Korespondenční Seminář z Programování

ZAČÁTEČNICKÁ KATEGORIE

27. ročník **KSP-Z** **Listopad 2014**

První série letošního KSP-Z je za námi. Doufáme, že se vám úlohy líbily, a zde vám přinášíme jejich autorská řešení. Pokud byste k čemukoliv měli nějaké dotazy, neváhejte se nás ptát na našem fóru.

Potěšilo nás, že se tolik z vás s vervou pustilo do řešení. Pokud se vám z nějakých úloh nepodařilo získat plný počet bodů, nebo jste při jejich řešení úplně tápali a raději nic neposlali, podívejte se na vzorová řešení. Stejně tak to doporučujeme i těm, kteří získali plný počet bodů – vždy je dobré podívat se na problém třeba z jiného úhlu pohledu.

Řešení první série začátečnické kategorie 27. ročníku KSP

27-Z1-1 Na zastávce

Autobusovou úlohu vyřešíme tím nejjednodušším postupem, jaký vůbec může být: přímou simulací. To znamená, že v programu budeme provádět operace odpovídající tomu, co by se dělo v reálném světě (nástup skupinky do autobusu, odjezd autobusu), jednu po druhé ve stejném pořadí, v jakém by se odehrály doopravdy.

Samozřejmě většina programovacích jazyků neumí pracovat ani s lidmi, ani s autobusy, tak si místo toho pořídíme několik číselných proměnných, které budou náš svět popisovat:

- počet lidí v autobusu stojícím na zastávce
- počet autobusů, které již odjely
- pořadové číslo skupinky, která je aktuálně na začátku fronty

Nyní stačí postupně projít všechny skupinky a pro každou z nich upravit tyto proměnné tak, aby popisovaly situaci po odbavení dané skupinky dle pravidel v zadání. To je poměrně přímočaré a podrobněji si to můžete prohlédnout v ukázkovém programu.

Pozor je třeba dát si zejména na „plus/minus jedničkové“ chyby: tedy například nezapomenout započítat i poslední nezaplněný autobus, nebo naopak pokud se poslední autobus zcela zaplní, nezapočítat navíc ještě jeden prázdný.

Přímá simulace obvykle nepatří mezi nejefektivnější řešení, ale v případě naší úlohy jím je. Časová složitost řešení je lineární v počtu čekajících skupinek, a lépe to určitě nejde, neb v kratším čase by program ani nestihl přečíst svůj vstup.

Program (C):
`http://ksp.mff.cuni.cz/viz/27-Z1-1.c`

Filip Štědronský

27-Z1-2 Kalkulačka

Myšlenkový postup řešení úlohy s kalkulačkou byl přímočarý, stačilo jen postupně načítat operátory a čísla a provádět s nimi zadané operace. Důležité ale bylo rozmyslet implementační detaily.

Načítat čísla a operátory ze vstupu je nejlepší dělat v cyklu – a to buď v lichých krocích čísla a v sudých operátory, nebo rovnou v každém kroku celou dvojici čísla a operátoru.

Ať to budeme provádět jakkoliv, jeden krok výpočtu vždy provedeme ve chvíli načtení dalšího čísla. V nějaké proměnné si budeme držet dosavadní výsledek, pak se podíváme na operátor (abychom nemuseli psát série if-podmínek, nabízejí některé jazyky zkratku konstrukcí switch) a provedeme to, co je po nás požadováno. Zde je také správné místo k ošetření dělení nulou, při dělení nulou neprovedeme nic.

Zbývají dvě otázky. První z nich je, kdy provádět vypisování mezivýsledků. Zadání úlohy vyžadovalo, abychom mezivýsledek vypisovali při každém načtení operátoru. Stačí si uvědomit, že to je to samé, jako když mezivýsledek vypíšeme ve chvíli jeho spočítání (protože další na vstupu přijde vždy operátor), a tak to také uděláme.

