

Korespondenční Seminář z Programování

ZAČÁTEČNICKÁ KATEGORIE

28. ročník

KSP-Z

Listopad 2015

První série letošního KSP-Z je za námi. Doufáme, že se vám úlohy libily, a zde vám přinášíme jejich autorská řešení. Pokud byste k čemukoliv měli nějaké dotazy, neváhejte se nás ptát na našem fóru. Porušilo nás, že se tolik z vás s verzou pustilo do řešení. Pokud se vám z nějakých úloh nepodařilo získat plný počet bodů, nebo jste při jejich řešení úplně tápali a raději ne neposlali, podívejte se na vzorová řešení.



Řešení první série začátečnické kategorie 28. ročníku KSP

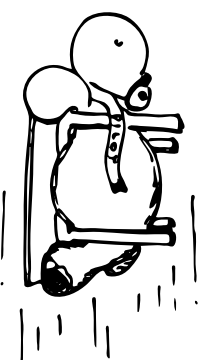
28-Z1-1 Kevinův leták

Nejprve si vyřešíme jednodušší úlohu: jak o nějaké posloupnosti závorek poznáme, jestli je (celá) správně uzávorkovaná? Určitě musí obsahovat stejný počet levých a pravých závorek – jinak těžko mohl tvořit páry.

Ovšem to nestačí. Uvažte třeba posloupnost $()()$. Ta obsahuje dvě levé a dvě pravé závorky, ale korektním uzávorkováním určitě není. Na třetí pozici se pokoušíme ukončit závorku „dřív, než začala“.

Jak takovou situaci poznáme? Podívráme-li se na první tři znaky naší posloupnosti, vidíme, že obsahují jednu levou závorku a dvě pravé. Obecně problém nastane, pokud existuje v posloupnosti nějaké místo takové, že v úseku od začátku po toto místo je více pravých závorek než levých. Pak ale spory jedna z těch prvních nemá před sebou žádnou levou, se kterou bychom ji mohli spárovat.

Pokud posloupnost obsahuje stejně levých a pravých závorek a nenastane problém popsaný výše, pak už si snadno rozmyslíte, že je vždy jde správně spárovat, a tedy uzávorkování je korektní.



Nyní k původní úloze. Představme si, že si chceme označit všechna místa v posloupnosti, kde může končit korektní uzávorkování (zakínající na začátku). Třeba pro příklad ze zadání to budou následující místa (označena hvězdičkou): $((())*(O*)()$.

Jak je najít? Budeme postupně procházet posloupnost od začátku a průběžně si počítat, kolik levých a pravých závorek už jsme viděli. Pokud narazíme na místo, kde jsme napočítali víc pravých než levých, můžeme rovnou skončit. Už víme, že takové uzávorkování korektní není, a příštím libovolných dalších závorek na konec už nemižeme napravit chybějící závorky nalevo od aktuální pozice.

Pokud tato situace ještě nastala a objevíme místo, před kterým je počet levých a pravých závorek stejný, označíme si jej. Podle toho, co jsme si řekli výše, uzávorkování končící na tomto místě je korektní.

Ukážme si průběh algoritmu na předchozím příkladu:

	(()	())	())	())
Pozice	1	2	3	4	5	6	7	8	9	10	11	
Počet levých	1	2	2	3	3	3	4	4	4	–	–	
Počet pravých	0	0	1	1	2	3	3	4	5	–	–	
Rozdíl	1	2	1	2	1	0	1	0	–1	–	–	
Akce					*		*	*	*	*	X	

Hvězdička znamená označení místa s korektním uzávorkováním a X konec prohledávání, když žádné další uzávorkování korektní být nemůže.

Na konci jen vypíšeme nejpravější označené místo. Všimněte si, že si nemůžeme ukládat všechna označená místa, stačí si vždy pamatovat poslední dosud nalezené. Další dvojnásobně zjednodušení je místo dvou počítadel použít jen jedno, uchovávející rozdíl počtu dosud načtených levých a pravých závorek (řádek „rozdíl“ v tabulce výše). Pak označujeme, když je toto počítadlo rovné nule, a končíme, pokud klesne pod nulu.

