Korespondenční Seminář z Programování

36. ročník **KSP** **Červenec 2024**

Milé řešitelky, řešitelé a řešitelčata!

Na křídlech léta za vámi plachtí vzorová řešení poslední série letošního KSP. Přejeme příjemné čtení, ať už někde u vody, nebo v přízračném svitu monitoru za tiché letní noci :)

Těšíme se na vás v příštím ročníku, jehož zadání se během září. A také na podzimním soustředění, na něž už jsme rozeslali pozvánky.

Vaši organizátoři

Vzorová řešení páté série třicátého šestého ročníku KSP

36-5-1 Opravování opravy vzducholodi

Každé rozdělení řešení mezi organizátory můžeme ohodnotit dvěma čísly – počtem organizátorů K' a časem potřebným k opravení L'. Na K' máme zadaný ostrý limit $K_{\max} := K$, zato L' se snažíme získat co nejmenší.

Co kdybychom řešili o trochu jinou úlohu? Tentokrát budeme mít horní limit na L', a to $L_{\max}$, optimalizovat budeme pro změnu K'. Jinak řečeno, snažíme se řešení rozdělit mezi co nejméně organizátorů tak, aby jim opravování trvalo nejvýše $L_{\max}$ mikrosekund.

Nějaký organizátor bude určitě muset opravit první řešení. Je jasné, že to bude první z jemu přidělených řešení, vybrat si můžeme jen to, kolik řešení opraví celkem. Určitě nic nezkazíme, pokud mu přidělíme tolik řešení, kolik jen můžeme – pokud na zbylé organizátory zbude méně řešení, tak zbylých organizátorů nemůže být potřeba více. Přidělíme mu tedy co nejvýše řešení a zbylá rozdělíme stejným způsobem. Můžeme tedy použít jednoduchý hladový algoritmus běžící v čase $\mathcal{O}(N)$.

Už víme, jak pro dané $L_{\max}$ určit minimální K', ale my chceme pro dané $K_{\max}$ určit minimální L'. K tomu využijeme binárního vyhledávání. Pomocí něj budeme hledat čas potřebný k opravení všech řešení. Dolní mez vyhledávání bude největší z časů $T_1, \ldots, T_N$, zatímco horní mez bude jejich součet.

Mějme nějaký časový limit $L_{\max}$. Použijeme náš hladový algoritmus k spočtení minimálního počtu organizátorů K'. Pokud $K' \leq K_{\max}$, tak $K_{\max}$ organizátorů stihne všechna řešení v tomto čase opravit, hledaný minimální potřebný čas tedy bude $L_{\max}$ nebo nižší, můžeme proto horní mez nastavit na $L_{\max}$. Naopak pokud $K' > K_{\max}$, tak v čase $L_{\max}$ všechna řešení není možné opravit a musíme vyzkoušet vyšší časový limit. Nastavíme proto dolní mez na $L_{\max} + 1$.

Tím zjistíme, kolik času $K_{\max}$ organizátorů potřebuje, aby opravili všechna řešení. Nakonec naposledy použijeme hladový algoritmus k tomu, abychom našli rozdělení, které vypíšeme. Počet kroků tohoto binárního vyhledávání bude logaritmický vzhledem k součtu $T_1, \ldots, T_N$. V každém kroku použijeme hladový algoritmus, co běží v lineárním čase. Celkem tedy získáme časovou složitost $\mathcal{O}(N \log S)$, kde $S = T_1 + \cdots + T_N$ je součet všech čísel na vstupu.

Program (C++):
`http://ksp.mff.cuni.cz/viz/36-5-1.cpp`

Úlohu připravili: Dan Skýpala, Ben Swart

36-5-2 Polygloti

Úlohu na vstupu snadno převedeme do řeči grafů: máme zadaný orientovaný graf a v něm nějaký startovní vrchol v_s a cílový vrchol v_c a chceme najít dvě cesty z v_s do v_c, které se co nejvícekrát rozdělí a zase potkají. Důležité je, že náš graf je speciální – všechny hrany jsou orientované směrem z vrcholu s nižším číslem do vrcholu s vyšším číslem. Jinými slovy, graf je *acyklický*, neboli *DAG* (z anglického *directed acyclic graph*). Nahlédneme totiž, že jakmile se z libovolného vrcholu po hranách vydáme pryč, už se do něj nikdy nemůžeme vrátit, a tedy tento vrchol nemůže být součástí žádného cyklu.

Úloha přímo vybízí k použití dynamického programování: není jasné, jak by takové optimální řešení mělo vypadat, a hladové přístupy stylu „najdi nejdelší cestu a pak k ní najdi druhou cestu s nejvíce rozděleními a spojeními“ zjevně nebudou dávat správné výsledky.

Pro zjednodušení popíšeme řešení, které pouze vypíše největší dosažitelný počet rozdělení. Lze ho upravit, aby umělo i zrekonstruovat nalezenou dvojici cest; pro detaily se podívejte do zdrojového kódu. Budeme chtít spočítat pole A, kde $A[i]$ značí následující: ze všech dvojic cest, které začínají v v_s a (obě) končí v i, vezmeme dvojici s největším počtem rozdělení a spojení, a tento počet chceme po doběhnutí programu mít spočítaný v $A[i]$. Po spočítání celého A pak jen vypíšeme $A[v_c]$ jako výstup programu.

Pro spočítání pole A se nám bude hodit předpočítat si odpovědi na dotazy typu „Kolik existuje různých cest z u do v?“. To uděláme jednoduchým dynamickým programem, který postupně spustíme pro každé u. Nejprve vyřešíme okrajové případy: z u do u existuje právě jedna cesta (ta nulové délky), kdežto z u do $v < u$ existuje vždy nula cest. Teď postupně spočítáme počet cest do $v = u + 1, u + 2, \ldots, n$. Pro daný vrchol v se podíváme na všechny jeho předchůdce $p_1, \ldots, p_k$ – tedy vrcholy ze kterých do v vede hrana. Do nich už máme počet cest z u správně spočítaný, a počet různých cest z u do v je přesně součet počtů cest z u do p_1, $p_2, \ldots,$ až p_k.

Tento předvýpočet nám zabere $\mathcal{O}(n + m) = \mathcal{O}(m)$ času pro jedno u, a tedy $\mathcal{O}(nm)$ dohromady (za předpokladu, že graf je souvislý, a tudíž $n = \mathcal{O}(m)$). Jeho výsledkem je dvojrozměrná tabulka T, kde v $T[u][v]$ máme spočítaný počet různých cest z u do v.