Poslední věcí je, jak výpočet odstartovat a jak ukončit. Ukončení je jednoduché, budeme i = brát jako operátor, jen se speciálním významem ukončení programu (všimněte si, že výsledek k tomuto operátoru už vypsala poslední operace).

Začátek výpočtu je složitější, protože vždy po načtení nového čísla chceme provést výpočet. Jaký ale provést pro první číslo? Abychom to nemuseli řešit speciální podmínkou, inicializujeme na začátku proměnnou s výsledkem na nulu a operátor na plus. A to je celé, na implementaci se podívejte v programech níže.

Program (C):
`http://ksp.mff.cuni.cz/viz/27-Z1-2.c`

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/27-Z1-2.py`

Jirka Setnička

27-Z1-3 Slovník T9

Začneme tím, že každé slovo přeložíme na jeho zápis v T9. Poté čísla rozdělíme do skupin tak, aby v jedné skupině skončila slova, která se v T9 píší stejně. A nakonec najdeme největší skupinu.

Jak to udělat konkrétně? V Pythonu si můžeme pořídit slovník, jehož klíče budou jednotlivé zápisy v T9. Ke každému

klíči přiřadíme pole, kam budeme ukládat všechna slova s tímto zápisem. Pak už jenom projdeme všechny klíče slovníku a najdeme ten, jehož pole je největší.

Rozmysleme si, jak rychlé naše řešení bude. Každá operace se slovníkem zabírá v průměru lineární čas s délkou klíče, nezávisle na tom, jak je slovník velký. (Slovník uvnitř funguje jako hešovací tabulka. Pokud vás zajímají detaily, nakoukněte do kuchařky o hešování.[1] Zde stačí vědět, že hešovací tabulky dovedou být velice rychlé, ale jenom v průměru; nejhorší případ může být až lineární s velikostí slovníku.)

Celý program proto poběží v průměrně lineárním čase se součtem délek všech slov, tedy s velikostí vstupu.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/27-Z1-3.py`

Na Céčkovém řešení si předvedeme jiný přístup: nejprve vytvoříme dvojice (*původní slovo*, *převedené slovo*). Pak tyto dvojice setřídíme podle převedených slov – můžeme se inspirovat kuchařkou o třídění,[2] případně použít knihovní funkci `qsort`.

Setříděním se dostanou k sobě slova se stejným zápisem, takže je snadno poznáme a najdeme největší takovou skupinu.

Opět si rozmysleme časovou složitost. Setřídění n hodnot kvalitním třídicím algoritmem (třeba MergeSortem) trvá $\mathcal{O}(n \log n)$ porovnání, projití setříděného slovníku spotřebuje dalších $\mathcal{O}(n)$ porovnání. A jelikož zadání omezuje slova na 15 písmen, můžeme předpokládat, že slova umíme porovnávat v konstantním čase. Časová složitost proto činí $\mathcal{O}(n \log n)$.

Program (C):
`http://ksp.mff.cuni.cz/viz/27-Z1-3.c`

Martin „Medvěd“ Mareš

27-Z1-4 Lyžař

Nejprve si pojďme rozmyslet, jak bychom řešili situaci pro kopec výšky dva, který by vypadal třeba takto:

```
  1
2 3
```

U takového kopce se stačí podívat doleva a doprava, a kde je vyšší číslo, tam pojedeme. Teď si pojďme ukázat, že i z velkého kopce se dá postupně vyrobit kopec výšky dva. Nejprve si to ukážeme na kopci výšky tři:

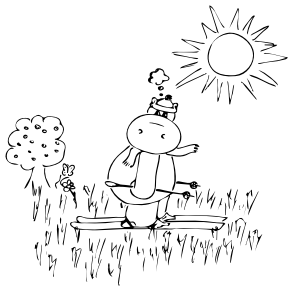
```
    1
  2 3
3 2 1
```

Pro snazší orientaci budeme jednotlivá místa na kopci označovat podle toho, v kolikáté jsou řadě seshora, a v kolikáté jsou řadě zleva, zapisovat to budeme jako $[3, 1]$ (což v tomto případě značí trojku ve třetím řádku vlevo).