Program (Python 3) - počítání obou druhů závorek:
`http://ksp.mff.cuni.cz/viz/28-Z1-1-1.py`

Program (Python 3) - stačí nám počítat jen levé:
`http://ksp.mff.cuni.cz/viz/28-Z1-1-2.py`

Filip Stěhorský

28-Z1-2 Sářina hra

Na vstupu dostaneme čísla N a M a máme zjistit, jestli po hře $M-2$ kol hry bude zastávka číslo N zakroužkována (kol je $M-2$, protože Sára začne kroužkovat až ve druhé zastávce – v prvním kole kroužkuje zastávky dělitelné 2 a v posledním zastávky dělitelné $M-1$).

Pomale řešení

Nejjednodušší řešení by bylo vytvořit si pole zakroužkovanos N příznaků, kde příznak zakroužkováno [1] značí, je-li i -tá zastávka zakroužkována. V tomhle poli ($M-2$)-krát nasimulujeme kolo Sářiny hry a nakonec přečteme příznak u N -té zastávky, čímž zjistíme, jestli skončila zakroužkována, nebo ne.

V Pythonu to jde napsat třeba takhle:

```
# N + 1, protože Python čísluje pole od nuly
zakrouzkovano = [False for _ in range(N + 1)]
for kolo in range(2, M):
    # Zakroužkuj každou k-tou zastávku.
    for zas in range(kolo, N + 1, kolo):
        zakrouzkovano[zas] = not zakrouzkovano[zas]
print("ANO" if zakrouzkovano[N] else "NE")
```

Jak dlouho to poběží? Utrčíte aspoň $O(N)$ – první kolo Sádky hry zakrouzkujte aspoň $\lfloor N/2 \rfloor$ zastávek, kterým potřebujeme upravit příznaky.¹

Kol je ale víc: první kolo potřebuje $N/2$ kroků, druhé $N/3$, třetí $N/4$, a tak dále. Kol je celkem M a každé potřebuje nejvyšší N kroků, proto nebude naše řešení potřebovat více než $O(N \cdot M)$ kroků. Pomalé řešení tedy poběží něco mezi $O(N)$ a $O(N \cdot M)$. (Přesná časová složitost je $O(N \log M)$, ale na ni teď příliš nezáležejí. Chcete-li zjistit, kde se vzalo $N \log M$, podívejte se do sekce o Eratosthenově síti v kapitole o teorii čísel.)²

Timhle zvládneme první dva malé vstupy, ale větší nestihneme spočítat.

Druhý pokus

Zakrouzkování žádne zastávky kromě N -té nás ale vlastně vůbec nezajímá, výsledek programu na nich nezáleží. Udržování příznaků, jestli jsou zastávky $2, \dots, N-1$ zakrouzkované, je plynutí času a paměti. Místo toho tedy projedeme kola $2, \dots, M-1$ a pro každé z nich se podíváme, jestli změna zakrouzkovanosti N -té zastávky. Krouzkování každé k -té zastávky změna zakrouzkovanosti zastávky N právě tehdy, když je N dělitelné bez zbytku k , neboli (v Pythonu) $N \% k == 0$.

```
# Je N-tá zastávka zakrouzkovaná?
zakrouzkovana = False
for kolo in range(2, M):
    if N % kolo == 0:
        zakrouzkovana = not zakrouzkovana
print("AMN" if zakrouzkovano else "NE")
```

Tohle řešení poběží v čase $O(M)$, a to je určitě zlepšení proti prvnímu pokusu. Na plný počet bodů ale ještě nestačí.

Řešení za všechny body

Každá změna zakrouzkovanosti N -té zastávky odpovídá tomu, že číslo kola k dělí N . Zastávka N je na konci zakrouzkovaná, pokud počet změn zakrouzkovanosti byl liché, jinak zakrouzkovaná není. Úloha se nás tedy vlastně ptá, jestli má N sudé, nebo liché množství dělitelů, kteří jsou menší než N a větší aspoň 2.

