Milí řešitelé a řešitelky!

Jako naši otcové a dědové, i my vám přinášíme řešení čtvrté série. Pátá série je rovněž uzavřená a opravujeme ji, seč můžeme, takže si užívejte začátek prázdnin a těšte se na soustředění!

Když už si vaše nohy tak pěkně lebedí nahoře, můžete se vyjádřit k uplynulé sérii, zkritizovat těžké úlohy, pochválit pilné organizátory a mudrovat nad tím, co bychom pro vás ještě mohli učinit, abyste se v KSPéčku cítili jako doma.

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

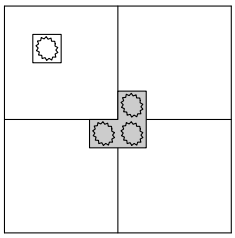
Všechny vaše komunikační touhy si můžete splnit na fóru *http://ksp.mff.cuni.cz/forum/* a záludné dotazy organizátorům lze zasílat e-mailem na adresu *ksp@mff.cuni.cz.*

Vzorová řešení čtvrté série dvacátého prvního ročníku KSP

21-4-1 Dláždění šachovnice

„Právě jste splnili kvalifikační test na soutěž o Nejlepšího crackera roku 2009!“ Podobnou zprávu by obdržela většina řešitelů této úlohy. Naopak co místy pokulhávalo byl popis řešení. Pojďme se tedy na celý problém šachovnice podívat pořádně.

Každou šachovnici o rozměru 2^N lze pokrýt L-ky bez jednoho libovolného políčka a budeme to dokazovat matematickou indukcí. Pro $N = 1$ máme šachovnici 2×2 a tu snadno vyplním jedním L-kem. Pro $N > 1$ celou šachovnici rozdělíme na čtvrtiny (o straně 2^{N-1}) a rádi bychom využili indukčního předpokladu. V jedné čtvrtině se nachází chybějící políčko a tuto čtvrtinu umíme z předpokladu celou vyplnit L-ky. Zbývá nám ještě vyplnit zbývající tři čtvrtiny. Abychom mohli nasadit indukční předpoklad, musíme z každé čtvrtiny vybrat jedno políčko, které nevyplníme. Pokud vybereme tři rohová políčka, jak je naznačeno na obrázku, jsme je schopni zaplnit jedním L-kem. Zbytek každé čtvrtiny potom umíme vyplnit z indukčního předpokladu. Tím jsme nalezli vyplnění celé šachovnice o velikosti 2^N a důkaz je dokončen.



Důkaz nám říká, jak vytvořit rekurzivní algoritmus, který bude vyplnění přímo hledat. Stačilo by napsat rekurzivní funkci, která by dostala velikost šachovnice a pozici vykousnutého políčka a vrátila pokrytí L-kami. Časová složitost algoritmu je počet políček šachovnice, tedy $\mathcal{O}(2^{2N})$ a rychleji to ani nejde, protože pozici přesně tolika L-ek musíme popsat. Psát však do řešení přímo program nebylo zapotřebí.

Pavel Klavík

21-4-2 Dosah kouzla

Úloha je zcela zřejmá. Prostě ke každému vrcholu projdu všechny ostatní a spočítám vzdálenosti. Z nich vybrat největší je cvičení první hodiny programování. Časová složitost je zcela očividně $\mathcal{O}(n^2)$.

Ale moment, 9 bodů za dva jednoduché cykly v sobě? To by se KSP dopustilo urážky na cti všech řešitelů. To půjde lépe.

Toto řešení by určitě fungovalo na libovolnou množinu bodů. Tak proč je zadaný sklep konvexní? Všimneme si, že když se posadíme do jednoho vybraného rohu k Mírielovi, od nás doleva vyběhne krysa a poběží po obvodu, tak se bude napřed stále vzdalovat, než doběhne do nejvzdálenějšího rohu, a potom už se zase bude pouze přibližovat. Jinými slovy, posloupnost vrcholů je v první části rostoucí a v druhé klesající.

Jak takové věci využít? Mohli bychom použít něco, co až nápadně připomíná binární vyhledávání v setříděném poli. Máme nějaký interval (na začátku je to celá posloupnost vrcholů n-úhelníku bez jednoho, ve kterém sedíme). Vybereme si vždy prostřední, tím interval rozdělíme na dvě části. Z každé části si vybereme jeden vzorek a vezmeme tu, kde vyjde vzdálenost větší (v té se bude nacházet největší). Pokud se nám vzdálenosti vzorků rovnají, pak interval nemůžeme rozpůlit podle prostředního, ale určitě bude největší mezi nimi. Pokud budeme brát vzorky ve vzdálenosti $\frac{1}{4}$ od krajů, pak interval také zmenšíme na polovinu.

Pokud takto nalezneme nejvzdálenější vrcholy pro každý vrchol, zlepšíme si časovou složitost na $\mathcal{O}(n \cdot \log n)$. S tím se ale ještě nespokojíme.

Když máme spočítaný nejvzdálenější vrchol od vrcholu, ve kterém právě sedíme a přesedneme si o jeden vrchol po směru hodinových ručiček (trochu obtížná představa na dobu, kdy byly hodiny přesýpací), co se stane s nejvzdálenějším vrcholem? Buď bude stále na tom samém místě, nebo se posune také po směru hodinových ručiček (nevíme ale, jak daleko). A vida, to přímo nabádá k tomu na tom založit algoritmus.

Na začátku tedy nějak seženeme nejvzdálenější vrchol k prvnímu vrcholu. Poté si $(n-1)\times$ „přesedneme“. Tím jakoby jednou oběhneme celé sklepení. Přestože nevíme, jak daleko se při každém přesunu pohne nejvzdálenější vrchol, po celém kolečku určitě skončí na stejném místě a nikdy nás nepředběhne (ty by nás musel dohnat a vrchol sám sobě nejvzdálenější být nemůže). Tedy, oběhne celé sklepení také právě jednou. Toto obíhání má tedy složitost $\mathcal{O}(n)$. Pokud stihneme spočítat nejvzdálenější vrchol v čase $\mathcal{O}(n)$, tak jsme vyhráli. Rychleji to už určitě nepůjde, načtení vstupu nám zabere alespoň takovéto množství času.

Co se týče paměťové složitosti, stačí nám pamatovat si jednotlivé vrcholy.