Nejprve se podíváme na místo $[2, 1]$: Kdyby kopec začínal zde, stačí se podívat jen doleva a doprava dolů a víme, kam se máme z tohoto místa vydat. Tedy k místu $[2, 1]$ přičteme hodnotu v místě $[3, 1]$, protože je větší než v místě $[3, 2]$.

Teď si představíme, že kopec začíná v místě $[2, 2]$, a provedeme pro něj stejný postup jako pro místo $[2, 1]$ (jen k němu přičteme hodnotu z $[3, 2]$, protože je větší než v $[3, 3]$). Nyní můžeme zapomenout na třetí řádek, protože už víme, do kterých míst se nám vyplatí jet z druhého řádku.

Po zapomenutí třetího řádku máme opět kopec výšky dva a pro něj už umíme zjistit řešení jednoduše. Tento postup ale nezáležel na tom, že je kopec vysoký zrovna tři řady. Když tento postup zobecníme, budeme umět vyřešit i kopec libovolné výšky.



Konstrukce zespodu

Zkusíme tedy zobecnit postup pro kopec výšky tři na kopec výšky N. Budeme ho postupně zespodu snižovat:

1. Pro i od 1 do $N - 1$:
2. $\quad max \leftarrow$ maximum z $[N, i]$ a $[N, i + 1]$
3. $\quad [N - 1, i] \leftarrow [N - 1, i] + max$

Na poslední řádek teď můžeme zapomenout. Tímto z kopce výšky N vyrobíme kopec výšky $N - 1$ a opakováním postupu se dostaneme až na kopec výšky jedna, který už je sám o sobě řešením.

Tento postup bude pro kopec výšky N trvat $\mathcal{O}(N^2)$, protože na každé místo se podíváme maximálně třikrát a všech míst je dohromady $\mathcal{O}(N^2)$.

Řešení, které jsme si právě předvedli, se dá považovat za jednu z technik *dynamického programování*, neboli skládání řešení velkých problémů z řešení malých.

Program (C):
`http://ksp.mff.cuni.cz/viz/27-Z1-4-dynamika.c`

Rekurzivní náhled

Druhý náhled na úlohu může být shora dolů. Kdybychom znali maximální součet na celé cestě začínající pod námi vlevo nebo vpravo, tak bychom si z nich už snadno vybrali, kam se máme vydat.

Toho můžeme využít pro řešení pomocí rekurze. Pokaždé se našeho programu zeptáme, jaký je součet vlevo a vpravo, porovnáme je, a pak jako výsledek vrátíme součet toho většího z nich a hodnoty v aktuálním místě.

Jen to nelze naprogramovat takto přímo. Kdybychom totiž pokaždé počítali výsledek pro všechna nižší místa, výpočet by se spouštěl pro každé místo mnohokrát. Kolikrát přesně může nastínit schéma níže. Znalejší z vás si možná všimli, že je to vlastně *Pascalův trojúhelník*:

```
      1
    1 1
  1 2 1
1 3 3 1
1 4 6 4 1
. . . . . .
```

[1] `http://ksp.mff.cuni.cz/viz/kucharky/hesovani`
[2] `http://ksp.mff.cuni.cz/viz/kucharky/trideni`

Vidíte, že počet rekurzivních volání roste velmi rychle. Na prvním řádku se ptáme na dvě hodnoty pod námi, na druhém se ptáme na tři hodnoty pod námi, z toho ale na jednu dvakrát, na třetím řádku se ptáme již čtyřikrát, na čtvrtém osmkrát a tak dále.

