou množinu stavů (tedy nějakou podmnožinu Q), do kterých automat může přejít,
- $F \subseteq Q$ je množina přijímacích stavů.

Zbývá nadefinovat, kdy je či není dané slovo nedeterministickým konečným automatem přijato. Výpočtem NKA nad slovem w rozumíme konkrétní průběh činnosti stroje při postupném čtení písmen slova w. Díky nedeterminismu je možných výpočtů pro jediné slovo více. Slovo w je tedy přijato, jestliže mezi všemi možnými výpočty nad slovem w existuje alespoň jediný, který končí v přijímacím stavu.

Příklad NKA nad abecedou $A = \{a, b\}$:



Stroj používá stavy $Q = \{1,2,3,4\}$, počáteční stavy jsou $S = \{1,2\}$ a koncové $F = \{1,3\}$. Ve stavech 1 a 4 jsou dvě možnosti, jak se při načtení písmena a může stroj zachovat.

ale je o trošku pracnější), nebo použijeme následující fígl: když budeme nějaký prvek mazat, najdeme ho a označíme jako smazaný. Nicméně při hledání nějakého jiného prvku se nemůžeme zastavit na tomto smazaném prvku, ale musíme hledat i za ním. Ovšem pokud nějaký prvek přidáváme, můžeme jím smazaný prvek přepsat.

A jakou hešovací funkci tedy použít? To je tak trochu magie a dobré hešovací funkce mají mimo jiné hlubokou souvislost s kryptografií a s generátory pseudonáhodných čísel. Obvykle se dělá to, že se hešovaný objekt rozloží na posloupnost čísel (třeba ASCII kódů písmen v řetězci), tato čísla se nějakou operací „slejí“ dohromady a výsledek se vezme modulo K. Operace na slévání se používají různé, od jednoduchého xoru až třeba po komplikované vzorce typu

```
#define mix(a,b,c) {                              \
  a-=b; a-=c; a^=(c>>13);                         \
  b-=c; b-=a; b^=(a<< 8);                         \
  c-=a; c-=b; c^=((b&0xffffffff)>>13);            \
  a-=b; a-=c; a^=((c&0xffffffff)>>12);            \
  b-=c; b-=a; b =(b ^ (a<<16)) & 0xffffffff;      \
  c-=a; c-=b; c =(c ^ (b>> 5)) & 0xffffffff;      \
  a-=b; a-=c; a =(a ^ (c>> 3)) & 0xffffffff;      \
  b-=c; b-=a; b =(b ^ (a<<10)) & 0xffffffff;      \
  c-=a; c-=b; c =(c ^ (b>>15)) & 0xffffffff;      \
}
```

My se ale spokojíme s málem a ukážeme si jednoduchý způsob, jak hešovat čísla a řetězce. Pro čísla stačí zvolit za velikost tabulky vhodné prvočíslo a klíč vymodulit tímto prvočíslem. (S hledáním prvočísel si samozřejmě nemusíme dělat starosti, v praxi dobře poslouží tabulka několika prvočísel přímo uvedená v programu.)

Rozumná funkce pro hešování řetězců je třeba

```
unsigned hash_string (unsigned char *str)
{
  unsigned r = 0;
  unsigned char c;

  while ((c = *str++) != 0)
    r = r * 67 + c - 113;

  return r;
}
```

Zde můžeme použít vcelku libovolnou velikost tabulky, která nebude dělitelná čísly 67 a 113. Šikovné je vybrat si například mocninu dvojky (což v příštím odstavci oceníme), ta bude s prvočísly 67 a 113 zaručeně nesoudělná. Jen si musíme dávat pozor, abychom nepoužili tak velkou hešovací tabulku, že by 67 umocněno na obvyklou délku řetězce bylo menší než velikost tabulky (čili by hešovací funkce časteji volila začátek heše než konec). Tehdy ale stačí místo našich čísel použít jiná, větší prvočísla.