Ještě si lze všimnout, že si při porovnávání můžeme zcela odpustit odmocňování, neboť odmocnina z kladných čísel je rostoucí a tedy $\sqrt{a} < \sqrt{b} \Leftrightarrow a < b$.

Michal „vorner“ Vaner

21-4-3 Stavění robota

Netrpaslická nebyla jen velikost Boendalova čísla, ale i obtížnost této úlohy. Provolejme třikrát sláva těm řešitelům,

kteří se namočení do algebry nezalekli, a pojďme si osvětlit řešení.

Jak si jistě pamatujete, cílem bylo spočítat kombinační číslo $\binom{N}{K}$, respektive jeho zbytek po dělení číslem R. Standardní metody na počítání kombinačních čísel zde selhávají – konstrukce dynamickým programováním by trvala příliš dlouho a rekurentní definice kombinačních čísel by potřebovala dlouhá čísla, která v povolených jazycích k dispozici nemáme. Je tedy třeba zkusit využít dělení modulo, které bychom tak jako tak museli provést.

Vezměme si hledané kombinační číslo $\binom{N}{K}$. Možná už ze školy víte, že kombinační číslo se dá také zapsat jako $H/K!$, kde H je součin K klesajících čísel od N do $N-K+1$. Řečeno jazykem matematikovým: Pokud

$$H = N \cdot (N-1) \cdot (N-2) \cdot \ldots \cdot (N-K+1),$$

pak

$$\binom{N}{K} = H/K!$$

Tento zápis je pro kombinační čísla přirozenější, neboť vystihuje lépe jejich podstatu – kombinační číslo je počet K-tic, které můžeme vybrat z N prvků, pokud neuvažujeme opakování. A přesně tak počítá i vzorec $H/K!$. Nejprve vybereme první prvek K-tice, na to máme N možností, pak zvolíme druhý, to máme $N-1$ možností (ten první znovu vybrat nemůžeme) a tak dále, až vybereme poslední, K-tý, na něj máme $N-K+1$ možností. Nicméně tímto výběrem jsme si porušili druhé pravidlo – některé K-tice se nám tam opakují. My ale víme, které a kolikrát. Tímto taháním prvků jsme vybrali každou K-tici právě tolikrát, kolik je zpřeházení (permutací) K prvků. Například pokud vybíráme trojice z prvků $\{1 \ldots 6\}$, pak trojici $(1,2,3)$ jsme vybrali ještě jako trojice $(1,3,2), (2,1,3), (2,3,1), (3,1,2), (3,2,1)$. Abychom všechny tyto případy zahodili, postačí nám celé číslo H vydělit počtem permutací na K prvků, a jak všichni ví, těch je $K!$.

Nám se tenhle zápis bude hodit, protože dělení modulo se chová hezky pří sčítání a násobení. Bota nás bude tlačit jen tehdy, když budeme chtít spočíst $H/K!$, neboť celočíselné dělení při operacích se zbytky k dispozici nemáme. Budeme si jej muset „odsimulovat“.

Půjdeme na to přes prvočíselné reprezentace, kterých se každý nabažil na základní škole – rozložíme zadane kombinacni číslo na mocniny prvočísel. Začneme zlehka, rozkladem součinu H. Nejprve si najdeme Eratosthenovým sítem všechna prvočísla od 1 do N. U čísel složených v tomto rozsahu si ještě poznačíme, jaké největší prvočíslo je dělí (to nám síto udělá takřka samo).

S prvočísly po ruce již můžeme rozkládat. Budeme si udržovat aktuální rozklad čísla (který nemusí být prvočíselný, například $360 = 72 \cdot 5$), setříděný podle základů. Tento rozklad si budeme pamatovat v poli, kde hodnota Z-tého prvku je rovna exponentu E takovému, že Z^E je přítomno v našem rozkladu. Tento seznam rozkladů budeme postupně drolit na menší kousky.

Vezměme největší neprobraný základ Z a odpovídající exponent E. Je-li Z číslo složené, pak se podíváme do síta a najdeme největší prvočíslo P, které Z dělí. Tím se nám rozloží Z^E na $P^E \cdot (Z/P)^E$. Obě čísla jsou menší než Z, přidáme je tedy s odpovídajícím exponentem do rozkladu a pokračujeme dále. Je-li Z prvočíslo, pak mohu tento základ a exponent vzít jako hotový (a přejdu k menšímu základu).

Pro náš příklad bude rozklad probíhat takto:

$$360 = 72^1 \cdot 5^1 = 24^1 \cdot 5^1 \cdot 3^1 = 8^1 \cdot 5^1 \cdot 3^2 = \\ = 5^1 \cdot 4^1 \cdot 3^2 \cdot 2^1 = 5^1 \cdot 3^2 \cdot 2^3.$$

Použití těchto neúplných rozkladů se nám velice hodí – algoritmus totiž může rozložit libovolné číslo zadané součinem, a přesně takové je H, které rozložit potřebujeme. Nyní přišel čas vyřešit dělení číslem $K!$. To můžeme udělat chytrým trikem – už dopředu víme, že výsledek bude celočíselný. Začněme tedy s následujícím rozkladem:

$$\binom{N}{K} = N^1 \cdot (N-1)^1 \cdot \ldots \cdot (N-K+1)^1 \cdot K^{-1} \cdot (K-1)^{-1} \cdot \ldots \cdot 2^{-1}$$

Rozklad je to korektní a algoritmus nám určitě neporuší, neboť jakmile budeme rozkládat čísla menší než K, už jsme rozložili všechna větší než K, a tedy (díky celočíselnosti výsledku) nikde nemůžeme narazit na záporný exponent.

Máme rozloženo, co dál? Teď si trochu započítáme. Jak jsme řekli už výše, při operacích modulo můžeme používat sčítání a násobení, jak jsme zvyklí. Tedy pokud chceme spočíst $\binom{N}{K}$ a už známe jeho prvočíselný rozklad, budeme dělit modulo ne celé číslo, ale jednotlivé mocniny prvočísel, a mezivýsledky vynásobíme dohromady. Na konci ještě nesmíme zapomenout celý součin vydělit modulo R, protože během násobení se nám mohlo stát, že jsme z modulo R utekli.