K dalším zrychlení si všimneme, že dělitelé se vyskytují v párech: ke každému k_1 , které je dělitelem N , existuje nějaké k_2 takové, že $N = k_1 \cdot k_2$. Tohle k_2 je taky dělitelem N . Dále v každém páru (k_1, k_2) je aspoň jedno k menší nebo rovné \sqrt{N} : kdyby byly obě větší, tak $k_1 \cdot k_2$ by muselo být víc než N , ale dohodli jsme se, že to bude přesně N .

Když nás tedy zajímají všichni různí dělitelé N , můžeme je generovat tak, že najdeme všechna k_1 mezi 1 a \sqrt{N} a ke každému dopočítáme k_2 . Pro každého z dělitelů můžeme zjistit, jestli menší než M a aspoň 2, a jestli je, přepneme za něj zakrouzkovanost N -té zastávky. Zbývá doložit ještě jeden detail: když N je k_1^2 , tak $k_2 = k_1$. V takovém případě musíme přepnout zakrouzkovanost N -té zastávky jenom jednou. Tentokrát řešení dobehne za $O(\sqrt{N})$, a to už si zaslouží 10 bodů.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/28-21-2.py`

Michal „Prvák“ Pokorný

28-21-3 Petrovy stromy

Zadání sice mohlo vypadat děsivě kvůli množství proměnných, které se v něm vyskytují, ve skutečnosti je ale prosté. Podél cesty se nám pravidelně střídají stromy a nás zajímá, kolik stromů se silně prostrídá, než uvidíme K -tý strom konkrétního druhu. V řešení budeme mluvit o topolech, ale myslíme tím prostě strom druhu J .

Připomejme ještě označení B pro délku bloku, který se nám pravidelně opakuje.

Přímocaré řešení je uložit si celý opakující se blok do pole. Následně můžeme toto pole procházet a pamatovat si, kolik topolů a kolik stromů celkem jsme už viděli. Až potkáme K -tý topol, zahášíme výsledek.

Takové řešení má ale složitost až $O(BK)$: to když bude v celém bloku jeden jediný topol. Jelikož vstup má velikost $O(B)$ a K může být mnohem větší než B , můžeme pojmnout podzvězení, že to musí jít rychleji.

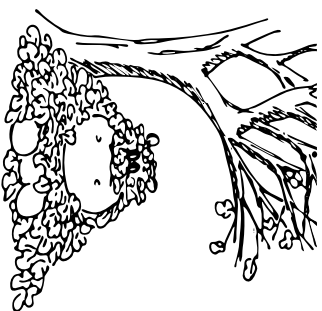
Jde, a rychlejší řešení není nijak těžké: pojďme na něj. Nejprve uvidíme nějaký počet opakujících se bloků, pak ještě část stromů z dalšího bloku a pak ten očekávaný K -tý topol.

Kolik bude bloků, které uvidíme celé? Tolik, aby se do nich spolehlivě vešlo $K-1$ topolů, takže $(K/S) - 1$, kde lomítko představuje celočíselné dělení (zbytek zahodíme) a S je počet topolů v bloku.

A kolik stromů uvidíme z dalšího bloku? Nejprve jiná otázka: kolik topolů ještě uvidíme? Tolik, kolik jich chybí do K . To většinou bude $K \bmod S$, kde „mod“ označuje zbytek po dělení. Ono slovo „většinou“ je v předchozí větě pro případ, kdy K je násobkem S , pak nám topolů chybí nikoliv 0, ale S .

Dejme tomu, že nám tedy do K chybí ještě T topolů. To ale znamena, že uvidíme tolik stromů, na kolikáté pozici je v bloku T -tý topol. Řekneme příkladem, pokud jsou třeba topoly na pozicích 2, 4 a 9 a chybí nám 2 topoly, uvidíme ještě 4 stromy, protože druhý topol je na pozici 4.