A co když nestačí pevná velikost heše? Použijeme „nafukovací“ heš. Na začátku si zvolíme nějakou pevnou velikost, sledujeme počet vložených prvků a když se jich zaplní víc než polovina (nebo třeba třetina; menší číslo znamená větší rychlost [méně kolizí], ale větší paměťové plýtvání), vytvoříme nový heš dvojnásobné velikosti (případně zaokrouhlené na vyšší prvočíslo, pokud to naše hešovací funkce vyžaduje) a starý heš do něj prvek po prvku vložíme.

To na první pohled vypadá velice neefektivně, ale protože se po každém nafouknutí heš zvětší na dvojnásobek, musí mezi přehešováním na N prvků a na $2N$ přibýt alespoň N prvků, čili průměrně provádíme jedno přehešování na každý vložený prvek.

Pokud navíc používáme mazání prvků popsané výše (u prvku si pamatujeme, že je smazaný, ale stále zabírá místo v heši), nemůžeme při mazání takového prvku snížit počet prvků v heši, ale na druhou stranu při nafukování můžeme takové prvky opravdu smazat (a konečně je odečíst z počtu obsazených prvků).

Pár poznámek na závěr:

- S hešováním se separovanými řetězci se zachází podobně, nafukování také funguje a navíc je snadno vidět, že po vložení N náhodných prvků bude v každé přihrádce (přihrádky odpovídají hodnotám hešovací funkce) průměrně N/K prvků, čili pro K velké řádově jako N konstantně mnoho. Pro srůstající řetězce to pravda být nemusí (protože jakmile jednou vznikne dlouhý řetězec, nově vložené prvky mají sklony „nalepovat se“ za něj), ale platí, že bude-li heš naplněna nejvýše na polovinu, průměrná délka kolizního řetízku bude omezená nějakou konstantou nezávislou na počtu prvků a velikosti heše. Důkaz si ovšem raději odpustíme, není úplně snadný.
- Bystrý čtenář si jistě všiml, že v případě prvočíselných velikostí heše jsme v důkazu časové složitosti nafukování trochu podváděli – z heše velikosti N přeci přehešováváme do heše velikosti větší než $2N$. Zachrání nás ale věta z teorie čísel, obvykle zvaná Bertrandův postulát, která říká, že mezi čísly t a $2t$ se vždy nachází alespoň jedno prvočíslo. Takže nový heš bude maximálně 4-krát větší, a tedy počet přehešování na jedno vložení bude nadále omezen konstantou.

Tips & Tricks: Z letáků KSP si můžete složit knížečku

Ačkoli to tak na první pohled rozhodně nevypadá, přidáním nedeterminismu do konečných automatů jsme ve skutečnosti nijak nezvedli „výpočetní sílu“ stroje.

Tvrzení. *Množina jazyků přijímaných deterministickými KA je stejná jako množina jazyků přijímaných nedeterministickými KA.*

Jinými slovy, ke každému NKA jsme schopni sestrojit ekvivalentní (přijímající stejný jazyk) DKA a naopak. Tato skutečnost rozhodně stojí za to být dokázána. První převod je jednoduchý, každý DKA je totiž pouze případem takového NKA, jehož přechodová funkce vrací vždy jednoprvkovou množinu stavů. Převod druhý je však už o poznání těžší.

Mějme libovolný nedeterministický konečný automat $M = (Q, A, S, \delta, F)$ a hledejme k němu ekvivalentní deterministický konečný automat $M' = (Q', A, q_0', \delta', F')$. Nejprve se zamysleme, jak bychom asi M simulovali „programátorsky“. Nejspíše bychom si napsali program, který pro vstupní slovo probacktrackuje všechny možné výpočty. Další metodou je v každé situaci, kdy se M rozhoduje z několika možností, spustit pro každou takovou možnost paralelně proces, který ji dále simuluje. Právě na této myšlence je založena naše konstrukce. Jak ovšem takový postup namodelovat omezenými prostředky konečných automatů?