Matematik by to napsal takto: Pokud si zapíšeme prvočíselný rozklad $\binom{N}{K}$ jako ${p_1}^{l_1} \cdot {p_2}^{l_2} \cdot \ldots \cdot {p_k}^{l_k}$, pak platí:

$$\binom{N}{K} (\bmod R) = {p_1}^{l_1} \cdot {p_2}^{l_2} \cdot \ldots \cdot {p_k}^{l_k} (\bmod R) = \\ = ({p_1}^{l_1} (\bmod R)) \cdot ({p_2}^{l_2} (\bmod R)) \cdot \ldots \\ \ldots \cdot ({p_k}^{l_k} (\bmod R)) (\bmod R)$$

Fajn, přešli jsme kopec, přeplavali řeku a nyní nám zbývá se poprat s tím, máme-li velkou mocninu prvočísla p^l (ještě stále se nám celý výsledek nevejde do integeru), jak rychle spočítat její zbytek modulo R. Naštěstí víme, že exponent l našeho prvočísla bude relativně malý (to, že exponent závisí nejvýše lineárně na N, můžeme vypozorovat například z rovnice $\sum_{k=0}^{N} \binom{N}{k} = 2^N$). Můžeme si tedy dovolit například přepočítání v $\mathcal{O}(log_2 N)$. To uděláme tak, že rozložíme l na součet mocnin dvojky, kde se každá vyskytuje nejvýše jednou (rozmyslete si, že každé číslo lze rozložit na takový součet). Nyní máme číslo ve tvaru $p^{2^a+2^b+2^c \ldots 2^z}$, kde $0 \leq a < b < cd \ldots < z \leq log_2 P$. To si podle starých dobrých aritmetických pravidel přepíšeme na $p^{2^a} \cdot p^{2^b} \cdots p^{2^z}$ a můžeme přistoupit k počítání. Jako obvykle u toho využijeme faktu, že zbytek součinu modulo dvou čísel je roven součinu zbytků jednotlivých čísel.

Začneme od 0, $p^1 mod R$ uložíme jako základ. Pak v každém kroku přistoupíme k další mocnině. Nejprve si ověříme, jestli tato mocnina zrovna v našem rozkladu figuruje, a pokud ano, číslo, kterým máme přinásobit výsledek, vypočítáme snadno z předchozího: $p^{2^a} = p^{2^{a-1}} \cdot p^{2^{a-1}}$.

A tím jsme hotovi! Ukázali jsme si, jak spočítat modulo mocninu velkého prvočísla, tyto výsledky vynásobíme dohromady pro každé prvočíslo, které se účastnilo rozkladu kombinačního čísla a celkový součin vypíšeme na výstup (nezapomeneme v průběhu dělit modulo R, aby nám výsledek nepřetekl).

Zbývá prohodit pár slov o časové a paměťové složitosti. Na rozklad na prvočísla jsme potřebovali dvě pole o N prvcích, zbytek se dá zvládnout jen s pár integery. Paměťové

nároky jsou tedy $\mathcal{O}(N)$. Co se týče nároků časových, bylo by třeba spočítat, jak rychle proběhlo hledání prvočísel Eratosthenovým sítem. To zde nebudeme dokazovat, pouze poznamenejme že je to $\mathcal{O}(N \log\log N)$. Co se týče druhé části algoritmu, bude asymptoticky alespoň $\mathcal{O}(N \log N)$. Na optimalitu si nároky neděláme, neboť u teorie čísel skoro vždy platí, že můžeme nasadit lepší síto a dosáhneme asymptoticky lepšího výpočtu.

Dík patří Zbyňkovi Faltů za vzorové řešení!

Martin Böhm & Martin Kruliš & CodEx

21-4-4 Heslo

K dané permutaci s prvky, které se mohou opakovat, lze nalézt následující v lineárním čase s její délkou (stačí najít nejdelší nerostoucí konec, cifru před ním prohodit s nejbližší vyšší z onoho konce a ten pak obrátit); K-násobné zopakování tohoto postupu vede k řešení úlohy, leč v čase $\mathcal{O}(KN)$, přičemž K by mohlo být poměrně velké i pro malá N, takže se pokusme o rychlejší algoritmus.

Kdybychom chtěli najít K-tou permutaci v lexikografickém uspořádání od začátku (tedy ne od nějaké dané na vstupu), vezmeme si nejprve tu nejnižší (cifry uspořádané vzestupně) a podíváme se, za jak dlouho se změní první cifra. Zřejmě se před tím musí vystřídat všechny permutace, které na ni začínají, a těch je tolik, kolik je permutací zbývajících cifer (označme tento počet P). $(P+1)$-ní permutace tedy vypadá tak, že na prvním místě je druhá nejmenší cifra a zbytek je opět setříděný vzestupně (rozmyslete si, že od první permutace se liší jen prohozením dvou cifer).

Pokud je $K \leq P$, cifru na prvním místě měnit nepotřebujeme a jen zopakujeme týž postup pro permutaci zbylých čísel. V opačném případě zaměníme první cifru s nejbližší vyšší, od K odečteme P, jakožto počet permutací, které jsme přeskočili, a opět můžeme postupovat stejně (podívat se, jestli dalším změnou na prvním místě K „nepřeskočíme“, a buď příslušné prohození udělat, nebo jít na další pozici) dokud K nesnížíme na 0.

Popsaný postup však funguje, jen když máme za cifrou, s níž právě pracujeme, cifry seřazené. Všimněme si ale, že stejně lze cifry i snižovat: pokud máme permutaci začínající na nějakou cifru, za níž jsou cifry již uspořádané, můžeme tu první prohodit s nejbližší nižší, čímž se dostaneme zpět o počet permutací na těchto cifrách bez té, kterou jsme dali na začátek.

Půjdeme tedy odzadu a cifru na každé pozici budeme postupně vyměňovat za menší až na tu nejnižší s tím, že cifry za ní jsou vždy seřazené (na začátku – pro pozici na konci – to platí triviálně a po skončení úprav tuto podmínku zřejmě rozšíříme i na aktuální pozici); zároveň budeme příslušně zvyšovat K. Ve chvíli, kdy K bude menší než počet permutací od aktuální pozice do konce, znamená to, že cifry nalevo měnit už nepotřebujeme a stačí jen najít K-tou permutaci na cifrách, které jsme prošli.

Abychom vždy nejbližší vyšší cifru k té aktuální našli rychle, místo „přeskládaného“ konce si budeme pamatovat počty jednotlivých cifer, které jsme viděli, což také stačí k tomu, abychom věděli, jak má vypadat.