Zpočátku samozřejmě neznáme ani S , ani pozice jednotlivých topolů. Obojí ale můžeme jednoduše zjistit při načítání vstupu. Pamatujme si, kolikátý strom právě načítáme, a taky si udržujeme informaci o počtu už načtených topolů. Když máme na vstupu topol, zvýšíme počet topolů a zároveň si uložíme jeho pozici do pole.



¹ Ω je přibližná známějšího O . Když řekneme, že nějaká funkce g je v $O(f(x))$, znamená to, že se stoupajícím x neroste rychleji, než nějaký násobek $f(x)$, čili „ g se dá shora ohraničit f “. Ω oproti tomu značí dohu odhadu: když g je v $\Omega(f(x))$, tak g roste se stoupajícím x aspoň tak rychle, jako f .

² `http://ksp.mff.cuni.cz/viz/kucharky/teorie-cisel`

Na konci pak jen využijeme tyto informace k dosazení a máme výsledek. Zbývá poslední věc, a to je složitost řešení. Musíme načíst vstup, to zvládneme v $O(B)$ (i když načítáme topol, provádíme jen konstantně mnoho operací). Závěrečné spočítání výsledku stihneme v konstantním čase, máme tedy celkovou časovou složitost $O(B)$. Paměťová složitost je $O(S)$, protože si musíme pamatovat pozice pro jednotlivé topoly.

Program (C):

```
http://ksp.mff.cuni.cz/viz/28-21-3.c
```

Karolína „Karygamma“ Burešová

28-Z1-4 Zúčtna zvědavost

Zadáni trochu křkolomně popisovalo graf, ve kterém hrany tvoří dvojice žáků, kteří se seznamovali. V tomto grafu jste měli najít počet komponent souvislosti. Pokud předchozí větvě rozumíte, asi se zde nic nového nedozvíte. Jinak velmi doporučíuji přečíst si naši kuchařku o grafech.³

Pokud se grafových zvířátek zatím bojíte, zkusím popsat řešení v jazyce zadání. Nejprve si celé zadání načteme do paměti. Nejlépe tak, že si pro každého žáka pořídíme seznam ostatních žáků, se kterými se seznámil. Nezapomněte, že pro každou dvojici musíme přidat dva záznamy – lidé se seznamují vzájemně.



Nyní budeme postupně procházet všechny žáky a pokud někoho ještě neznáme zařazeného v žádné třídě, vytvoříme pro něj novou (zvyšíme počítadlo tříd). Potom budeme procházet žáky, se kterými se seznámil, a u všech si označíme, že už jsme je zadrželi. Tuhle operaci musíme provádět rekursivně, tj. označit si i sousedy sousedů, sousedy sousedů sousedů a tak dále.

Prakticky si pořídíme funkci označ(Ž), která označí žáka Ž, a zavolá sama sebe na všechny spolžáky Z. Volání sebe sama se říká rekurze. Důležité ale je, aby někdy přestala – pokud už je žák označený, funkce ihned skončí a nic nedělá. Popsáním algoritmu se říká prohlédávání do hloubky; více o něm a jeho aplikacích najdete v kuchařkách.

Nejprve si pojďme uvědomit, kolik potřebujeme paměti. Pro uložení seznamů spolžáků potřebujeme N seznamů velikých nejvýše N – ale budou mít dohromady 2N prvků. Protože ale nevíme, které z čísel je větší, řekneme, že na tuto část potřebujeme $O(N + M)$ paměti. Při dodržení podmíněk ze zadání bude $M < N^2$, ale může být i řádově menší.

Na uložení příznaků označení nám stačí N proměnných, pak už potřebujeme jen pár počítadel. Dohromady se tedy vejde do $O(N + M)$ paměti.

Časová složitost vyjde stejně. Při procházení si stačí uvědomit, že po každé dvojici projdeme jen dvakrát (jednou z každé strany), a jakmile třídu uzavřeme, už se na žádného žáka z ní nepodíváme. Třída s jedním žákem je ale také třída a prohlédávání, které hned skončí, musíme také započítat. Všechna prohlédávání dohromady spotřebují $O(N + M)$ času – $O(N)$ za žáky, $O(M)$ za dvojice.