Dalo by se napsat, že se na i-tém řádku ptáme řádově na 2^i hodnot pod námi, což nám dává celkovou časovou složitost $\mathcal{O}(2^N)$. Druhý možný náhled dávající stejnou složitost vypadá tak, že se podíváme na možné cesty, kudy se můžeme vydat. Při cestě dolů z kopce se na každém místě rozhodujeme mezi dvěma směry a toto rozhodnutí děláme N-krát, což nám opět dává $\mathcal{O}(2^N)$ možností.

Abychom tak zbytečně mnohokrát nepočítali něco, co už víme, vždy si vypočtenou hodnotu uložíme do pomocného pole stejné velikosti, jako má sjezdovka, a zapamatujeme si, že pro tuto cestu již výsledek známe.

Když se pak v programu budeme ptát na hodnotu cesty, nejprve zjistíme, jestli už ji máme spočítanou, a pokud ano, jen vrátíme výsledek. Jinak spočítáme cestu, uložíme výsledek opět do pole a zapamatuje si, že už pro toto místo cestu spočítanou máme.

Na každé místo se tak budeme ptát maximálně dvakrát, což bude trvat $\mathcal{O}(N^2)$, neboli lineárně s velikostí vstupu. Lépe to jistě nepůjde, neboť vstup musíme určitě přečíst celý. Kdybychom ho celý nečetli, můžeme do některého nepřečteného místa dosadit dostatečně velkou hodnotu, aby změnila optimální cestu, ale náš program by takové změněné řešení neměl jak poznat.

Ukázali jsme tedy dvě různá řešení, která ve výsledku vedou k něčemu velmi podobnému. První implementaci jsme ukázali již výše, na druhou se můžete podívat zde:

Program (C):
`http://ksp.mff.cuni.cz/viz/27-Z1-4-rekurze.c`

Vojta Sejkora

27-Z1-5 Cédéčko z koncertu

Pomalé řešení

Ze zadání víme, že máme najít souvislý úsek písniček takový, že součet jejich délek je přesně K. Písniček je N, a tak si můžeme zvolit až N různých písniček, kterými by CDčko mohlo začínat. Ke každému možnému začátku existuje až N možných posledních písniček, což nám dává řádově N^2 možností (dvojic začátků a konců). Tak si je zkusme všechny projít a pro každou spočítat délku písniček mezi nimi (včetně jich samotných).

Jak to provedeme? Pojedeme v nějakém cyklu začátkem přes celou posloupnost písniček, v něm dalším cyklem přes možné konce a v každém takovém úseku ještě třetím cyklem spočítáme součet délek všech v něm obsažených písniček. Spočtení každého úseku bude trvat čas $\mathcal{O}(N)$, a pro N^2 úseků tedy celkově $\mathcal{O}(N^3)$.

To se nám ale moc nelíbí, tak to zkusíme zlepšit použitím prefixových součtů z naší základní kuchařky.[3] Čas potřebný na spočítání úseku se tím sníží na $\mathcal{O}(1)$ a celkově tedy na $\mathcal{O}(N^2)$, ale stále zbytečně počítáme pořád dokola skoro ty stejné věci. Ovšem my máme rádi rychlé algoritmy, nedá se to tedy zlepšit?

Zrychlujeme

Algoritmus bude pracovat následovně. Budeme mít tři proměnné: začátek a a konec b, které budou ukazovat na první, respektive poslední písničku aktuálního úseku, a součet S délek písniček aktuálního úseku (na začátku bude velký jako délka první písničky). V každém kroku výpočtu porovnáme S s číslem K, mohou nastat tři možnosti:

- $S = K$: V tomto případě jsme vyhráli a nalezli jsme úsek se součtem K začínající písničkou a a končící b.
- $S < K$: Zde vidíme, že se nám na CD ještě něco vejde, tak zkusíme přidat další písničku. To znamená, že konec úseku posuneme o jednu písničku dál (b zvýšíme o jedna) a délku b-té písničky přičteme k S. Opakujeme porovnání.
- $S > K$: Tady už přidání další písničky nemůže pomoci (rozdíl by se pouze zvětšoval), tedy víme, že a-tou písničkou nemůže hledaný úsek začínat. Ovšem další písničkou ano, proto od S odečteme délku a-té písničky a a posuneme o jednu pozici dál na následující písničku. Opakujeme porovnání.