Zbývá si už jen rozmyslet, jak zjišťovat počty permutací. Jelikož se cifry mohou opakovat, musíme $N!$ ještě vydělit faktoriály počtů jednotlivých cifer. Počítat to pokaždé celé by stálo příliš času, ale naštěstí vždy potřebujeme jen drobnou úpravu, pokud do permutace délky n přidáváme nějakou cifru c, počet permutací stačí vynásobit $(n+1)$ a vydělit novým počtem cifer c; analogicky pak pro odebírání a vyměňování cifer.

Kromě načtení vstupu projdeme celou permutaci nejvýše dvakrát a pro každou pozici potřebujeme jen konstantní čas, takže časová složitost je $\mathcal{O}(N)$, stejně tak i paměťová. To ovšem za předpokladu, že čísla, se kterými budeme pracovat, budou dostatečně malá na to, abychom čas spotřebovaný na jednotlivé operace mohli za konstantní považovat. Nicméně, jak bylo naznačeno v úvodu, počty permutací mohou být poměrně velké; v případě permutací deseti cifer by jich mohlo být $\mathcal{O}(10^N)$ a na potřebné výpočty s takovýmito čísly by byl nutný čas $\mathcal{O}(\log 10^N) = \mathcal{O}(N)$, což by vedlo na celkovou časovou složitost $\mathcal{O}(N^2)$.

Roman Smrž

21-4-5 Znaky

Úlohu vyřešíme jednoduchým algoritmem: v každém kroku přečteme jeden znak ze vstupu, převedeme ho do nějaké vhodné reprezentace a postupně ho porovnáme se všemi dosud přečtenými jedinečnými znaky. Pokud se neshoduje s žádným z nich, přidáme ho do seznamu. Protože nám v této úloze jde hlavně o paměťovou náročnost, soustředíme se na to, jak reprezentovat znaky v paměti co nejúsporněji.

Vzhledem k tomu, že v každém řádku znaku je vybarvený právě jeden pixel, nabízí se řešení ukládat znak jako posloupnost N čísel s hodnotami $1 \ldots N$, kde i. číslo udává, v kolikátém sloupci i. řádku je vybarvený pixel. Kolik bitů bude jeden takovýto znak zabírat? Na každý řádek potřebujeme $\lceil \log_2 N \rceil$ b, na celý znak tedy $N \cdot \lceil \log_2 N \rceil$ b. Abychom takovéto velikosti opravdu dosáhli, nemůžeme ukládat každý řádek do samostatného prvku pole, ale budeme znak v této reprezentaci chápat jako posloupnost bitů, které rozdělíme po osmi a uložíme do pole bytů. Na konci pole pak zbude až sedm nevyužitých bitů, ale s tím už nic neuděláme. Časová složitost tohoto řešení je $\mathcal{O}(K \cdot (N^2 + KN))$, protože každý z K znaků nejdříve načtu v čase $\mathcal{O}(N^2)$ a pak porovnám s až K přečtenými znaky řádek po řádku. Lepší časové složitosti by šlo dosáhnout třeba použitím binárního vyhledávání, ale o čas nám tentokrát moc nejde.

K dosažení menší paměťové náročnosti můžeme využít toho, že každé číslo sloupce se v zápisu znaku vyskytuje jen jednou. Hodnota pro každý řádek tak nebude určovat, v kolikátém sloupci ze všech možných je vybarvený pixel, ale budeme brát v úvahu jen povolené sloupce, tedy ty, ve kterých ještě není žádný vybarvený pixel. Když zapíšu každý řádek jako dvojici číslo sloupce / maximální číslo sloupce, tak znak s $N = 5$ a zápisem 2/5, 5/5, 3/5, 1/5, 4/5 (zabírající 15 b) můžu s využitím výše zmíněného postupu zapsat jako 2/5, 4/4, 2/3, 1/2, 1/1 (8 b). Je vidět, že na uložení předposledního řádku stačí jeden bit a pro poslední řádek je to dokonce nula bitů. Na zapsání jednoho znaku tak potřebujeme $\sum_{i=0}^{N-1} \lceil \log_2(N-i) \rceil$ b $= \sum_{i=1}^{N} \lceil \log_2 i \rceil$ b.

Zatím jsme ještě nevyužili toho, že na každé diagonále je vybarvený nejvýše jeden pixel. Použijeme stejný postup jako v předchozím případě, jenom pro každý řádek budou povolené ty pixely, pro které sloupec a obě diagonály, na kterých leží, neobsahují zatím žádný pixel. Výše uvedený znak tak můžeme přepsat jako 2/5, 2/2, 2/2, 1/1, 1/1 (5 b). Celková velikost takto zapsaného znaku může být různá i pro dva znaky se stejným N. Časová složitost těchto dvou řešení, využívajících povolené sloupce, je $\mathcal{O}(K \cdot (N^2 + KN^2)) = \mathcal{O}(K^2 N^2)$. Zhoršení oproti první variantě je způsobené tím,

že při porovnávání znaků musíme navíc pro každý řádek znaku ze seznamu už přečtených projít seznamy povolených sloupců, abychom zjistili, kolik bitů máme přečíst při čtení následujícího řádku.

Další možností, jak ještě snížit paměťové nároky je použití trie, ale my zkusíme něco jiného: řešení, které má ještě menší paměťové nároky, ale na druhou stranu zase velkou časovou složitost. Jeho myšlenka je velice jednoduchá: všechny možné znaky si očíslujeme a pro každý znak si budeme pamatovat jen jeho číslo. Pokud je možných znaků M, tak každý z nich bude zabírat $\lceil \log_2 M \rceil$ b. Jeden znak ani do méně bitů zapsat nejde, protože pak by možných zápisů bylo méně než různých znaků. Když teď do seznamu přečtených jedinečných znaků ukládáme přímo čísla, mohli bychom, podobně jak jsme to už dělali, počítat s tím, že číslo, které se v seznamu už vyskytlo, tam znovu nebude. Jde to ale líp. Tentokrát nám totiž nezáleží na pořadí a tak si můžeme tento seznam udržovat setříděný. Když v seznamu najdu číslo x, tak vím, že za ním musí následovat číslo mezi $x+1$ a M, takže na jeho uložení bude stačit $\lceil \log_2(M-x) \rceil$ b. Jinou variantou, jak si přečtené znaky ukládat je posloupnost M bitů, kde i. bit určuje, jestli jsem už přečetl znak s číslem i. Rozumným kompromisem mezi těmito dvěma řešeními je použít ze začátku seznam znaků a jakmile jeho velikost překročí M bitů, přejít na druhou variantu. Jak ale vlastně určíme číslo znaku, který dostaneme na vstupu? Snadno, procházíme postupně všechny permutace N čísel a pokud některá z nich vyhovuje definici znaku, započítáme ji. Pokud narazíme na tu z nich, která je shodná se zpracovávaným znakem, známe jeho číslo. Ještě před přečtením prvního znaku ze vstupu ale musíme podobným způsobem projít všechny znaky, abychom věděli, kolik jich je celkem a kolik bitů tak budeme potřebovat na jeden znak. A nakonec časová složitost: $\mathcal{O}(NN!+K(NN!+N!)) = \mathcal{O}(KNN!)$. Na začátku totiž spočítáme včechny možné znaky a pak pro každý znak na vstupu určíme jeho číslo a v případě první varianty jej porovnáme s už přečtenými unikátními znaky, kterých nemůže být víc než $N!$. Takto velká časová složitost samozřejmě způsobí praktickou nepoužitelnost popsaného algoritmu už i pro relativně malá N.