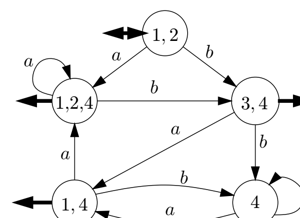
V novém konečném automatu M' především podstatně rozšíříme množinu stavů, položíme $Q' = P(Q)$, kde $P(Q)$ značí množinu všech podmnožin množiny stavů Q původního stroje M. Jeden stav stroje M' tedy bude kódován hned několika stavy stroje M. Počáteční stav bude $q_0' = S$, čili množina obsahující všechny počáteční stavy stroje M. Přechodovou funkci $\delta'(q', a)$ pro $q' = \{q_1, q_2, \ldots, q_k\} \in Q'$ a $a \in A$ definujeme takto:

$$\delta'(\{q_1, q_2, \ldots, q_k\}, a) = \delta(q_1, a) \cup \delta(q_2, a) \cup \cdots \cup \delta(q_k, a)$$

Všimněte si, že v jednom stavu z Q' jsme schopni najednou simulovat hned několik stavů původního stroje. Stejně tak nová přechodová funkce δ' provede paralelní výpočet hned pro několik stavů původního stroje najednou. Zbývá položit $F' = \{q' \in Q';\ \exists q \in q' : q \in F\}$, čili přijímací stav stroje M' je taková množina stavů stroje M, ve které se vyskytuje alespoň jeden přijímací stav stroje M.

Právě jsme si ale uvědomili, že v jednom stavu z Q' paralelně simulujeme několik různých výpočtů původního stroje. Podle definice přijímání nedeterministickým konečným automatem je přijímací stav z F' takový, že alespoň jeden výpočet z odpovídající množiny výpočtů stroje M je přijímací. Oba stroje tedy přijímají stejný jazyk, čímž je důkaz hotov. Počet stavů nového stroje M' nám sice oproti M exponenciálně narostl, ale je stále konečný a splňuje tak definici konečného automatu.

Nás ukázkový stroj tedy po převodu bude vypadat takto (nikdy nedosažitelné stavy z obrázku vynecháme):



Zavedením nedeterminismu jsme tedy nijak nerozšířili možnosti konečných automatů, nicméně právě předvedená věta se dá třebas využít k zjednodušení nejrůznějších důkazů. O strojích, jejichž nedeterministická verze má větší výpočetní sílu než verze deterministická, si povíme v příštích dílech seriálu.

Ale nyní už asi netrpělivě čekáte na další

Soutěžní úlohy:

- Každý stroj má svá omezení a nejinak tomu je v i případě konečného automatu. Nechť U je jazyk nad dvoupísmennou abecedou $\{(,)\}$, který je tvořen všemi správně uzávorkovanými výrazy. Např. slovo $()(())$ patří do U, slovo $))($ do U nepatří. Pokud možno formálně dokažte, že nemůže existovat konečný automat, který by rozpoznával jazyk U. *[6 bodů]*
- Nechť L_1 a L_2 jsou libovolné regulární jazyky. Zřetězením regulárních jazyků L_1 a L_2 nazveme jazyk $L_1.L_2 = \{uv;\ u \in L_1, v \in L_2\}$. Tedy např. pro $L_1 = \{a, ab\}$ a $L_2 = \{c^i;\ i \in N\}$ je $L_1.L_2 = \{ac, abc, acc, abcc, \ldots\}$. Dokažte, že $L_1.L_2$ je také regulární jazyk. *[5 bodů]*

Připomínáme, že vaše řešení by měla obsahovat matematicky správné zdůvodnění a **nikoli zdrojový kód programu** počítajícího bůhvíco.

Recepty z programátorské kuchařky

V tomto dílu programátorské kuchařky si povíme něco o hešování. (V literatuře se také často setkáme s jinými přepisy tohoto anglicko-českého patvaru (hashování), či více či méně zdařilými pokusy se tomuto slovu zcela vyhnout a místo „heš“ používat například termín asociativní pole.) Na heš se můžeme dívat jako na pole, které ale neindexujeme po sobě následujícími přirozenými čísly, ale hodnotami nějakého jiného typu (řetězci, velkými čísly, apod.). Hodnotě, kterou heš indexujeme, budeme říkat *klíč*. K čemu nám takové pole může být dobré?