Pokud začátek nebo konec chceme posunout, ale už není na jakou písničku, tak ohlásíme, že neexistuje úsek písniček, který by bylo možné na CD zapsat (ve skutečnosti stačí kontrolovat pouze konec, neboť pokud bychom a zvýšili tak, že by „ukazovalo“ na neexistující písničku, tak by byl součet nulový, a tedy bychom posouvali b).

Důkaz správnosti

Algoritmus úspěšně seběhl, tak si ověřme, že bude fungovat vždy. Už víme, že možných úseků je řádově N^2, musíme je ale procházet všechny? Pokud je součet našeho úseku od a do b příliš velký, tak odebráním první písničky současně vyřadíme všechny další nezkontrolované úseky začínající touto písničkou (zkontrolovali jsme jenom úseky končící maximálně v b).

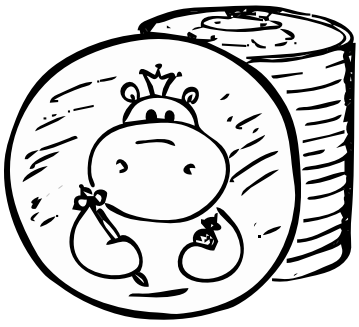
Ovšem každý takový úsek by měl součet určitě delší než úsek (a, b), který už sám byl příliš dlouhý. Vyloučili jsme tedy jen úseky, které by nás stejně nezajímaly.

Obdobným argumentem se můžeme podívat na posouvání b o jedna dál (všechny vyloučené ještě nezpracované úseky by byly příliš malé). Náš algoritmus tak vylučuje pouze ty úseky, u nichž už víme, že by nevyhovovaly. Všechny ostatní zkontrolujeme, tudíž pokud řešení existuje, najdeme ho.

Už víme, že algoritmus funguje, tak se pojďme podívat, jak dlouho mu to trvá. Při hledání úseku v každém kroku posuneme a nebo b o jedna dál, písniček je N, každá může být nejvýše jednou označena jako a a nejvýše jednou jako b. Tedy nejpozději po $2N$ krocích dojdeme na konec a ukončíme prohledávání. Celý algoritmus má tedy časovou složitost $\mathcal{O}(N)$.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/27-Z1-5.py`

Katka Zákravská & Jirka Setnička



[3] `http://ksp.mff.cuni.cz/viz/kucharky/zakladni-algoritmy`

27-Z1-6 Žárovky

Pre lepšie pochopenie riešenia si trošku upravíme zápisy. Ku žiarovkám pridáme ešte jednu, ktorá bude stále svietiť, pomenujme ju žiarovka *Nádej*.

Pôvodný počet žiaroviek si označíme ako $n-1$ a pridaním Nádeje budeme mať n žiaroviek. Nádej vždy svieti, a teda nám problém úlohy s $n-1$ žiarovkami upravuje na problém s n žiarovkami. Prečo tomu tak je? V pôvodnom prípade rozsvecujeme žiarovky v počtoch $0, 1, 2, \ldots, n-1$. V druhom prípade to upravíme na $1, 2, \ldots, n$, čo je vlastne Nádej + $(0, 1, 2, \ldots, n-1)$.

Zjednodušená úloha

Úlohu si na začiatok ešte trošku zjednodušíme. Obmedzíme počet žiaroviek len na mocniny dvojky $(1, 2, 4, 8, \ldots)$. Na túto zjednodušenú úlohu použijeme nasledovný algoritmus:

1. Žiarovky si rozdelíme na dve časti (polovice).
2. K časti, ktorá neobsahuje Nádej, pridelíme jeden spoločný vypínač.
3. Ak časť obsahuje len Nádej, tak program ukončíme (táto žiarovka už nepotrebuje vypínač, lebo sa nedá vypnúť). Inak opakujeme algoritmus od kroku jedna zavolaním sa na časť obsahujúcu Nádej.