Petr Onderka

21-4-6 Nejtěžší číslo

Začneme podúlohou **b)**, neboť nejtěžší číslo bylo nakonec překvapivě lehké :-) Stačí si totiž vzpomenout na trik z řešení Pana Cowmesse (úloha 21-3-6): pomocí `x@x` „sklepeme“ všechny jedničky v čísle směrem k nejnižšímu řádu. Pak si ještě všimneme toho, že číslo `a` je těžší než číslo `b` právě tehdy, když `a@a` je větší než `b@b`, a řešení je nasnadě:

```
sem:    a = A[n] @ A[n]
        if a<=m => goto tam
        m = a
        y = A[n]
tam:    n = n-1
        if n>0 => goto sem
```

Program pro n čísel použije $4n + 2m$ instrukcí, kde m říká, kolikrát se během procházení pole zvýší váha čísla. To se může stát nejvýše 32-krát, takže můžeme počet instrukcí odhadnout shora výrazem $4n + 64$.

Toto řešení můžeme ještě zrychlit (díky, Jitko!), když si uvědomíme, že si místo hodnoty zatím nejtěžšího čísla a jeho sklepané verze stačí pamatovat pozici takového čísla ve vstupu:

```
sem:    B[n] = A[n] @ A[n]
        if B[n]<=B[m] => goto tam
        m = n
tam:    n = n-1
        if n>0 => goto sem
        y = B[m]
```

Tak počet provedených instrukcí snížíme na $4n + m + 1 \leq 4n + 33$.

Podúloha **a)**, tedy zvážení jednoho čísla, byla naopak lehce překvapivá :-) Nejrychlejší známý program má totiž přibližně $2 \cdot 10^9$ instrukcí, z nichž se ovšem pro každý vstup provede nejvýše pět.

Jak funguje? Nejprve si zadané číslo sklepeme a pak rozebereme 33 případů, které mohou nastat. To bychom přímočaře zvládli 32 podmínkami nebo pomocí půlení intervalu 6 podmínkami. Ale my raději nepoužijeme podmínku žádnou: Využijeme toho, že parametr instrukce skoku může být nejen konkrétní číslo (či návěští), ale také obsah registru, a jedním skokem se přesuneme na správné místo v programu, které pouze vrátí výsledek. Vypadat to bude takto (v závorkách jsou napsány adresy instrukcí):

```
(0)       z = x@x
(1)       z = z+4
(2)       jump z
(3)       w = 32          // při z=2^32-1
(4)       jump hotovo     // při z=0
(5)       w = 1           // při z=1
(6)       jump hotovo
(7)       w = 2           // při z=3
(8)       jump hotovo
          ...
(11)      w = 3           // při z=7
          jump hotovo
          ... ...
(2^31+3)  w = 31          // při z=2^31-1
hotovo:
```

(Všimněte si, že `x@x` stále potřebujeme, protože kdybychom chtěli rozebrat úplně všechny případy, nevešel by se program do 2^{32} instrukcí, které dovedeme adresovat.)

Jako zákusek přidáváme ještě řešení, které si vystačí s 12 instrukcemi bez jediného skoku. Konstanty začínající `0b` jsou dvojkové. Zkuste přijít na to, proč funguje:

```
y = x & 0b10101010101010101010101010101010
x = x & 0b01010101010101010101010101010101
y = y >> 1
x = x + y
y = x & 0b00110000110000110000110000110000
z = x & 0b00001100001100001100001100001100
x = x & 0b11000011000011000011000011000011
y = y >> 4
z = z >> 2
x = x + y
x = x + z
w = x % 63
```

Martin Mareš & Milan Straka

Úloha 21-3-2 – Dosah kouzla – program

```
program kouzlo;

const max = 1000;
{ Sklep s více vrcholy už je prakticky kruhový }

type
    tcoord = record
        x, y: real;
    end;
    tcorners = array[0..max-1] of tcoord;

var
    n: integer;
    corners: tcorners;
    f: text;
    best, bestLoc: real;
    bestPos, bestPair1, bestPair2: integer;
    i: integer;

function dist2(i, j: integer): real;
begin
    dist2 := (corners[i].x - corners[j].x) * (corners[i].x
    - corners[j].x) + (corners[i].y - corners[j].y)
    * (corners[i].y - corners[j].y);
end;

begin
    { Trocha načítání }
    assign(f, 'kouzlo.in');
    reset(f);
    n := 0;
    while not seekeof(f) do begin
        read(f, corners[n].x, corners[n].y);
        inc(n);
    end;
    close(f);
    { Zatím nic nenalezeno }
    bestPos := 1;
    best := 0;
    { Ke všem najdeme ten nejlepší}
    for i := 0 to n - 1 do begin
        bestLoc := dist2(i, bestPos mod n);
        { Posouváme tak dlouho, dokud se to zlepšuje }
        while dist2(i, (bestPos + 1) mod n) > bestLoc do
        begin inc(bestPos);
            bestLoc := dist2(i, bestPos mod n);
        end;
        { Zlepšilo se celkově? }
        if bestLoc > best then begin
            bestPair1 := i;
            bestPair2 := bestPos;
            best := bestLoc;
        end;
    end;
    { Vypsat }
    writeln('[', corners[bestPair1].x, ',',
        corners[bestPair1].y, '], [', corners[bestPair2].x,
        ',', corners[bestPair2].y, ']');
end.
```