- Aplikace typu slovník – máme zadán seznam slov a jejich významů a chceme k zadanému slovu rychle najít jeho význam. Vytvoříme si heš, kde klíče budou slova a hodnoty jim přiřazené budou překlady.
- Rozpoznávání klíčových slov (například v překladačích programovacích jazyků) – klíče budou klíčová slova, hodnoty jim přiřazené v tomto příkladě moc význam nemají, stačí nám vědět, zda dané slovo v heši je.
- V nějaké malé části programu si u objektů, se kterými pracujeme, potřebujeme pamatovat nějakou informaci navíc a nechceme kvůli tomu do objektu přidávat nové datové položky (třeba proto, aby nám zbytečně nezabíraly paměť v ostatních částech programu). Klíčem heše budou příslušné objekty.
- Potřebujeme najít v seznamu objekty, které jsou „stejné“ podle nějakého kritéria (například v seznamu osob ty, co se stejně jmenují). Klíčem heše je jméno. Postupně procházíme seznam a pro každou položku zjišťujeme, zda už je v heši uložena nějaká osoba se stejným jménem. Pokud není, aktuální položku přidáme do heše.

Potřebovali bychom tedy umět do heše přidávat nové hodnoty, najít hodnotu pro zadaný klíč a případně také umět z heše nějakou hodnotu smazat.

Samozřejmě používat jako klíč libovolný typ, o kterém nic nevíme (speciálně ani to, co znamená, že dva objekty toho typu jsou stejné), dost dobře nejde. Proto potřebujeme

ještě *hešovací funkci* – funkci, která objektu přiřadí nějaké malé přirozené číslo $0 \le x < K$, kde K je velikost heše (ta by měla odpovídat počtu objektů N, které v ní chceme uchovávat; v praxi bývá rozumné udělat si heš o velikosti zhruba $K = 2N$). Dále popsaný postup funguje pro libovolnou takovou funkci, nicméně aby také fungoval rychle, je potřeba, aby hešovací funkce byla dobře zvolena. K tomu, co to znamená, si něco řekneme níže, prozatím nám bude stačit představa, že tato funkce by měla rozdělovat klíče zhruba rovnoměrně, tedy že pravděpodobnost, že dvěma klíčům přiřadí stejnou hodnotu, by měla být zhruba $1/K$.

Ideální případ by nastal, kdyby se nám podařilo nalézt funkci, která by každým dvěma klíčům přiřazovala různou hodnotu (i to se může podařit, pokud množinu klíčů, které v heši budou, známe dopředu – viz třeba příklad s rozpoznáváním klíčových slov v překladačích). Pak nám stačí použít jednoduché pole velikosti K, jehož prvky budou obsahovat jednak hodnotu klíče, jednak jemu přiřazená data:

```
struct položka_heše
{
  int obsazeno;
  typ_klíče klíč;
  typ_hodnoty hodnota;
} heš[K];
```

A operace naprogramujeme zřejmým způsobem:

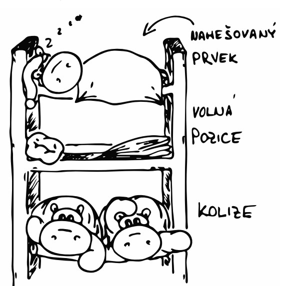
```
void přidej (typ_klíče klíč,typ_hodnoty hodnota)
{
  unsigned index = hešovací_funkce (klíč);

  // Kolize nejsou, čili heš[index].obsazeno=0.
  heš[index].obsazeno = 1;
  heš[index].klíč = klíč;
  heš[index].hodnota = hodnota;
}

int najdi (typ_klíče klíč, typ_hodnoty *hodnota)
{
  unsigned index = hešovací_funkce (klíč);

  // Nic tu není nebo je tu něco jiného.
  if (!heš[index].obsazeno ||
      !stejný (klíč, heš[index].hodnota))
    return 0;

  // Našel jsem.
  *hodnota = heš[index].hodnota;
  return 1;
}
```

Normálně samozřejmě takové štěstí mít nebudeme a vyskytnou se klíče, jimž hešovací funkce přiřadí stejnou hodnotu (říká se, že nastala *kolize*). Co potom?