Naším prvým krokom bude ukázať, že takéto rozdelenie vypínačov je správne – teda, že rozsvieti ľubovoľný počet žiaroviek z intervalu 1 až n. Využijeme k tomu indukciu. Mať rozsvietenú práve jednu žiarovku vieme pomocou žiarovky Nádej, pričom všetky ostatné vypínače sú vypnuté.

Implikáciu ukážeme takto: ak vieme postupne rozsvietiť $k/2$ žiaroviek (do stavu 1 až $k/2$ rozsvietených) a zároveň máme vypínač, ktorý rozsvieti druhú polovicu (teda presne $k/2$ žiaroviek), tak vieme rozsvietiť aj ľubovoľný počet žiaroviek od 1 do k. Stačí nám na to rozsvietiť druhú polovicu jedným vypínačom. A potom k takto rozsvietenej polovici vieme pridať 1 až $k/2$ rozsvietených žiaroviek.

Ďalej si ukážeme, že naše riešenie je optimálne. Náš algoritmus potrebuje i vypínačov pre 2^i žiaroviek. Každý vypínač sa môže nachádzať v jednej z dvoch polôh, a to buď zapnutý, alebo vypnutý. S i vypínačmi môžeme popísať práve 2^i rôznych stavov. Z rôzne „postláčaných“ vypínačov chceme získať rôzny počet rozsvietených žiaroviek. Ak by sme vypínačov mali len $i-1$, tak s nimi vieme dosiahnuť maximálne 2^{i-1} stavov. My ale potrebujeme 2^i rôznych zasvietení (1 až 2^i).

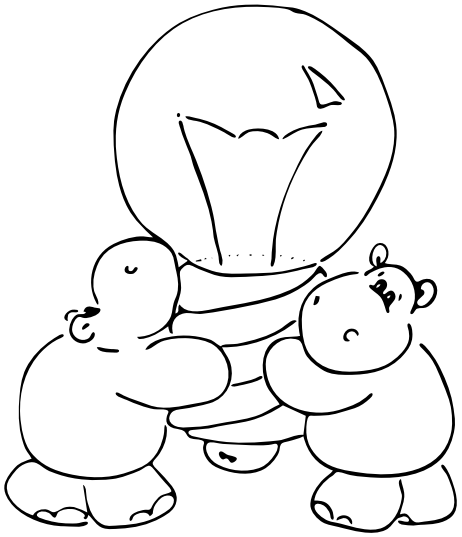
Pôvodná úloha

Na záver sa vrátime k pôvodnému zadaniu úlohy. Teda hľadáme riešenie pre ľubovoľný počet žiaroviek, nie len pre mocninou dvojky. No nezúfajme, naše predošlé riešenie sme nerobili zbytočne. Skupinu n žiaroviek si rozdelíme na dve časti. Prvá časť bude obsahovať našu Nádej a počet žiaroviek v tejto časti bude rovný najväčšej mocnine dvojky, ktorá je menšia alebo rovná n. Exponent tejto mocniny si označíme ako i.

Druhá časť bude obsahovať všetky zvyšné žiarovky a bude určite menšia ako tá prvá časť. Je to preto, lebo ak by bola druhá časť väčšia alebo rovná ako tá prvá, tak by sme potom mohli vziať väčšiu mocninu dvojky v prvej časti, než sme vzali. Prvú časť, ako sme dokázali v zjednodušenej úlohe, vieme najlepšie vyriešiť s i vypínačmi. Celú druhú časť pripojíme na jeden spoločný vypínač. Prvými i vypínačmi vieme rozsvietiť 1 až 2^i žiaroviek a posledným vypínačom zvyšok, teda $n - 2^i$ žiaroviek. Ak chceme rozsvietiť viac ako 2^i žiaroviek, stačí nám rozsvietiť druhú časť a k tomu doplňujúci počet žiaroviek z prvej časti.