Nyní už můžeme přejít k samotnému výpočtu pole A. Připomeňme, že $A[i]$ značí počet rozdělení nejlepší dvojice cest začínajících v v_s a končících ve vrcholu i. Jak z hodnot

$A[1], \ldots, A[i-1]$ spočítat $A[i]$? Snadno: libovolná dvojice cest, která se potkává v $i \neq v_s$, se potkává v nějakém $j < i$, odkud pak obě cesty nějak pokračují do i. Můžeme tedy začít s $A[i] = 0$, pak vyzkoušet všechny $j < i$ a rozlišit tři případy podle počtu cest vedoucích z i do j:

- Pokud $T[j][i] \geq 2$, pak $A[i]$ můžeme zlepšit na $A[j] + 1$, jelikož můžeme nejlepší dvojici cest do j prodloužit do i a vyrobit přitom další rozdělení.
- Pokud $T[j][i] = 1$, pak $A[i]$ můžeme zlepšit na $A[j]$, jelikož můžeme nejlepší dvojici cest do j prodloužit do i, ale bez vyrobení dalšího rozdělení.
- Pokud $T[j][i] = 0$, pak pro toto j nic neděláme, jelikož cesty do j nejdou prodloužit do i.

Můžeme tedy projít všechna $i = 1, \ldots, n$ a pro každé postupně vyzkoušet všechna $j = 1, \ldots, i-1$. Takto v čase $\mathcal{O}(n^2)$ spočítáme celé A. Časová složitost celého algoritmu je tak $\mathcal{O}(nm + n^2) = \mathcal{O}(nm)$, paměťová $\mathcal{O}(n^2)$.

Ještě drobný detail: různých cest mezi dvojicí vrcholů může být řádově 2^n, takže pro počítání T přesně bychom museli pracovat s $\mathcal{O}(n)$-bitovými čísly. Tomu se vyhneme tak, že všechny hodnoty větší než 2 nahradíme při ukládání do T dvojkou. Chování algoritmu se tím nijak nezmění.

Program (C++):
`http://ksp.mff.cuni.cz/viz/36-5-2.cpp`

Kvadratické řešení

Složitost $\mathcal{O}(nm)$ stačila na plný počet bodů, jde to však i lépe. Ukážeme si řešení, které pracuje v čase $\mathcal{O}(n^2)$, což je výrazně lepší pro husté grafy.

Stále budeme počítat pole A, ale bez pomoci tabulky T, jejíž konstrukce nás brzdila. Místo toho si pro každý vrchol v budeme pamatovat množinu $M[v]$ všech vrcholů u takových, že $A[u] = A[v]$ a z u existuje cesta do v. Platí totiž následující tvrzení:

Tvrzení. Mějme vrchol v a jeho předchůdce $p_1, \ldots, p_k$ a označme největší hodnotu $A[p_i]$ jako x. Pak buď $A[v] = x$, nebo $A[v] = x + 1$ a druhá situace nastane právě tehdy, když pro dva různé předchůdce p_i, p_j s $A[p_i] = A[p_j] = x$ platí, že $M[p_i] \cap M[p_j] \neq \emptyset$.

Pojďme se přesvědčit o jeho pravdivosti. Určitě $A[v] \geq x$, jelikož nejlepší dvojici cest do nějakého předchůdce x umíme vždy prodloužit až do x. Nerovnost platí ostře právě tehdy, když existuje nějaký vrchol w s $A[w] \geq x$, ze kterého existují dvě cesty do v. Nutně ale musí platit $A[w] \leq x$ (jelikož optimální dvojice cest do w jde prodloužit do v a tudíž i nějakého p_i), a tudíž $A[w] = x$. Víme též, že ony dvě cesty z w do v se musejí spojit až v x, jinak by už nějaké p_i mělo $A[p_i] = x+1$, což není možné. Jinými slovy, musejí existovat dva *různí* předchůdci v dosažitelní z w. Označme je p_i a p_j. Díky dosažitelnosti platí $A[p_i] = A[p_j] = x$. Z definice M ale plyne, že $w \in M[p_i]$ a $w \in M[p_j]$, přesně, jak jsme chtěli.

Tvrzení nám dává jednoduchý návod na algoritmus: postupně pro každé $i = 1, \ldots, n$ počítáme $A[i]$ a ruku v ruce s ním i $M[i]$. Množiny M budeme reprezentovat pomocí hešovacích tabulek. Pro každé i vždy nejprve spočítáme sjednocení S všech $M[p_1], \ldots, M[p_k]$; přitom už rovnou kontrolujeme, zda se nám nějaký prvek objevil dvakrát. Pokud ne, pak dle předchozího tvrzení víme, že $A[i] = x$, a můžeme položit $M[i] = S$. Pokud ano, pak naopak víme, že $A[i] = x + 1$, a jelikož jediný vrchol s $A[\cdot] = x + 1$, ze kterého je dosažitelný vrchol i, je vrchol i samotný, položíme $M[i] = \{i\}$.

Jaké má toto řešení časovou složitost? Na první pohled by se mohlo zdát, že jsme si vůbec nepomohli. Každé $M[i]$ obsahuje až n prvků a za každou hranu tak můžeme provést až $\Theta(n)$ práce při sjednocování. Pokud ale sjednocování budeme dělat chytře, a zastavíme se hned, jak objevíme duplicitní prvek, pak v každém vrcholu sjednocováním strávíme nanejvýš $\mathcal{O}(n)$ času – buď totiž nenarazíme na duplikát a pak jsme zákonitě museli vyrobit množinu s nejvýše n prvky, nebo jsme na duplikát narazili, ale opět nanejvýše po probrání n prvků. Celková časová složitost pak tedy opravdu je $\mathcal{O}(n^2)$. (Přesněji řečeno se jedná o *průměrnou* časovou složitost, protože hešujeme.)

Program (Rust):
`http://ksp.mff.cuni.cz/viz/36-5-2.rs`

Úlohu připravili: Filip Hejsek, Říša Hladík, Kristýna Petrlíková, Ben Swart

36-5-3 Všechny cesty vedou do Říma

Lehká varianta

Nejdřív vyřešme zjednodušenou variantu – vzdálenost Adama i Kačky od Říma je stejná, tedy $R_b = R_m$. (Čili oba dojdou do Říma za stejný počet kroků.) Všimněme si, že od prvního společného vrcholu budou jejich cesty stejné – z každého vrcholu je jednoznačně určeno, kam cesta bude pokračovat, až dojdou do Říma. Tedy cesty budou mít dvě části – první rozdílnou a druhou od prvního společného vrcholu, kde se oba vždy budou vyskytovat ve stejných vrcholech.

Z toho se nabízí snadný algoritmus v čase $\mathcal{O}(S_b + S_m)$: Dokud Adam a Kačka nejsou ve stejném vrcholu, tak se oba přesunou do dalšího vrcholu. Jakmile se oba vyskytnou ve stejném vrcholu, tak máme první společný, a můžeme skončit. Udržujeme si jen aktuální pozici Adama a Kačky, takže se vejdeme do požadované $\mathcal{O}(1)$ paměti.