Jedno z řešení je založit si pro každou hodnotu hešovací funkce seznam, do kterého si uložíme všechny prvky s touto hodnotou. Funkce pro vkládání pak bude v případě kolize přidávat do seznamu, vyhledávací funkce si vždy spočítá hodnotu hešovací funkce a projde celý seznam pro tuto hodnotu. Tomu se říká *hešování se separovanými řetězci.*

Jiná možnost je v případě kolize uložit kolidující hodnotu na první následující volné místo v poli (cyklicky, tj. dojdeme-li ke konci pole, pokračujeme na začátku). Samozřejmě pak musíme i příslušně upravit hledání – snadno si rozmyslíme, že musíme projít všechny položky od pozice, kterou nám poradí hešovací funkce, až po první nepoužitou položku. Tento přístup se obvykle nazývá *hešování se srůstajícími řetězci* (protože seznamy hodnot odpovídající různým hodnotám hešovací funkce se nám mohou spojit). Implementace pak vypadá takto:

```
void přidej (typ_klíče klíč,typ_hodnoty hodnota)
{
  unsigned index = hešovací_funkce (klíč);

  while (heš[index].obsazeno)
  {
    index++;
    if (index == K)
      index = 0;
  }

  heš[index].obsazeno = 1;
  heš[index].klíč = klíč;
  heš[index].hodnota = hodnota;
}

int najdi (typ_klíče klíč, typ_hodnoty *hodnota)
{
  unsigned index = hešovací_funkce (klíč);

  while (heš[index].obsazeno)
  {
    if (stejný (klíč, heš[index].klíč))
    {
      *hodnota = heš[index].hodnota;
      return 1;
    }

    // Něco tu je, ale ne to, co hledám.
    index++;
    if (index == K)
      index = 0;
  }

  // Nic tu není.
  return 0;
}
```



Jaká je časová složitost tohoto postupu? V nejhorším případě bude mít všech N objektů stejnou hodnotu hešovací funkce. Pak hledání může přeskakovat postupně všechny, čili složitost v nejhorším případě může být až $O(NT + H)$, kde T je čas pro porovnání dvou klíčů a H je čas na spočtení hešovací funkce. Laicky řečeno, pro nalezení jednoho prvku budeme muset projít celý heš (v lineárním čase).

Nicméně tohle se nám obvykle nestane – pokud velikost pole bude dost velká (alespoň dvojnásobek prvků heše) a zvolili jsme dobrou hešovací funkci, pak v průměrném případě bude potřeba udělat pouze konstantně mnoho porovnání, tj. časová složitost hledání i přidávání bude jen $O(T + H)$. A budeme-li schopni prvky hešovat i porovnávat v konstantním čase (což například pro čísla není problém), získáme konstantní časovou složitost obou operací.

Mazání prvků může působit menší problémy (rozmyslete si, proč nelze prostě nastavit u mazaného prvku „obsazeno“ na 0). Pokud to potřebujeme dělat, buď musíme použít separované řetězce (což se může hodit i z jiných důvodů,

Ačkoli to tak na první pohled rozhodně nevypadá, přidáním nedeterminismu do konečných automatů jsme ve skutečnosti nijak nezvedli „výpočetní sílu“ stroje.

Tvrzení. *Množina jazyků přijímaných deterministickými KA je stejná jako množina jazyků přijímaných nedeterministickými KA.*

Jinými slovy, ke každému NKA jsme schopni sestrojit ekvivalentní (přijímající stejný jazyk) DKA a naopak. Tato skutečnost rozhodně stojí za to být dokázána. První převod je jednoduchý, každý DKA je totiž pouze případem takového NKA, jehož přechodová funkce vrací vždy jednoprvkovou množinu stavů. Převod druhý je však už o poznání těžší.