Ešte potrebujeme ukázať, že potrebný počet vypínačov je najmenší možný. Pre n žiaroviek si nájdeme najbližšiu menšiu mocninu dvojky než n. Túto mocninu si označíme ako 2^i. Následne využijeme podobnú úvahu, akú sme použili pri 2^i žiarovkách. Ak by nám stačilo i vypínačov, tak vieme popísať 2^i stavov. Pritom vieme, že $2^i < n$, teda potrebujeme najmenej $i+1$ vypínačov.

Janka Bátoryová & Karolína „Karryanna“ Burešová



Výsledková listina první série začátečnické kategorie 27. ročníku KSP

	řešitel	*škola*	*ročník*	*sérií*	*Z1-1*	*Z1-2*	*Z1-3*	*Z1-4*	*Z1-5*	*Z1-6*	*série*	*celkem*
0.					8	10	10	12	12	14	66,0	66,0
1.	Jakub Pelc	G UherBrod	1	5	8	10	10	12	12	14	66,0	66,0
2.–3.	Miroslav Hrabal	GTomkovaOL	1	1	8	10	10	12	8	14	62,0	62,0
	Martin Scheubrein	G MNám Třb	3	1	8	10	10	12	8	14	62,0	62,0
4.	Jiří Štěpanovský	G MNám Třb	3	1	8	10	10	12	8	11	59,0	59,0
5.	Jakub Matěna	GČeskoliPH	3	1	8	10	10	12	7	11	58,0	58,0
6.	Jan Kaifer	GČesBrod	−1	1	8	10	10	12	9	8	57,0	57,0
7.–8.	Michal Töpfer	G DrJPekMB	2	4	8	10	10	12	10	4	54,0	54,0
	Lukáš Vlček	CírkG Plzeň	1	1	8	10	10	12	7	7	54,0	54,0
9.–10.	Matěj Fencl	GOA Chodov	1	1	8	10	10	12	8	3	51,0	51,0
	Lukáš Červený	G_Trutnov	1	1	8	10	10	12	6	5	51,0	51,0
11.	Jakub Neruda	GTNovákBO	4	2	8	10	10	12	10		50,0	50,0
12.–13.	Lukáš Fruněk	GLesníZlín	2	2	8	10	10	12	7	2	49,0	49,0
	Ondřej Švanda	G BO-Řeč	4	1	8	10	10	12	5	4	49,0	49,0
14.	Lukáš Mičan	GČeskáČB	1	1	8	10	10	12	3	3	46,0	46,0
15.–16.	Petr Klanica	GJarošeBO	2	1	8	10		12	7	8	45,0	45,0
	Zoltán Onódy	SPŠE NZám	4	1	8	10	10	12	5		45,0	45,0
17.	Jakub Jirkal	GJungmanLT	0	1	8	10	8	12	4	2	44,0	44,0
18.–19.	Daniel Nigrin	GÚstavníPH	2	1	8	10	10		6	8	42,0	42,0
	David Žáček	GZborovPH	2	1	8	10	10	1	9	4	42,0	42,0
20.–29.	Patrik Bak	G Sobrance	4	1	8	10	10	12			40,0	40,0
	Tat Dat Duong	G Wicht	2	1	8	10	10	12			40,0	40,0
	Václav Fabík	ZŠKřídloBO	0	5	8	10	10	12			40,0	40,0
	Karolína Kuchyňová	GMLerchaBO	4	1	8	10	10	12			40,0	40,0
	Jakub Lukeš	GNAlejíPH	2	5	8	10	10	12			40,0	40,0
	Vojtěch Lukeš	GPikaPL	3	1	8	10	10	12			40,0	40,0
	Jiří Moravčík	GUHradiště	1	1	8	10	10	12			40,0	40,0
	Michal Převrátil	GKlatovy	2	3	8	10	10	12			40,0	40,0