Zobecnění

Nicméně vzdálenost Adama a Kačky od Říma není stejná. To snadno opravíme. Nejdřív si simulací spočteme vzdálenosti Adama i Kačky od Říma. Nyní nám akorát stačí posunout vzdálenějšího od Říma o $|R_m - R_b|$. A potom můžeme použít algoritmus výše.

Zjištění vzdálenosti od Říma zabere $\mathcal{O}(R_b + R_s)$, takže celý algoritmus potrvá $\mathcal{O}(R_b + R_s + S_b + S_m) = \mathcal{O}(R_b + R_s)$. Navíc si udržujeme vzdálenosti od Říma a jejich rozdíl, takže se stále vejdeme do $\mathcal{O}(1)$ paměti.

Zlepšujeme

Předchozí řešení je pomalé v případě, kdy se Adamova a Kaččina cesta potkají relativně brzy, tedy když R_b a R_s je o hodně větší než S_b a S_m. Nyní si ukážeme řešení pracující v čase $\mathcal{O}(S_b + S_m)$. Opět zkusíme najít $|R_m - R_b|$, ale tentokrát na to půjdeme chytřeji. Předpokládejme, že Adam je od Říma dál. Nejdřív uděláme k kroků s Kačkou (hodnotu k určíme za chvíli) a zapamatujeme si, kde jsme skončili – označme tento vrchol v_k. Poté budeme dělat nejvýš k kroků s Adamem. Pokud se po k' krocích dostaneme do v_k, tak platí $|R_m - R_b| = k - k'$, protože zbytek cesty je stejný. Pokud v_k neprojdeme, tak víme, že alespoň jeden po k krocích neskončil ve společné části, neboli $k < \max(S_b, S_m)$.

Nejdřív se zbavme předpokladu, že Adam je od Říma dál. Pokud to neplatí, pak Kačka nemá šanci dohnat Adama.

Proto můžeme vyzkoušet obě možnosti a nejvýš jedna vydá správný výsledek.

Ale jak si správně tipnout k, aby $k \geq \max(S_b, S_m)$? Budeme postupně zkoušet mocniny dvojky – nejdřív vyzkoušíme 1, potom 2, 4, 8, … Protože jsme se v předchozím pokusu netrefili, tak v posledním pokusu s k_ℓ kroky přestřelíme $\max(S_b, S_m)$ ani ne dvakrát:

$$\frac{k_\ell}{2} < \max(S_b, S_m) \leq k_\ell.$$

A kolik nás stálo zkoušení práce?

$$1 + 2 + 4 + \cdots + \frac{k_\ell}{2} + k_\ell \leq 2k_\ell.$$

Zde jsme využili vzorce pro částečný součet geometrické posloupnosti: $1 + 2 + 4 + 8 + \cdots + 2^n = 2^{n+1} - 1$.

Proto hledání rozdílu vzdáleností stihneme v čase $\mathcal{O}(k_\ell) = \mathcal{O}(\max(S_b, S_m))$. Po srovnání rozdílu $|R_m - R_b|$ můžeme použít algoritmus v lehké variantě. Zároveň si ověřme, že používáme jen $O(1)$ paměti. Stačí, když si budeme pamatovat:

– aktuální k
– kde Kačka aktuálně je a kolik kroků nám zbývá
– vrchol v_k, kde Kačka skončila
– kde je právě je Adam a zbývající počet kroků
– aktuální $|R_m - R_b|$

Takže máme algoritmus s časem $\mathcal{O}(S_b + S_m)$ a $\mathcal{O}(1)$ pamětí.

Úlohu připravili: Martin „Medvěd“ Mareš, Dan Skýpala

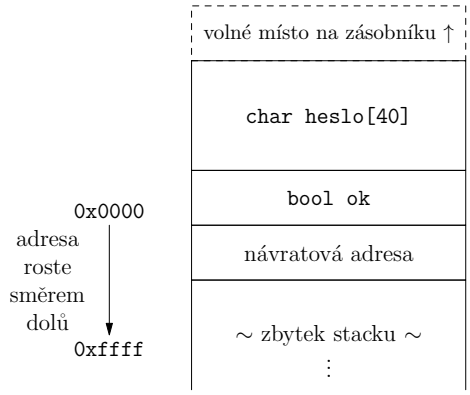
36-5-4 Hackovaci soutez

`Segmentation fault (core dumped)`, objevilo se na obrazovce společně s oběma vlajkami a Kačka mohla začít jásat. Dokonce ji ostatní soutěžící, oslněni jejím výkonem, pozvali po soutěži do hospody a Kačka zjistila, že vůbec nejsou tak ostří, jak na první pohled vypadali.

Než začneme úlohu řešit, musíme zjistit, co je s programem špatně – kde je v něm zranitelnost. V případě našeho programu to bude velmi jednoduché, neboť se hned na jednom z prvních řádků se volá funkce `gets`. Napovědět by nám mohlo, že když si otevřeme manuálovou stránku téhle funkce pomocí `man gets`, popis funkce začíná větou „Never use this function.“ Dokonce od verze C11 byla funkce odstraněna ze standardu a od glibc 2.16 funkce není exportovaná v headerech.

Proč je `gets` tak špatná? Jak známo, v C si musíme veškerou správu paměti řešit sami. Vždy máme vyhrazenou nějakou část paměti a držíme si ukazatel na její začátek. Abychom však s pamětí mohli správně pracovat, musíme si pamatovat i to, jak je ta část paměti dlouhá, jinak hrozí, že omylem zapíšeme za ni a přepíšeme cizí data. Všimněme si proto, že `gets` bere pouze jeden argument, kterým je ukazatel na cílový buffer. Do něj pak zapisuje, dokud mu uživatel posílá data, nehledě na tom, jestli už při tom nepřepisuje cizí paměť. Vzhledem k tomu, že nikdy nebudete mít k dispozici nekonečnou paměť, `gets` *vždy* způsobuje tzv. *buffer overflow*.

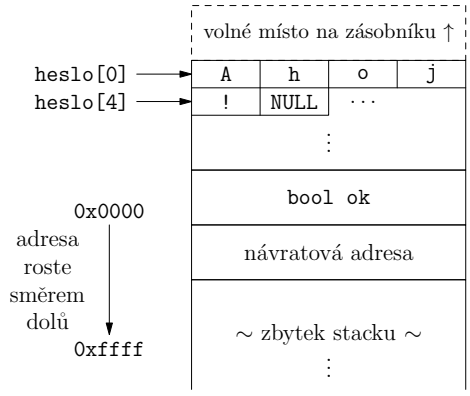
Máme tedy *buffer overflow* na bufferu `heslo` a můžeme přepisovat cokoliv, co se zrovna v paměti nachází za ním. Nejprve si všimneme, že buffer `heslo` je lokální proměnná, a nachází se proto na zásobníku. Můžeme tedy přepsat nějaká data na zásobníku, ale jaká? Vzpomeňte si na kuchařku k této úloze,[1] kde jsme si rozvržení zásobníku ukazovali. Zásobník roste směrem k nižším adresám a náš buffer je úplně poslední věc na zásobníku, proto je přímo za ním celý zbytek zásobníku.