Dodejme jen, že v některých jazycích může být volání funkce zbytečně náročná operace a může čítrt příliš mnoho pa-

mětí. Potom se hodí rekurzi nahradit cyklem a žáky si ukládat do zásobníku „růčete“. Co je to zásobník a jak ho použít se dočtete ve výše zmíněné kuchařce.

Pokud místo zásobníku použijeme frontu, dostaneme prohlédávání do šířky, které používá vzorové řešení v Pythonu. Jinak jsou obě možnosti ekvivalentní.

Program (C++):

```
http://ksp.mff.cuni.cz/viz/28-21-4.cpp
```

Program (Python 3):

```
http://ksp.mff.cuni.cz/viz/28-21-4.py
```

Ondra Hlavatý

28-Z1-5 Dvě fronty na oběd

Jsem v jídelně a máme tu dvě fronty hladových studentů seřazených podle výšky. Potřebujeme, aby se spojili v jednu frontu (také seřazenou). Jak se ale mají žáci co nejrychleji stromat?

Některí z vás možná poznali, že se vlastně jedná o „sklěvací“ funkci merge z algoritmu MergeSort, o kterém si můžete více přečíst v naší kuchařce o třídění.⁴ A pokud je to pro vás nové, pojďme si ukázat, jak na to.

První žák v nové frontě musí být nejmenší ze všech. To znamená, že to bude buď první žák první fronty, nebo první žák druhé fronty. Tito dva jsou nejmenší ve své skupince (která je seřazená), tudíž určité nikdo za nimi není menší.

Porovnáme jejich výšku a menšího pošleme do nové fronty. Možná by vás mohlo napadnout poslat do nové fronty oba, ale to udělat nemůžeme. Žák druhý v pořadí může být křidně menší než první žák z jiné fronty. Posíláme tedy vždy jen jednoho.

U jedné skupinky se tak druhý nejmenší stál nejmenším a toho opět porovnáme s nejmenším z druhé fronty. Pokud by porovnávání žáci byli stejně vysokí, můžeme poslat libovolného, nebo dokonce oba dva (ani u jedné frontě není nikdo menší než oni).

Vždy tedy porovnáme dva aktuálně nejmenší žáky a menšího z nich pošleme na konec nové fronty. Když už bude jedna fronta prázdná, můžeme zbytek té druhé poslat do nové fronty tak, jak je (žáci jsou už seřazení).

Než vám prozradíme časovou a paměťovou složitost, je tu jeden implementační detail. Poslat žáka na konec nové fronty znamená, že informaci pouze předkopírujeme. „Odebratého“ žáka nemůžeme, jenom se ve frontě podíváme na dalšího. Mázat data (a tím všechna ostatní posouvat) by bylo zbytečné a neefektivní.

Každým porovnáním vybereme jednoho žáka, kterého pak pošleme na konec nové fronty. Pokud bylo v první skupince A žáků a ve druhé B, potom bude v nové frontě A+B žáků. Takže provedeme maximálně A+B – 1 porovnání, než to bude, pokud po vyprázdnění jedné fronty zbudete ve druhé více než jeden žák či pokud jsou porovnávání stále stejně vysozí a my je pošleme do fronty oba. Ale nás zajímá nejhorší případ, takže časová složitost bude $O(A + B)$.

Program (Python 3):

```
http://ksp.mff.cuni.cz/viz/28-21-5.py
```

Katka Zákrevská © Zúčka Dvřáková

³ <http://ksp.mff.cuni.cz/viz/kucharcky/grafy>

⁴ <http://ksp.mff.cuni.cz/viz/kucharcky/trideni>

28-Z1-6 Osm čípnů

Kevin si zvolí tři čísla: a, b a c . Když máme spočítat číslo k uložené do i -tého čípu, použijeme vzorec $a \cdot x_i - 3 + b \cdot x_i - 2 + c \cdot x_i - 1$ a spočítáme, kolik vyjde po vydělení výsledku číslem N^3 . Dostaneme zadaná čísla prvních tří čípnů x_1, x_2 a x_3 a nějaké hodnoty velké K a snažíme se co nejrychleji spočítat K .