	Jiří Sejkora	GVoděraPH	3	1	8	10	10	12			40,0	40,0
	Jan Václavek	GUnOrl	3	2	8	10	10	12			40,0	40,0
30.	David Tvrdý	GHeyrovPH	2	1	8	10	10	10			38,0	38,0
31.–32.	Tomáš Chvosta	GPří	4	1	8	10	10	1			29,0	29,0
	Tomáš Troján	G Cheb	−1	1	8	10	10	1			29,0	29,0
33.–36.	Jan Burda	G_Holice	0	4	8	10	10				28,0	28,0
	Lukáš Holeczy	GTep	3	1	8	10	10				28,0	28,0
	Zdeněk Pavlátka	GMikulášPL	3	3	8	10	10				28,0	28,0
	Daniel Pluskal	G BO-Řeč	1	1	8	10	10	0			28,0	28,0
37.	Tereza Kotěšovcová	GKlatovy	4	1	8	10	0		6	3	27,0	27,0
38.	Janek Hlavatý	ZŠ_DukelČB	−4	4	8	10			7		25,0	25,0
39.	Dominika Tanglová	G_Nymburk	2	1	8	10				4	22,0	22,0
40.	Jan Sliacky	G_Benesov	4	1	8		10		3		21,0	21,0
41.–42.	Victoria María Nájares Romero	GZborovPH	1	1	8				6	6	20,0	20,0
	Benedikt Žour	G UherBrod	0	3	8	10	2				20,0	20,0
43.–50.	Mirolsav Březík	GLesníZlín	0	1	8	10					18,0	18,0
	Nhat Minh Dinh Huy	GKadaň	2	1	8	10					18,0	18,0
	Michal Nekvinda	BiGyBBHK	4	1	8	10					18,0	18,0
	Alexej Popovič	SlovanGOL	3	1	8	10					18,0	18,0
	Pavel Souček	G_Nymburk	3	1	8	10					18,0	18,0
	Vojta Staněk	PORGPha	1	1	8	10					18,0	18,0
	David Ucháč	eduSOŠ PA	2	1	8	10					18,0	18,0
	Petr Šíma	GKlatovy	1	5	8	10					18,0	18,0
51.	Matúš Maďar	GHorMichal	3	1	2					11	13,0	13,0
52.	Jan Vozár	G UherBrod	1	4		10			1,5		11,5	11,5
53.	Zuzana Šimečková	GČeskáČB	4	1	2			1		8	11,0	11,0
54.–58.	Michaela Bačová	G UherBrod	4	3		10					10,0	10,0
	Antonín Bruštík	G UherBrod	4	5		10					10,0	10,0
	David Dvořáček	G UherBrod	4	3		10					10,0	10,0
	Viktor Kovařík	G UherBrod	4	4		10					10,0	10,0
	Jakub Šmahovský	G Pezinok	3	1	8	2					10,0	10,0
59.	Roman Ondráček	GBoskovice	1	1		0	8	0		1	9,0	9,0

	řešitel	*škola*	*ročník*	*sérií*	*Z1-1*	*Z1-2*	*Z1-3*	*Z1-4*	*Z1-5*	*Z1-6*	*série*	*celkem*
60.–65.	Matej Hockicko	TAPoprad	1	1	2	6	0				8,0	8,0
	Michael Kozel	GZborovPH	1	1	8						8,0	8,0
	Ján Pavlus	GTNovákBO	2	1	8						8,0	8,0
	Martin Sklenár	GTajBanBys	3	1	8						8,0	8,0
	Michaela Svatošová	GKepleraPH	1	1	8						8,0	8,0
	Petr Zelina	GJarošeBO	2	1	8						8,0	8,0
66.	Markéta Machalová	G Wicht	2	1	2	2		0			4,0	4,0
67.	Ľuboš Kolumber	SpojŠ Popr	3	1	2	0					2,0	2,0