Úloha 21-3-3 – Stavění robota – program

```
program robot;

var
    sito     : array [ 1..1000000 ] of longint;
    N, K, M  : longint;
    i, j     : longint;
    vysledek : longint;
    {rozklad[i] odpovídá největšímu
     exponentu z takovému, že i^z dělí (N nad K).}

    rozklad    : array [ 1..1000000 ] of longint;
    mocnina  : longint;

begin
    readln(N,K,M);

    for i:=2 to N do begin
        sito[i]:=0;
        rozklad[i]:=0;
    end;

    {Jednoduché zrychlení.}
    if K > N-K then
        K:=N-K;

    {Eratostenovo síto, které navíc počítá
     největší prvočíslo dělící každé složené
     číslo.}
    for i:=2 to N do begin
        if sito[i]=0 then begin
            j:=i*2;
            while j<=N do begin
                sito[j]:=i;
                j:=j+i;
            end;
        end;
    end;

    {Kombinačníčíslo je součin čísel od
     N-K+1 do N ...}
    for i:=N-K+1 to N do
        inc(rozklad[i]);

    {...dělen k! }
    for i:=2 to K do
        dec(rozklad[i]);

    vysledek:=1;
    for i:=N downto 2 do begin
        if sito[i]=0 then begin
            mocnina:=i mod M;
            while rozklad[i]<>0 do begin
                {Nejvyšší číslo rozkladu je prvočíslo,}
                {zpracujeme je. Rozložíme exponent na
                součet mocnin dvojky a ty spočítáme
                prostým mocněním modulo M.}

                if rozklad[i] mod 2 = 1 then
                    vysledek:=(vysledek*mocnina) mod M;
                {Zkus vyššíninu.}
                mocnina:=(mocnina*mocnina) mod M;
                rozklad[i]:=rozklad[i] div 2;
            end;
        end else begin
            {Nejvyšší číslo rozkladu je číslo složené.}
            rozklad[sito[i]]:=rozklad[sito[i]]+rozklad[i];
            rozklad[i div sito[i]]:=rozklad[i div sito[i]]
                +rozklad[i];
        end;
    end;

    writeln(vysledek);
end.
```

Úloha 21-3-4 – Heslo – program

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

```
int n, k;
int heslo[N_MAX];       // celá permutace
int pocty[10];          // počty jednotlivých cifer v podpermutaci
                        // od aktuální pozice do konce
int poc_perm = 1;       // počet podpermutací, které mohou vytvořit
                        // cifry zaznamenané v poli 'pocty'

int main(void)
{
        scanf("%d%d\n", &n, &k);
        for (int i = 0; i < n; i++) heslo[i] = getchar()-'0';

        int i;
        for (i = n-1; k >= poc_perm; i--) {

                // každou cifru postupně vyměňujeme za menší ...
                for (int j = heslo[i]-1; j >= 0; j--) {
                        if (pocty[j]) {
                                poc_perm *= pocty[j]--;
                                poc_perm /= ++pocty[heslo[i]];

                                k += poc_perm;
                                heslo[i] = j;
                        }
                }

                // ... a nakonec rozšíříme podpermutaci i o aktuální pozici
                poc_perm *= n-i;
                poc_perm /= ++pocty[heslo[i]];
        }

        for (i++; i < n; i++) {
                // najdeme nejnižší dostupnou cifru ...
                heslo[i] = 0;
                while (!pocty[heslo[i]]) heslo[i]++;

                // ... tu odebereme z podpermutace ...
                poc_perm *= pocty[heslo[i]]--;
                poc_perm /= n-i;

                // ... a poté se ji budeme poukoušet postupně zvyšovat
                for (int j = heslo[i]+1; j < 10 && k >= poc_perm; j++) {
                        if (pocty[j]) {
                                k -= poc_perm;
                                poc_perm *= pocty[j]--;
                                poc_perm /= ++pocty[heslo[i]];
                                heslo[i] = j;
                        }
                }
        }

        for (int i = 0; i < n; i++) putchar(heslo[i]+'0');
        putchar('\n');
        return 0;
}
```