Možná by se slušelo si říct, jak vlastně taková pole fungují. Na první pohled se to může zdát jako triviální otázka, ale bude se hodit. Intuitivně je pole nějaká posloupnost políček stejného typu, přičemž ke všem políčkům můžeme přistupovat v konstantním čase. Nás ale bude zajímat i implementace. Céčková pole jsou tak jednoduchá, jak to jen jde. V proměnné `heslo`, která pole reprezentuje, je uložený jen ukazatel na začátek místa vyhrazeného pro pole a operátor `a[i]` je tak jen zkratka za `*(a+i)`. Prostě k počáteční adrese pole přičteme index (implicitně vynásobený velikostí políčka) a vezmeme hodnotu z místa v paměti, kam ukazuje výsledný ukazatel.

To nám pomůže nahlédnout dvě věci. Jednak je zde dobře vidět, jak jednoduše k takovým přetečením dochází – když indexujeme pole, říkáme prostě „ukaž mi kus paměti o X míst dál“. Tam, když nejsme opatrní, může být už úplně něco jiného. Také vidíme, že i když stack roste směrem k nižším adresám, pole na něm uložená musí pořád růst stejně jako ta mimo zásobník – směrem k vyšším adresám. Když se předává pole jako argument, není totiž nikde specifikováno, jestli se jedná o pole na zásobníku nebo mimo něj.

Pojďme se tedy podívat, jak konkrétně bude vypadat zásobník, když do bufferu `heslo` napíšeme třeba `Ahoj!`.



Z toho už je vidět, že když nám buffer přeteče a budeme se pokoušet napsat data do paměti někam za něj, první věc, kterou přepíšeme, bude booleovská proměnná `ok`. V C neexistují opravdové booleany, ve skutečnosti jsou to celá čísla. Pokud je v proměnné uložená 0, považuje se za `false`, jinak se považuje za `true`. Teď už tedy víme, jak získat

[1] `http://ksp.mff.cuni.cz/viz/kucharky/prace-s-pameti`

první vlajku: potřebujeme přepsat hodnotu proměnné `ok` na cokoliv jiného než nulu, která je v ní zpočátku uložená. Teoreticky by nám tedy mělo stačit do bufferu `heslo` napsat 41× libovolný znak (třeba `A`), tím buffer přeteče, přepíše hodnotu `ok` na nenulovou a máme vyhráno. Stačit to ale nebude, protože proměnné typicky nejsou v paměti uložené těsně za sebou, ale kompilátor se často snaží, aby proměnné byly uložené na nějakých „kulatých“ adresách, takže mezi proměnnými může být ponechané volné místo. To, kolik znaků musíme poslat, už lze ověřit experimentálně; v našem případě to bude 48.

Hned další věc, kterou na zásobníku najdeme, je dvojice uložených hodnot registrů `rbp` a `rsp` při volání funkce (hodnotu `rbp` jsme pro jednoduchost v obrázcích výše neznázorňovali). Zajímat nás bude ta druhá z nich, neboť se jedná o návratovou adresu. Jak jsme si pověděli v kuchařce,[2] návratové adresy jsou asi to nejdůležitější, co se na zásobník ukládá. Když se na konci `main`u zavolá instrukce `ret`, vezme se posledních osm bytů ze zásobníku a na danou adresu se skočí, ať je tam cokoliv. Proto když do programu hodíme `A`ček příliš mnoho, program vyhodí Segmentation fault, čímž nám operační systém dává najevo, že jsme se pokoušeli přistupovat k paměti způsobem, kterým nemáme. V tomto konkrétním případě jsme se pokusili vykonat instrukci uloženou na adrese `0x4141414141414141`, kde bude s velkou pravděpodobností nenaalokovaná paměť, kterou rozhodně nemáme právo spouštět.

Kdyby se nám tedy povedlo místo osmi `A`ček nahradit návratovou adresu adresou nějakého spustitelného kódu, program na ni šťastně skočí a kód vykoná. Přesně takový spustitelný kód přeci máme. Konkrétně funkci `print_flag2`, která je za normálních okolností nedosažitelná. Stačí tedy prostě v našem „hesle“ nahradit správnou osmice `A`ček její adresou. Jediné, na co je potřeba si dát pozor, je tzv. endianita – čísla jsou na `x86` *little endian*, jako první je v paměti uložený *nejméně významný byte,* tedy ten, který je ve standardním zápisu až poslední.

Je zde také jedna věc, kterou nám autor úlohy řešení značně zjednodušil. Program, který běží na serveru, je totiž kompilovaný s parametrem `-no-pie`, tedy že program *nemá* být *Position Independent Executable*. Dnes je totiž poměrně častý default, že se programy kompilují tak, aby nezáleželo na tom, na jakou adresu se program do paměti načte. Typicky se všechny skoky v programu nahradí relativními (tedy místo „skoč na adresu `X`“ se použije „skoč o `X` bytů dál/zpět“) a skoky na knihovní funkce se vyřeší speciální tabulkou, kterou doplní linker/OS při startu programu. OS pak může využít techniky ASLR (*Address space layout randomization*), tedy při načítání programu do paměti ho prostě hodí na náhodnou adresu v paměti, aby bylo daleko obtížnější skákat do kódu a dělat neplechu. Tahle funkce byla ale při kompilaci vypnutá, takže adresy všech funkcí známe dopředu a můžeme je zjistit například pomocí příkazu `objdump -S`, který nám vypíše *disassembly* strojového kódu, včetně názvů funkcí a adres, na kterých jsou umístěné.

Disassembly je převedení strojového kódu (tedy jedniček a nul, kterým rozumí procesor) na assembly – lidsky čitelný seznam instrukcí, které procesor vykonává. V našem případě je to užitečné jen pro získání adresy funkce. V reálných *Hackovacích soutěžích*, ale i v reálné praxi, často nemáme k dispozici zdrojový kód programu, jen zkompilovanou binárku, takže musíme vyčítat co přesně kód dělá právě z jednotlivých instrukcí disassembly.

Teď ještě pár slov o mnoha různých způsobech, jak se od těchto teoretických poznatků dostat k vlajce. Vzhledem k tomu, že samotné provedení exploitu spočívá v poslání několika desítek konkrétních bytů, existuje mnoho způsobů, jak na to.