Mějme libovolný nedeterministický konečný automat $M = (Q, A, S, \delta, F)$ a hledejme k němu ekvivalentní deterministický konečný automat $M' = (Q', A, q_0', \delta', F')$. Nejprve se zamysleme, jak bychom asi M simulovali „programátorsky“. Nejspíše bychom si napsali program, který pro vstupní slovo probacktrackuje všechny možné výpočty. Další metodou je v každé situaci, kdy se M rozhoduje z několika možností, spustit pro každou takovou možnost paralelně proces, který ji dále simuluje. Právě na této myšlence je založena naše konstrukce. Jak ovšem takový postup namodelovat omezenými prostředky konečných automatů?

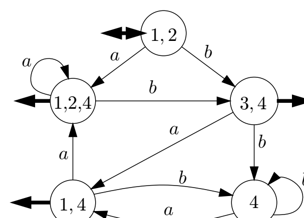
V novém konečném automatu M' především podstatně rozšíříme množinu stavů, položíme $Q' = P(Q)$, kde $P(Q)$ značí množinu všech podmnožin množiny stavů Q původního stroje M. Jeden stav stroje M' tedy bude kódován hned několika stavy stroje M. Počáteční stav bude $q_0' = S$, čili množina obsahující všechny počáteční stavy stroje M. Přechodovou funkci $\delta'(q', a)$ pro $q' = \{q_1, q_2, \ldots, q_k\} \in Q'$ a $a \in A$ definujeme takto:

$$\delta'(\{q_1, q_2, \ldots, q_k\}, a) = \delta(q_1, a) \cup \delta(q_2, a) \cup \cdots \cup \delta(q_k, a)$$

Všimněte si, že v jednom stavu z Q' jsme schopni najednou simulovat hned několik stavů původního stroje. Stejně tak nová přechodová funkce δ' provede paralelní výpočet hned pro několik stavů původního stroje najednou. Zbývá položit $F' = \{q' \in Q';\ \exists q \in q' : q \in F\}$, čili přijímací stav stroje M' je taková množina stavů stroje M, ve které se vyskytuje alespoň jeden přijímací stav stroje M.

Právě jsme si ale uvědomili, že v jednom stavu z Q' paralelně simulujeme několik různých výpočtů původního stroje. Podle definice přijímání nedeterministickým konečným automatem je přijímací stav z F' takový, že alespoň jeden výpočet z odpovídající množiny výpočtů stroje M je přijímací. Oba stroje tedy přijímají stejný jazyk, čímž je důkaz hotov. Počet stavů nového stroje M' nám sice oproti M exponenciálně narostl, ale je stále konečný a splňuje tak definici konečného automatu.

Nás ukázkový stroj tedy po převodu bude vypadat takto (nikdy nedosažitelné stavy z obrázku vynecháme):



Zavedením nedeterminismu jsme tedy nijak nerozšířili možnosti konečných automatů, nicméně právě předvedená věta se dá třebas využít k zjednodušení nejrůznějších důkazů. O strojích, jejichž nedeterministická verze má větší výpočetní sílu než verze deterministická, si povíme v příštích dílech seriálu.

Ale nyní už asi netrpělivě čekáte na další

Soutěžní úlohy:

- Každý stroj má svá omezení a nejinak tomu je v i případě konečného automatu. Nechť U je jazyk nad dvoupísmennou abecedou $\{(,)\}$, který je tvořen všemi správně uzávorkovanými výrazy. Např. slovo ()(()) patří do U, slovo))(do U nepatří. Pokud možno formálně dokažte, že nemůže existovat konečný automat, který by rozpoznával jazyk U. *[6 bodů]*
- Nechť L_1 a L_2 jsou libovolné regulární jazyky. Zřetězením regulárních jazyků L_1 a L_2 nazveme jazyk $L_1.L_2 = \{uv;\ u \in L_1, v \in L_2\}$. Tedy např. pro $L_1 = \{a, ab\}$ a $L_2 = \{c^i;\ i \in N\}$ je $L_1.L_2 = \{ac, abc, acc, abcc, \ldots\}$. Dokažte, že $L_1.L_2$ je také regulární jazyk. *[5 bodů]*

Připomínáme, že vaše řešení by měla obsahovat matematicky správné zdůvodnění a **nikoli zdrojový kód programu** počítajícího bůhvíco.