Úloha 21-3-5 – Znaky – program

```
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>


struct znak { //znak reprezentovaný polem řádků, pro každý řádek je hodnotou číslo vybrveného sloupce
        unsigned * radky;
};

struct bitove_pole {
        uint8_t * bity;
        unsigned pozice1, pozice2; //aktuální pozice při průchodu, pozice1 určuje byte, pozice2 bit v bytu
        unsigned obsazeno1, obsazeno2; //počet plně obsazených bytů, obsazených bitů v posledním bytu
        unsigned alokovano; //alokovaných bytů
};

unsigned N, K;
struct bitove_pole prectene; //seznam přečtených znaků

unsigned log_2(unsigned x) { //spočítá ceil(log_2(x))
        x = x - 1;
        unsigned result = 0;
```

```
        while (x > 0)
        {
                result++;
                x >>= 1;
        }
        return result;
}

//vrátí číslo zadaného znaku; pokud je zadaný znak větší než největší platný znak, vrátí počet možných znaků
//prochází postupně permutace řádků a porovnává se zadaným znakem
unsigned cislo_znaku(struct znak z) {
        struct znak generovany;
        generovany.radky = malloc(sizeof(&generovany.radky) * N);
        bool * sloupce = calloc(N, sizeof(bool));       //sloupce a diagonály zakázané v aktuálním řádku
        bool * diag1 = calloc(2 * N - 1, sizeof(bool));
        bool * diag2 = calloc(2 * N - 1, sizeof(bool));
        generovany.radky[0] = 0;
        unsigned shodnych_radku = 0;
        unsigned nalezenych_znaku = 0;
        unsigned i = 0;
        bool konec = false;
        while (!konec) {
                unsigned j = generovany.radky[i];
                if (j >= N)     { //prošli jsme celý řádek
                        if (i == 0)
                                konec = true; //prošli jsme všechny znaky
                        else { //vracíme se o řádek výš
                                i--;
                                j = generovany.radky[i];
                                sloupce[j] = false;
                                diag1[i+j] = false;
                                diag2[N+i-j] = false;
                                generovany.radky[i]++;
                        }
                }
                else if (!sloupce[j] && !diag1[i+j] && !diag2[N+i-j])   { //jde umístit pixel
                        if (z.radky[i] == j && i == shodnych_radku)
                                shodnych_radku++;
                        if (i == N - 1) { //poslední řádek
                                if (shodnych_radku == N)
                                        konec = true; //našli jsme shodný znak
                                else { //našli jsme další (neshodný) znak, vracíme se o řádek výš
                                        nalezenych_znaku++;
                                        i--;
                                        j = generovany.radky[i];
                                        sloupce[j] = false;
                                        diag1[i+j] = false;
                                        diag2[N+i-j] = false;
                                        generovany.radky[i]++;
                                }
                        } else { //jdeme o řádek níž
                                sloupce[j] = true;
                                diag1[i+j] = true;
                                diag2[N+i-j] = true;
                                i++;
                                generovany.radky[i] = 0;
                        }
                } else
                        generovany.radky[i]++; //zkusíme další sloupec v řádku
        }
        free(generovany.radky);
        free(sloupce);
        free(diag1);
        free(diag2);
        return nalezenych_znaku;
}

void nacti_znak(struct znak * z) { //do z uloží znak načtený ze vstupu
unsigned x;
  for (unsigned i = 0; i < N; i++)
    for (unsigned j = 0; j < N; j++)
    {
      scanf("%u", &x);
      if (x == 1)
        z->radky[i] = j;
    }
}

//určuje, jestli ještě jsou v bitovém poli p nějaké nepřečtené bity
bool jde_cist_bity(struct bitove_pole * p) {
        return (p->pozice1 < p->obsazeno1) || ((p->pozice1 == p->obsazeno1) && (p->pozice2 < p->obsazeno2));
```

```
}

unsigned cti_bity(struct bitove_pole * p, unsigned n) { //přečte n bitů z bitového pole
        unsigned result = 0;
        unsigned precteno = 0;
        while (precteno < n) {
                if (n - precteno >= 8 - p->pozice2) //ctu do konce bytu
                {
                        result |= (p->bity[p->pozice1] >> p->pozice2) << precteno;
                        precteno += 8 - p->pozice2;
                        p->pozice1++;
                        p->pozice2 = 0;
                } else {
                        result |= ((p->bity[p->pozice1] >> p->pozice2) & ((1 << (n-precteno)) - 1)) << precteno;
                        p->pozice2 += n - precteno;
                        precteno += n - precteno;
                }
        }
        return result;
}

//přečte z bitového pole číslo z {a, ..., b}
unsigned cti_z_intervalu(struct bitove_pole * p, unsigned a, unsigned b) {
        return a + cti_bity(p, log_2(b-a+1));
}

//zapíše na konec bitového pole n bitů uložených v parametru x
void zapis_bity(struct bitove_pole * p, unsigned x, unsigned n) {
        if ((p->alokovano - p->obsazeno1) * 8 - p->obsazeno2 < n) { //musíme zvětšit pole
                p->alokovano = p->alokovano + (n - ((p->alokovano - p->obsazeno1) * 8 - p->obsazeno2) + 7) / 8;
                //+7 kvůli zaokrouhlení nahoru
                p->bity = (uint8_t *)realloc(p->bity, p->alokovano);
        }
        unsigned zapsano = 0;
        while (zapsano < n) {
                if (p->obsazeno2 == 0)
                        p->bity[p->obsazeno1] = 0; //vynuluju nově přidělenou paměť
                if (n - zapsano < 8 - p->obsazeno2) {
                        p->bity[p->obsazeno1] |= (x >> zapsano) << p->obsazeno2;
                        p->obsazeno2 += n - zapsano;
                        zapsano += n - zapsano;
                }       else {
                        p->bity[p->obsazeno1] |= ((x >> zapsano) & ((1 << (8 - p->obsazeno2)) - 1)) << p->obsazeno2;
                        zapsano += 8 - p->obsazeno2;
                        p->obsazeno1++;
                        p->obsazeno2 = 0;
                }
        }
}

//zapíše x na konec p, víme, že x je z {a, ..., b}
void zapis_z_intervalu(struct bitove_pole * p, unsigned x, unsigned a, unsigned b) {
        zapis_bity(p, x - a, log_2(b-a+1));
}

//přidá do bitového pole hodnotu x, M je maximální hodnota
//zachovává setříděnost, takže musíme přepsat celé pole, abychom mohli vložit novou hodnotu
void zapis_cislo(struct bitove_pole * p, unsigned x, unsigned M) {
        struct bitove_pole nove;
        nove.alokovano = p->alokovano;
        nove.bity = malloc(nove.alokovano);
        nove.obsazeno1 = nove.obsazeno2 = 0;
        bool zapsano = false;
        unsigned min = 0;
        p->pozice1 = p->pozice2 = 0;
        while (jde_cist_bity(p)) {
                unsigned c = cti_z_intervalu(p, min, M);
                if (!zapsano && x < c) {
                        zapis_z_intervalu(&nove, x, min, M);
                        min = x + 1;
                        zapsano = true;
                }
                zapis_z_intervalu(&nove, c, min, M);
                min = c + 1;
        }
        if (!zapsano)
                zapis_z_intervalu(&nove, x, min, M);

        free(p->bity);
        *p = nove;
}
```

```
bool get_bit(uint8_t * p, unsigned i) { //přečte bit na pozici i z pole bytů
	return (p[i / 8] & (1 << (i % 8))) != 0;
}

void set_bit(uint8_t * p, unsigned i, bool x) { //zapíše bit x na pozici i v poli bytů
	if (x) p[i / 8] |= (1 << (i % 8));
	else p[i / 8] &= ~(1 << (i % 8));
}

//hlavní funkce programu, vrací počet jedinečných znaků na vstupu
unsigned jedinecnych_znaku(void) {
	prectene.bity = NULL;
	prectene.alokovano = prectene.obsazeno1 = prectene.obsazeno2 = 0;
	unsigned znaku = 0;
	bool zpusob2 = false;
	scanf("%u %u", &N, &K);
	struct znak z;
	z.radky = (unsigned *)malloc(N * sizeof(&z.radky));
	for (unsigned i = 0; i < N; i++)
		z.radky[i] = N;
	unsigned M = cislo_znaku(z);
	unsigned logM = log_2(M);
	for (unsigned i = 0; i < K; i++) {
		nacti_znak(&z);
		unsigned c = cislo_znaku(z);
		if (!zpusob2) { //použijeme první způsob: seznam přečtených znaků, uložených pod svým číslem
			prectene.pozice1 = prectene.pozice2 = 0;
			unsigned min = 0;
			bool nalezeno = false;
			while (!nalezeno && jde_cist_bity(&prectene)) {
				unsigned c2 = cti_z_intervalu(&prectene, min, M);
				if (c2 == c)
					nalezeno = true;
				else
					min = c2 + 1;
			}
			if (!nalezeno) {
				znaku++;
				zapis_cislo(&prectene, c, M);
			}
			if (prectene.alokovano >= (M + 7) / 8) { //přejdeme na druhý způsob
				zpusob2 = true;
				uint8_t * bity = (uint8_t *)calloc((M + 7) / 8, 1);
				prectene.pozice1 = prectene.pozice2 = 0;
				unsigned min = 0;
				while (jde_cist_bity(&prectene)) {
					unsigned c = cti_z_intervalu(&prectene, min, M);
					set_bit(bity, c, true);
					min = c + 1;
				}
				free(prectene.bity);
				prectene.bity = bity;
			}
		}	else {
			//používáme druhý způsob: pole bitů, když je i. bit 1, tak jsme už někdy přečetli znak s číslem i
			if (!get_bit(prectene.bity, c)) {
				znaku++;
				set_bit(prectene.bity, c, true);
			}
		}
	}
	free(z.radky);
	free(prectene.bity);
	return znaku;
}

int main(void) {
	printf("%u", jedinecnych_znaku());
	return 0;
}
```