Ten technicky nejjednodušší způsob je prostě otevřít spojení se serverem pomocí `nc` a nadatlovat byty ručně. U první vlajky je to proveditelné, protože stačí napsat hodně `A`ček za sebou, u druhé vlajky je to horší, neboť asi neexistuje žádné klávesnicové rozvržení, které by umělo napsat nulový byte. Můžeme použít například příkaz `printf` nebo Python k vypsání divných znaků pomocí escapování:

```
\x15\x12\x40\x00\x00\x00\x00\x00
```

Jedno z organizátorských řešení pak používá příkaz `xxd` pro převedení z hexdumpu, což trochu vylepšuje čitelnost:

```
(
echo -n "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
xxd -r <<<"0: 0000 0000 0000 0001" # bool ok
xxd -r <<<"0: 0000 0000 0000 0000" # rbp
xxd -r <<<"0: 1512 4000 0000 0000" # rip
echo
) | nc vm.kam.mff.cuni.cz 13337
```

Celé řešení:
`http://ksp.mff.cuni.cz/viz/36-5-4-solve.sh`

Upgradovat náš exploit můžeme přepsáním do Pythonu. Některé části *payloadu* si tak můžeme nechat vygenerovat nějakou funkcí, což kód učiní trochu čitelnějším. Také získáme trochu lepší kontrolu nad tím, jak a kdy se konkrétní data pošlou na server. Druhé organizátorské řešení používá právě Python a knihovnu `sockets`:

```
s = socket.socket(socket.AF_INET,
                  socket.SOCK_STREAM)
s.connect(("vm.kam.mff.cuni.cz", 13337))
s.recv(len("Zadejte heslo: "))
s.sendall(
    b"a" * 56 +
    0x401215.to_bytes(8, 'little') +
    b"\n"
)
```

Celé řešení:
`http://ksp.mff.cuni.cz/viz/36-5-4-solve.py`

Zde nám Python pomůže v čitelnosti tím, že můžeme opakování `A`ček udělat násobením stringů, také na převedení adresy funkce `print_flag2` na osmibytový little-endian integer můžeme použít funkci `to_bytes`.

Kde nám Pythoní `sockets` trochu práci přidělají, je přijímání výstupu z programu. Funkce `recv`[3] totiž přijímá vždy jen jeden packet, a když se stane, že server na druhé straně odpověď rozdělí do více paketů, musíme `recv` zavolat víckrát. Nejlepší je ho volat ve smyčce, dokud `recv` nevrátí prázdný bytearray, čímž zjistíme, že se druhá strana odpojila.

Ti z vás, kteří se dočetli až sem, si asi říkáte, že je to všechno takové hrozně zdlouhavé. Musíme dlouho rozmýšlet jak

[2] `http://ksp.mff.cuni.cz/viz/kucharky/prace-s-pameti`
[3] `https://docs.python.org/3/library/socket.html#socket.socket.recv`

přesně jsou hodnoty v paměti uložené, abychom se s adresou strefili do správného místa, nebo místo toho zdlouhavě zkoušíme hádat počty `A`ček, než nám to vyjde. Také je to celé velmi neprůhledné, program nám toho moc neřekne, jen občas spadne na segfault a po dlouhém snažení třeba vypíše vlajku. Proto si ještě na konec představíme, jak se dá úloha řešit snadno a rychle pomocí dvou nástrojů navíc – debuggeru `gdb` a Pythoní knihovny `pwntools`.

První trik použijeme k rychlému zjištění, jaký je *offset* adresy v payloadu, tedy, kolik `A`ček musíme napsat před adresu, aby se trefila na správné místo. Použijeme k tomu funkci `cyclic` z pwntools. Tahle funkce nám vygeneruje tzv. De Bruijnovu posloupnost,[4] tedy posloupnost, kde se každé podslovo velikosti n (v našem případě 4) vyskytne právě jednou. Nebudeme chtít posloupnost celou, bude nám stačit pouze její prefix tak dlouhý, aby stačil na způsobení buffer overflow, 100 znaků by mělo stačit. Délka prefixu se podává jako první argument funkci, která se dá také zavolat z příkazové řádky pomocí `pwn cyclic 100`. Vygenerovanou sekvenci si zkopírujeme do schránky.

Následně spustíme náš program v `gdb` zavoláním příkazu `gdb 36-5-4-vuln`. Poté, co se ocitneme v příkazové řádce debuggeru zavoláme příkaz `run` a program se spustí. Ze schránky vložíme připravenou sekvenci a – segfault. Tentokrát program nespadl úplně, ale debugger si ho drží pozastavený ve stavu těsně před pádem, takže se můžeme podívat, jak přesně k pádu došlo. Nejdřív použijeme příkaz `disassemble`, který nám vypíše (úplně stejně jako `objdump` výše) disassembly, navíc se šipkou, která ukazuje na místo v kódu, kde se program zarazil. Šipka ukazuje na instrukci `ret` na konci `main`u, což potvrzuje naši hypotézu uvedenou o pár odstavců výše. Adresa, na kterou se program pokusil skočit, je stále na vrchu zásobníku a můžeme si jí přečíst pomocí příkazu `x/gx $rsp` (`/gx` říká, že cheme **g**iant (64-bitový) integer v he**x**u). Když získanou adresu předhodíme funkci `cyclic_lookup`, nebo opět v příkazové řádce `pwn cyclic -l 0x616161706161616f` dostaneme zpět číslo 56, což je přesně ten offset, který jsme hledali.

Tím to ale se schopnostmi pwntools rozhodně nekončí. Jednou z hlavních funkcí je jednotné rozhraní pro interakci s programem, ať už ho spouštíme jako lokální proces, připojujeme se na server pomocí TCP nebo třeba přes sériový port. Můžeme si tak celý exploit zdebugovat lokálně a pak výměnou jednoho řádku provést exploit na serveru. Knihovna si také umí načíst informace o programu z jeho binárky a podle těch se pak řídit, nebo spustit automaticky `gdb` pro debugging. Nakonec existuje příkaz `pwn template`, který vygeneruje většinu exploitu komplet za vás.

Zde je příklad exploitu napsaného pomocí pwntools:
`http://ksp.mff.cuni.cz/viz/36-5-4-exploit.py`

Všimněte si, že templatovadlo do exploitu automaticky napsalo informace o binárce, například to, že není PIE. Funkce `flat` dostane na vstupu slovník s offsety a daty, která na ten offset patří, a vrátí bytearray, kde jsou prázdná místa vyplněna `cyclic`em. Také automaticky převádí hodnoty do správného formátu, takže adresu předanou jako prostý int `0x401215` automaticky převede do správného formátu 64-bitového little-endian ukazatele. Pro získání adresy samotné se používají *symboly* načtené z binárky. Funkce `interactive` připojí vstup a výstup programu přímo k terminálu, takže dojde mimo jiné k vypsání vlajky.

Podle parametrů příkazové řádky se pak exploit buď připojí na vzdálený server, nebo spustí binárku lokálně. S parametry `LOCAL GDB` pak rovnou v novém okně terminálu otevře `gdb`.

Úplně poslední doporučení na závěr je plugin pro gdb jménem pwndbg,[5] ale objevování jeho výhod je ponecháno jako cvičení. ;)

Úlohu připravili: Honza Černohorský, Adam Kolník, Dan Skýpala

36-5-X1 Superstring

Nejprve se zamyslíme nad tím, jak zkontrolovat, zda se v nějakém řetězci (seně) vyskytují všechna zadaná slova (jehly).