Recepty z programátorské kuchařky

V tomto dílu programátorské kuchařky si povíme něco o hešování. (V literatuře se také často setkáme s jinými přepisy tohoto anglicko-českého patvaru (hashování), či více či méně zdařilými pokusy se tomuto slovu zcela vyhnout a místo „heš“ používat například termín asociativní pole.) Na heš se můžeme dívat jako na pole, které ale neindexujeme po sobě následujícími přirozenými čísly, ale hodnotami nějakého jiného typu (řetězci, velkými čísly, apod.). Hodnotě, kterou heš indexujeme, budeme říkat *klíč*. K čemu nám takové pole může být dobré?

- Aplikace typu slovník – máme zadán seznam slov a jejich významů a chceme k zadanému slovu rychle najít jeho význam. Vytvoříme si heš, kde klíče budou slova a hodnoty jim přiřazené budou překlady.
- Rozpoznávání klíčových slov (například v překladačích programovacích jazyků) – klíče budou klíčová slova, hodnoty jim přiřazené v tomto příkladě moc význam nemají, stačí nám vědět, zda dané slovo v heši je.
- V nějaké malé části programu si u objektů, se kterými pracujeme, potřebujeme pamatovat nějakou informaci navíc a nechceme kvůli tomu do objektu přidávat nové datové položky (třeba proto, aby nám zbytečně nezabíraly paměť v ostatních částech programu). Klíčem heše budou příslušné objekty.
- Potřebujeme najít v seznamu objekty, které jsou „stejné“ podle nějakého kritéria (například v seznamu osob ty, co se stejně jmenují). Klíčem heše je jméno. Postupně procházíme seznam a pro každou položku zjišťujeme, zda už je v heši uložena nějaká osoba se stejným jménem. Pokud není, aktuální položku přidáme do heše.

Potřebovali bychom tedy umět do heše přidávat nové hodnoty, najít hodnotu pro zadaný klíč a případně také umět z heše nějakou hodnotu smazat.

Samozřejmě používat jako klíč libovolný typ, o kterém nic nevíme (speciálně ani to, co znamená, že dva objekty toho typu jsou stejné), dost dobře nejde. Proto potřebujeme

ještě *hešovací funkci* – funkci, která objektu přiřadí nějaké malé přirozené číslo $0 \leq x < K$, kde K je velikost heše (ta by měla odpovídat počtu objektů N, které v ní chceme uchovávat; v praxi bývá rozumné udělat si heš o velikosti zhruba $K = 2N$). Dále popsaný postup funguje pro libovolnou takovou funkci, nicméně aby také fungoval rychle, je potřeba, aby hešovací funkce byla dobře zvolena. K tomu, co to znamená, si něco řekneme níže, prozatím nám bude stačit představa, že tato funkce by měla rozdělovat klíče zhruba rovnoměrně, tedy že pravděpodobnost, že dvěma klíčům přiřadí stejnou hodnotu, by měla být zhruba $1/K$.

Ideální případ by nastal, kdyby se nám podařilo nalézt funkci, která by každým dvěma klíčům přiřazovala různou hodnotu (i to se může podařit, pokud množinu klíčů, které v heši budou, známe dopředu – viz třeba příklad s rozpoznáváním klíčových slov v překladačích). Pak nám stačí použít jednoduché pole velikosti K, jehož prvky budou obsahovat jednak hodnotu klíče, jednak jemu přiřazená data:

```
struct položka_heše
{
  int obsazeno;
  typ_klíče klíč;
  typ_hodnoty hodnota;
} heš[K];
```

A operace naprogramujeme zřejmým způsobem:

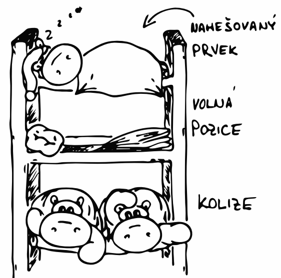
```
void přidej (typ_klíče klíč,typ_hodnoty hodnota)
{
  unsigned index = hešovací_funkce (klíč);

  // Kolize nejsou, čili heš[index].obsazeno=0.
  heš[index].obsazeno = 1;
  heš[index].klíč = klíč;
  heš[index].hodnota = hodnota;
}

int najdi (typ_klíče klíč, typ_hodnoty *hodnota)
{
  unsigned index = hešovací_funkce (klíč);

  // Nic tu není nebo je tu něco jiného.
  if (!heš[index].obsazeno ||
      !stejný (klíč, heš[index].hodnota))
    return 0;

  // Našel jsem.
  *hodnota = heš[index].hodnota;
  return 1;
}
```

Normálně samozřejmě takové štěstí mít nebudeme a vyskytnou se klíče, jimž hešovací funkce přiřadí stejnou hodnotu (říká se, že nastala *kolize*). Co potom?

Jedno z řešení je založit si pro každou hodnotu hešovací funkce seznam, do kterého si uložíme všechny prvky s touto hodnotou. Funkce pro vkládání pak bude v případě kolize přidávat do seznamu, vyhledávací funkce si vždy spočítá hodnotu hešovací funkce a projde celý seznam pro tuto hodnotu. Tomu se říká *hešování se separovanými řetězci.*

Jiná možnost je v případě kolize uložit kolidující hodnotu na první následující volné místo v poli (cyklicky, tj. dojdeme-li ke konci pole, pokračujeme na začátku). Samozřejmě pak musíme i příslušně upravit hledání – snadno si rozmyslíme, že musíme projít všechny položky od pozice, kterou nám poradí hešovací funkce, až po první nepoužitou položku. Tento přístup se obvykle nazývá *hešování se srůstajícími řetězci* (protože seznamy hodnot odpovídající různým hodnotám hešovací funkce se nám mohou spojit). Implementace pak vypadá takto:

```
void přidej (typ_klíče klíč,typ_hodnoty hodnota)
{
  unsigned index = hešovací_funkce (klíč);

  while (heš[index].obsazeno)
  {
    index++;
    if (index == K)
      index = 0;
  }

  heš[index].obsazeno = 1;
  heš[index].klíč = klíč;
  heš[index].hodnota = hodnota;
}

int najdi (typ_klíče klíč, typ_hodnoty *hodnota)
{
  unsigned index = hešovací_funkce (klíč);

  while (heš[index].obsazeno)
  {
    if (stejný (klíč, heš[index].klíč))
    {
      *hodnota = heš[index].hodnota;
      return 1;
    }

    // Něco tu je, ale ne to, co hledám.
    index++;
    if (index == K)
      index = 0;
  }

  // Nic tu není.
  return 0;
}
```



Jaká je časová složitost tohoto postupu? V nejhorším případě bude mít všech N objektů stejnou hodnotu hešovací funkce. Pak hledání může přeskakovat postupně všechny, čili složitost v nejhorším případě může být až $O(NT + H)$, kde T je čas pro porovnání dvou klíčů a H je čas na spočtení hešovací funkce. Laicky řečeno, pro nalezení jednoho prvku budeme muset projít celý heš (v lineárním čase).

Nicméně tohle se nám obvykle nestane – pokud velikost pole bude dost velká (alespoň dvojnásobek prvků heše) a zvolili jsme dobrou hešovací funkci, pak v průměrném případě bude potřeba udělat pouze konstantně mnoho porovnání, tj. časová složitost hledání i přidávání bude jen $O(T + H)$. A budeme-li schopni prvky hešovat i porovnávat v konstantním čase (což například pro čísla není problém), získáme konstantní časovou složitost obou operací.

Mazání prvků může působit menší problémy (rozmyslete si, proč nelze prostě nastavit u mazaného prvku „obsazeno“ na 0). Pokud to potřebujeme dělat, buď musíme použít separované řetězce (což se může hodit i z jiných důvodů,