Výsledková listina dvacátého prvního ročníku KSP po čtvrté sérii

		škola	*ročník*	*sérií*	*2141*	*2142*	*2143*	*2144*	*2145*	*2146*	*série*	*celkem*
1.	Vojtěch Kolář	G Neratov	3	10	8	9	10	9	10	12	41,1	156,0
2.	Vítězslav Plachý	GJiříPoděb	3	4	4	9	10	1	3	11	36,4	154,7
3.	Petr Čermák	GEBenešeKL	3	4	8	3	5	5		9	33,1	142,1
4.	Filip Hlásek	GMikul23PL	2	9	8		10			9	27,6	140,8
5.	Jitka Novotná	G Bílovec	4	6	8		10			10	29,0	129,5
6.	Michal Bilanský	GLepařovJČ	3	4			10			2	13,8	122,2
7.	Pavel Veselý	G Strakon	4	16		5	6		4	9	17,9	113,1
8.	Karel Tesař	SPŠE Plzeň	3	6	6		4	1		5	21,3	107,7
9.	Jiří Cidlina	GVoděraPH	4	3							0,0	106,7
10.	Vlastimil Dort	GŠpitálsPH	3	14	8		6			12	25,1	97,8
11.	Lukáš Ptáček	GJAKŽeliez	3	3							0,0	92,5
12.	Alexander Mansurov	GNVPlániPH	0	3			5			7	17,2	87,3
13.	Martin Zikmund	G Turnov	1	4		2		5	2		14,5	79,6
14.	Pavel Taufer	ArcibisGPH	3	6	8			4		3	18,4	72,7
15.	David Věčorek	GTNovákBO	3	3							0,0	72,6
16.	Filip Štědronský	GMikul23PL	2	8							0,0	71,9
17.	Jiří Setnička	G25březnPH	2	5	8				6	3	20,8	63,0
18.	Jan Vaňhara	G Holešov	4	3							0,0	56,2
19.	Petr Pecha	SPŠSVsetín	2	5	8						8,0	46,6
20.	Karel Král	G Most	3	4	8	2	2				15,3	44,6
21.	Barbora Janů	GKepleraPH	2	4			0				0,0	44,3
22.	Štěpán Šimsa	GJungmanLT	0	3	8		2				12,0	41,2
23.	Alžběta Pechová	SPŠSVsetín	4	7			0				0,0	39,4
24.	Libor Plucnar	GBezručeFM	4	11			2				2,0	38,2
25.	Filip Sládek	GNámestovo	3	1							0,0	38,1
26.	Karel Kolář	GŠpitálsPH	4	3							0,0	36,3
27. – 28.	Tomáš Pikálek	GBoskovice	2	1							0,0	35,6
	Jan Veselý	G Strakon	2	1							0,0	35,6
29.	Lukáš Chmela	GJŠkodyPŘ	0	1							0,0	35,2
30.	Ondřej Pelech	GJNerudyPH	4	1							0,0	33,4
31.	Radim Cajzl	GNoMěsNMor	2	19		5				8	7,9	33,0
32.	Petr Zvoníček	G Slavičín	3	3		2					3,9	30,4
33.	David Formánek	GJarošeBO	2	2							0,0	27,9
34. – 35.	Karolína Burešová	G ČesLípa	2	1							0,0	27,7
	Kateřina Lorenzová	G Česká ČB	2	4	8						8,0	27,7
36.	Milan Rybář	GJungmanLT	4	3							0,0	27,4
37.	Honza Žerdík	G Příbor	4	1							0,0	25,7
38.	Jan Škoda	GMikul23PL	2	4							0,0	23,3
39.	Alena Bušáková	G Trutnov	2	1							0,0	22,6
40.	Hynek Jemelík	GJarošeBO	2	2							0,0	19,0
41.	Jakub Sochor	G Bílovec	4	1							0,0	17,1
42.	Martin Holec	G Slavičín	2	3							0,0	15,5
43.	Mirek Jarolím	GMikul23PL	3	4							0,0	15,3
44.	David Vondrák	GDašickáPA	3	2							0,0	14,8
45.	Petr Babička	VOŠGSvetla	4	8		1					1,4	11,6
46.	Pavel Kratochvíl	VOŠGSvetla	1	6							0,0	10,2
47. – 49.	Jiří Daněk	GKřenováBO	3	1							0,0	10,0
	Jan Kostecký	VOŠŠumperk	2	2		3					5,5	10,0
	Jan Matějka	G Jírov ČB	4	5							0,0	10,0
50.	Dominik Smrž	GOhradníPH	0	3							0,0	8,7
51.	Jiří Keresteš	SPŠE Plzeň	3	5							0,0	7,4
52.	Pavol Rohár	G Košice	3	1			3				6,0	6,0
53.	Stanislav Fořt	GCoubTábor	1	8							0,0	5,9
54.	Igor Koníček	G UherBrod	3	1							0,0	5,7
55.	Ladislav Maxa	GKepleraPH	3	1							0,0	5,5