Pořídíme si vyhledávací automat Aho-Corasicková (viz Kuchařka o hledání v textu,[6] případně kapitola 13.3 v Průvodci labyrintem algoritmů). Automatem pak budeme procházet seno a udržovat si množinu jehel, které jsme už viděli (třeba jako posloupnost nul a jedniček).

Jak dlouho to potrvá? Označme n počet slov, m jejich celkovou délku, s délku sena a a počet znaků abecedy. Konstrukce automatu trvá $\mathcal{O}(ma)$, projití sena $\mathcal{O}(s)$ plus $\mathcal{O}(1)$ na každý nahlášený výskyt, kterých je přinejhorším $\mathcal{O}(ns)$.

Seno nicméně neznáme. Vytvoříme si proto graf popisující všechny možné výpočty předchozího algoritmu a najdeme v něm nejkratší úspěšný výpočet.

Konkrétněji: vrcholy grafu budou stavy výpočtu. To jsou uspořádané dvojice (S, M), kde S je stav automatu a M množina nalezených jehel. Pro každý vrchol (S, M) a každý znak Z abecedy sestrojíme hranu, která odpovídá pokračování výpočtu po přečtení znaku Z ze sena – tedy provedení jednoho kroku automatu a rozšíření množiny M o všechny právě nalezené jehly. Tím získáme nový stav a novou množinu, tedy nový vrchol. K vytvořené hraně si zapamatujeme znak Z.

V tomto grafu pak hledáme nejkratší cestu z počátečního vrcholu (*počáteční stav*, $\emptyset$) do libovolného vrcholu, jehož množina obsahuje všechny jehly. Znaky na hranách této cesty nám řeknou hledaný „nadřetězec“.

Rozmysleme si časovou složitost. Automat má $\mathcal{O}(m)$ stavů, množin jehel existuje 2^n. Graf proto má $V = \mathcal{O}(2^n m)$ vrcholů, z každého vede a hran, takže hran je celkem $E = Va = \mathcal{O}(2^n ma)$. Spočítat cíl jedné hrany trvá $\mathcal{O}(m + n)$, protože jeden krok automatu v nejhorším případě proskáče po zpětných hranách až do kořene (což může být až m hladin) a nahlásí až n výskytů, které přidáme do množiny jehel velké $\mathcal{O}(n)$. To vše se dohromady vejde do $\mathcal{O}(m)$, takže celý graf sestrojíme v čase $\mathcal{O}(Em) = \mathcal{O}(2^n m^2 a)$. V tomto čase stihneme i konstrukci automatu a hledání nejkratší cesty.

Tuto složitost jde ještě trochu zlepšit. Jednak si můžeme pro každý stav automatu S a každý znak Z předpočítat, kam nás zavede jeden krok automatu, a tím se v hlavní části algoritmu vyhnout zpětným hranám. To není nic jiného než převedení vyhledávacího automatu na obyčejný konečný automat. Předvýpočet provedeme po hladinách: z kořene vedou všechny kroky zpět do kořene, na každé

[4] `https://cs.wikipedia.org/wiki/De_Bruijnova_posloupnost`
[5] `https://github.com/pwndbg/pwndbg`
[6] `http://ksp.mff.cuni.cz/viz/kucharky/hledani-v-textu`

další hladině pak znak Z buď má svou dopřednou hranu, nebo z něj automat jde zpětnou hranou do stavu na vyšší hladině, kde už máme předpočteno. To stihneme ve stejném čase $\mathcal{O}(ma)$, jaký nás stálo vytvořit automat.

Také můžeme množiny reprezentovat bitovými maskami. Na to potřebujeme předpokládat, že náš počítač pracuje s n-bitovými čísly v konstantním čase, ale to je docela opodstatněný předpoklad u algoritmu s exponenciální složitostí. S bitovými maskami pak běžné množinové operace (přiřazení, průnik, sjednocení, rozdíl, dotaz na přítomnost prvku) běhají v konstantním čase.

Nakonec si můžeme pro každý stav automatu předpočítat množinu nalezených jehel, opět reprezentovanou bitovou maskou. Tento předvýpočet opět provedeme po hladinách v čase lineárním s velikostí automatu.

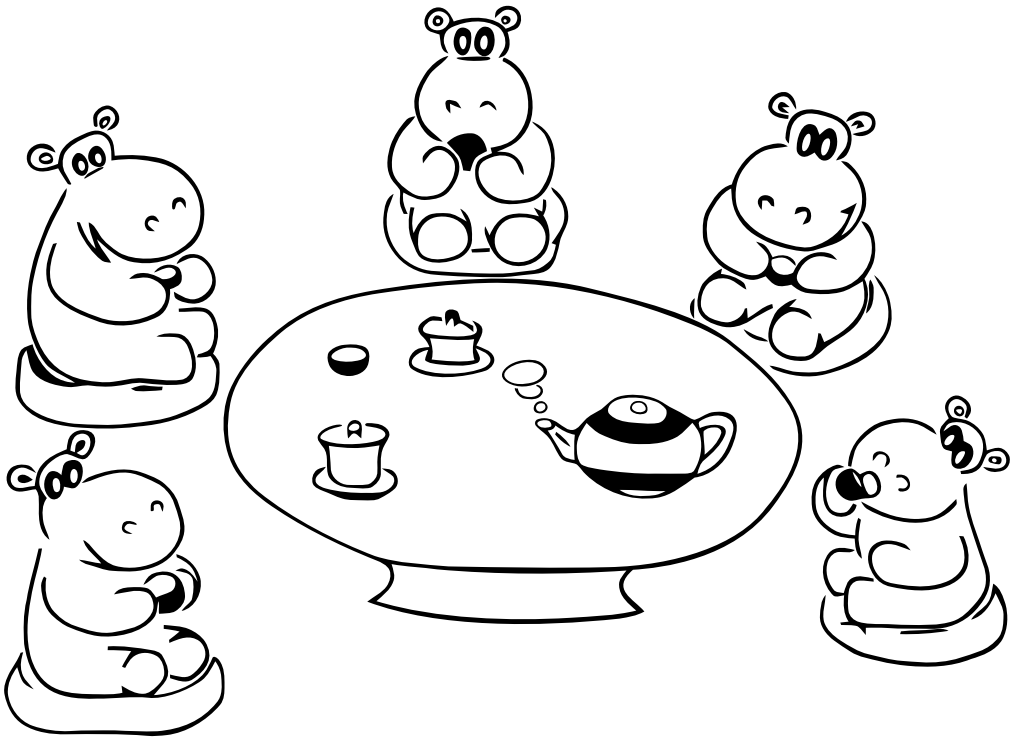
S těmito třemi triky můžeme cíl jedné hrany nalézt v konstantním čase, takže se konstrukce grafu zrychlí na $\mathcal{O}(E) = \mathcal{O}(2^n ma)$. Do tohoto času se zase vejde jak konstrukce automatu, tak prohledání grafu.

Úlohu připravil: Martin „Medvěd“ Mareš

36-5-S Učení bez učitele

Vzorová řešení všech dílů seriálu vydáme během prázdnin.

Úlohu připravil: Michal Kodad



Závěrečná výsledková listina 36. ročníku KSP

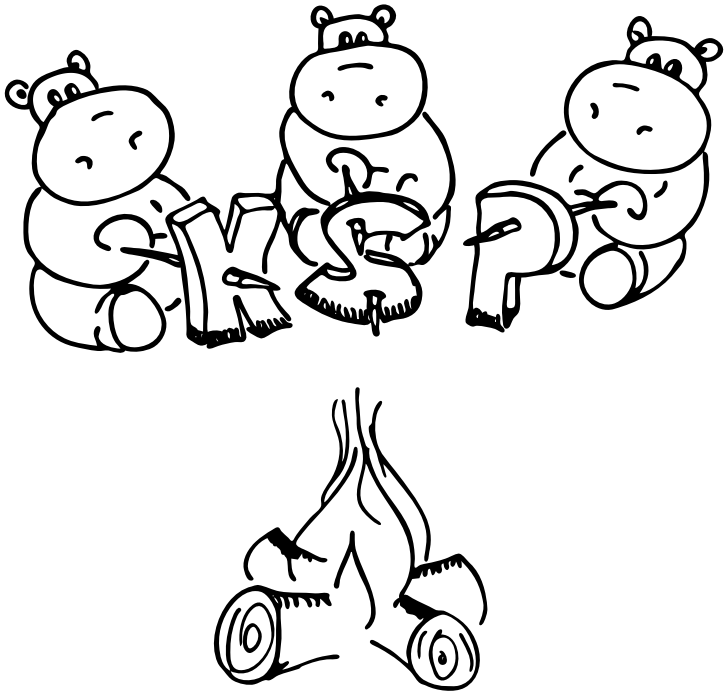
	řešitel	*škola*	*ročník*	*sérií*	*5-1*	*5-2*	*5-3*	*5-4*	*5-S*	*5-S2*	*5-X1*	*série*	*KSP-X*	*celkem*
0.					10	12	12	11	12	3	10	60,0	40,0	300,0
1.	Jáchym Kouba	GJŠkodyPŘ	4	15	10	12	10	11	15	3		61,0	4,0	292,5
2.	Patrik Prítrský	GGrössBA	3	10	10	12	12	11	13	3		61,0	5,0	292,0
3.	Anna-Kristina Migel	SPŠSmíchov	1	10	10	12	10	11	8	3		54,0	0,0	265,5
4.	Petr Starý	GJírovcČB	2	10	10	12	10	4	15,5	3		54,5	0,0	252,0
5.	Ondřej Sedláček	GOPavla PH	3	10	10	0	10	11	10	3		44,0	0,0	247,5
6.	Richard Dobíšek	MensaG	3	8	10	12	10	4				36,0	0,0	227,0
7.	Jan Hrubec	GOpenGaBab	1	5	10	1	10	11	8	3		43,0	0,0	201,5
8.	Albert Bakoč	GZborovPH	3	14	6	12	10				9	28,0	14,0	199,5
9.	Erik Ježek	SPŠSmíchov	2	10	10		10					20,0	18,0	180,0
10.	Adam Houdek	SOŠ Březová	−1	9	10	12	12					34,0	6,0	173,5
11.	Lukáš Snížek	SOŠ Březová	3	5				4				4,0	0,0	168,0
12.	Vladimír Sklenár	HydeS USA	4	14	10			4				14,0	1,0	167,0
13.	Petr Ševček	KarlínG	4	4								0,0	0,0	166,0
14.–15.	Jonáš Doležal	ParkLane	3	5	10	1	10	11	11,5			43,5	0,0	165,5
	Luka Králík	GNZatlanPH	3	5	10	0	10	11	10	3		44,0	0,0	165,5
16.	David Hromádka	GNAlejíPH	3	5	10	12		4				26,0	0,0	159,0
17.	Vojtěch Vlachovský	SPŠ MB	4	4								0,0	0,0	155,5
18.	Ondřej Nečas	GLesníZlín	4	4								0,0	0,0	154,0
19.	Antonín Maloň	GJarošeBO	4	9	10	12						22,0	17,0	152,0
20.	Alexandr Bihun	GJírovcČB	4	6	10			4				14,0	0,0	149,5
21.	Michael Ambros	GTomkovaOL	1	7	10				13	3		26,0	0,0	144,0
22.	Daniel Čech	GBezručeFM	4	5								0,0	0,0	133,0
23.	Martin Šindelář	GGrössBA	4	8								0,0	6,0	132,0
24.–25.	Daniel Culliver	GZborovPH	4	11								0,0	0,0	122,5
	David Surga	ParkLane	3	5	0							0,0	0,0	122,5
26.	Matúš Púll	GZborovPH	4	13	10			11				21,0	0,0	115,0
27.	Erik Sabol	GČeskoliPH	4	10	10			4				14,0	0,0	110,0
28.	Lucian Poljak	GJŠkodyPŘ	2	10	10		10					20,0	0,0	107,0
29.	Lukáš Franta	GZborovPH	2	5	0							0,0	0,0	105,0
30.	Jakub Prítrský	GŠtefánNM	2	3								0,0	7,0	94,5
31.	Filip Maňas	GJarošeBO	3	7	0	12		4				16,0	0,0	90,5
32.	Matěj Hošek	GVolgogrOS	2	4								0,0	0,0	90,0
33.–34.	Lucie Roskovská	GOhradníPH	3	3	6							6,0	1,0	84,5
	Vlastimil Vít	GZborovPH	4	2								0,0	3,0	84,5
35.	Filip Sichrovský	GČesLípa	2	5				4				4,0	0,0	81,0
36.	Martin Vagner	GVoděraPH	1	5				11				11,0	0,0	80,5
37.	Maximilián Ján Furman	GLS Bard	4	2								0,0	0,0	68,5
38.–39.	Michael Jarvis	GŠpitálsPH	2	6								0,0	0,0	64,5
	Matyáš Martinek	GLesníZlín	4	3								0,0	0,0	64,5
40.	Áron Kálmán	GGrössBA	4	3								0,0	0,0	61,0
41.	Ondřej Pupík	GRožnovPR	4	11								0,0	0,0	59,0
42.	Svatava Šimečková	GJarošeBO	2	4								0,0	0,0	58,5
43.	Tomáš Mácha	ArcibisGPH	1	2								0,0	0,0	57,5
44.	Viliam Gottweis	GGrössBA	2	2								0,0	0,0	55,0
45.	Jakub Hampl	GMělník	4	10	10			11				21,0	0,0	51,0
46.	Finley Stuart	GPísnickáPH	3	8								0,0	0,0	45,5
47.–48.	Tomáš Kodaj	GGrössBA	2	2								0,0	0,0	43,0
	Michal Mentzl	G UherBrod	4	2								0,0	0,0	43,0
49.	Jan Koška	GJírovcČB	4	1								0,0	0,0	40,5
50.	Samuel Dembinný	SPŠ Kladno	3	2								0,0	0,0	39,5
51.	Martin Vanko	GKepleraPH	3	3								0,0	0,0	36,5
52.	Kateřina Vomelová	GÚstavníPH	4	9								0,0	0,0	34,0
53.	Jáchym Löwenhöffer	GEvolutionJM	3	6	0		10					10,0	0,0	33,0
54.	Darek Hancko	GJSzŠahy	4	2								0,0	0,0	27,0
55.–56.	Jiří Novák	JazG HK	3	3								0,0	0,0	26,0
	Jakub Svatuška	ArcibisGPH	3	3								0,0	0,0	26,0

	řešitel	*škola*	*ročník*	*sérií*	*5-1*	*5-2*	*5-3*	*5-4*	*5-S*	*5-S2*	*5-X1*	*série*	*KSP-X*	*celkem*
57.	Vít Kaděra	G Wicht	2	7								0,0	0,0	25,0
58.	Jakub Dobiáš	GZborovPH	3	1								0,0	0,0	24,5
59.	Martin Madzin	G Stropkov	4	1								0,0	0,0	23,0
60.–61.	Petr Laškevič	GNAlejíPH	3	2				11				11,0	0,0	22,0
	Vít Mitáš	GPolička	2	6								0,0	0,0	22,0
62.	Jan Václavek	GJarošeBO	2	1			10	11				21,0	0,0	21,0
63.–64.	Dominik Dembinný	ZŠMR Kladno	0	1								0,0	0,0	20,0
	Jan Ševeček	G UherBrod	2	5								0,0	0,0	20,0
65.	Tobiáš Halíř	GVoděraPH	3	2								0,0	0,0	17,5
66.–67.	Tomáš Kraus	KřesťGPH	2	4	0							0,0	0,0	17,0
	Ondřej Nevěřil	GZábřeh	2	2								0,0	0,0	17,0
68.	David Pacák	G Brandýs	3	5								0,0	0,0	16,5
69.	Ladislav Jareš	SPŠ MB	4	2								0,0	0,0	14,0
70.–71.	Adam Handl	GBenesov	2	2	10	0						10,0	0,0	13,0
	Miroslav Kolouch	GJírovcČB	4	3				4				4,0	0,0	13,0
72.–74.	Honza Kocourek	ParkLane	4	7				11				11,0	0,0	11,0
	Roman Pochylý	GUHradiště	4	1								0,0	0,0	11,0
	Petr Šácha	GUHradiště	4	1								0,0	0,0	11,0
75.–76.	Filip Mach	GZborovPH	2	1								0,0	0,0	10,0
	Ivan Žemlička	GÚstavníPH	3	4								0,0	0,0	10,0
77.	Vilem Ucik	GJungmanLT	3	1								0,0	0,0	8,5
78.–80.	Tereza Písková	GZborovPH	4	1								0,0	0,0	8,0
	Nicolas Skuček	1.ITGPH	3	1								0,0	0,0	8,0
	Jakub Zahradník	GNAlejíPH	4	1								0,0	0,0	8,0
81.	Jonáš Menšík	GJŠkodyPŘ	2	3								0,0	0,0	7,0
82.	Ondřej Novák	G Brandýs	2	6								0,0	0,0	6,0
83.–84.	Petr Šišlák	GZborovPH	3	2								0,0	0,0	5,0
	Matyáš Zahradník	GNAlejíPH	4	1								0,0	0,0	5,0
85.–86.	Daniel Linda	SJec	3	1								0,0	0,0	4,0
	Jan Slíva	MensaG	3	10				4				4,0	30,0	4,0
87.–92.	Adam Baborák	GRokycany	2	1								0,0	0,0	3,0
	Matěj Bajgar	GJírovcČB	1	1								0,0	0,0	3,0
	Filip Jarolím	G Wicht	2	2								0,0	0,0	3,0
	Vojtěch Kosina	G Chrudim	3	2								0,0	0,0	3,0
	Elias Mocik	GJHroncaBA	3	1								0,0	0,0	3,0
	Štěpán Sedmík	SPŠSmíchov	4	1								0,0	0,0	3,0
93.	Ricardo Bolemant	GŠahy	3	2								0,0	0,0	2,0
94.–99.	Filip Dvořák	GMilevsko	1	1								0,0	0,0	1,0
	Aleš Gabrhelík	SŠTEBrno	3	1								0,0	0,0	1,0
	Ondřej Hrabě	GBenesov	4	1								0,0	0,0	1,0
	Kryštof Maxera	GJírovcČB	3	8								0,0	0,0	1,0
	Mikuláš Pater	ZŠUT	0	1								0,0	0,0	1,0
	Jaroslav Picek	BiGyBBHK	4	1								0,0	0,0	1,0

Závěrečná výsledková listina 36. ročníku KSP-X

	řešitel	*škola*	*ročník*	*sérií*	*1-X1*	*3-X1*	*4-X1*	*5-X1*	*celkem*
0.					10	10	10	10	40,0
1.	Jan Slíva	MensaG	3	10	10	10	10		30,0
2.	Erik Ježek	SPŠSmíchov	2	10	8	10			18,0
3.	Antonín Maloň	GJarošeBO	4	9	7	10			17,0
4.	Albert Bakoč	GZborovPH	3	14	5			9	14,0
5.	Jakub Prítrský	GŠtefánNM	2	3	7				7,0
6.–7.	Adam Houdek	SOŠ Březová	−1	9	6				6,0
	Martin Šindelář	GGrössBA	4	8			6		6,0
8.	Patrik Prítrský	GGrössBA	3	10	5				5,0
9.	Jáchym Kouba	GJŠkodyPŘ	4	15	4				4,0
10.	Vlastimil Vít	GZborovPH	4	2	3				3,0
11.–12.	Lucie Roskovská	GOhradníPH	3	3	1				1,0
	Vladimír Sklenár	HydeS USA	4	14	0	1			1,0

Bonusové úlohy z jednotlivých sérií se nepočítají do bodování ročníku. Mají svou vlastní výsledkovou listinu a za jejich úspěšné vyřešení (alespoň polovina bodů za úlohu) udělujeme speciální odměny.



matfyz

KSP pro vás připravují studenti Matematicko-fyzikální fakulty Univerzity Karlovy.
Realizace projektu byla podpořena Ministerstvem školství, mládeže a tělovýchovy.

Webové stránky: *https://ksp.mff.cuni.cz/*
E-mail: *ksp@mff.cuni.cz*
Organizátoři a kontakty: *https://ksp.mff.cuni.cz/kontakty/*