Nejhdodnější by bylo vyrobit si pole o K prvích, ve kterém budeme držet hodnotu každého čípu. První tři hodnoty do něj uložíme rovnou a zbytek dopočítáme.



Takové řešení potřebuje $O(K)$ paměti na uložení pole a bude trvat čas $O(K)$. Mý ale vlastně nepotřebujeme znát všech K čísel ve všech čípech; zajímá nás jenom to poslední. A když počítáme číslo K -tého čípu, stačí nám k tomu čísla čípnů $(K-1), (K-2)$ a $(K-3)$. Podobně když počítáme číslo v čípu $(K-1)$, potřebujeme znát jenom čísla čípnů $(K-2), (K-3)$ a $(K-4)$.

Obecně si stačí pamatovat čísla jenom tři kroky dozadu – díky tomu naše spotřeba paměti bude konstantní, nezávisle na K . Protoněme, ve kterých budeme držet poslední tři čísla čípnů, si označme jako x, y a z . Můžeme třeba ve for-cyklus $(K-1)$ krát upravit hodnoty proměnných (x, y, z) na $(y, z, (a \cdot x + b \cdot y + c \cdot z) \bmod N)$.

```
# V x, y, z si teď budeme držet čísla
# posleďních 3 čípnů.
x, y, z = x1, x2, x3
for i in range(K - 1):
    x, y, z = y, z, (a * x + b * y + c * z) % N
print(x)
```

V zadání bylo napsáno, že do K nestihneme napočítat po jedné – to znamená, že cyklus provádějící K kroků bude příliš pomalý. Ukážeme si dva způsoby, které jsou rychlejší. Když se pozorně podíváte na zadání, všimnete si, že se čísla po nějaké době opakují. V zadání je předěl mezi opakovanými zvýrazněni větší mezerou. Pokud bychom znali délku opakování se úseků p , můžeme skoro s jistotou říci, že K -tý čípn má stejné číslo, jako $(K \bmod p)$ -tý. Číslo $K \bmod p$ bude určitě menší než p , takže bychom to stihnout mohli. Nejpřevratnější pozorování: přestože zadání neslabilo, jak velká budou x_1, x_2 a x_3 , můžeme místo nich vzít rovnou jejich zbytek po dělení N . Pokud $N \bmod N = f$, pak existuje i takové, že $a = i \cdot N + j$. Potom $(a \cdot x) \bmod N = (i \cdot N \cdot x) \bmod N + (j \cdot x) \bmod N = 0 + (j \cdot x) \bmod N$. Cožoliv, co násobíme N , bude N dělitelné, tedy bude mít zbytek po dělení N nula.

Další důležitá myšlenka plyne už z odstavce výše. K určení čísla i -tého čípu nám stačí čísla třech čípnů dozadu. Pokud je tedy trojice čísel před čípnem i stejná jako před čípnem j , musí být x_j stejné číslo jako x_i . A díky tomu se rovnají i x_{j+1} s x_{i+1} . A tak dále. Jakmile tedy najdeme stejnou trojici podmnutě, nutně už musíme být v opakujícím se bloku, v *periodě*.

Hledání periody trochu komplikuje to, že posloupnost čísel může mít předperiody – opakující se úsek nemusi začínat na začátku posloupnosti.

Například pro $(a, b, c) = (0, 1, 1)$, $N = 2$ posloupnost může vypadat takto:

1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, ...

Všimněte si, že kromě prvních se tři jedničky za sebou v posloupnosti už neobjeví. Jak se tedy s předperiodou vypořádá? Pořídíme si trojrozměrné pole čísel, v každém rozměru velké N . Do něj si budeme označovat trojice po sobě jdoucích čísel čípnů a číslo kroku i , ve kterém jsme danou trojici potkali. Jakmile narazíme na stejnou trojici podmnutě v kroku j , našli jsme periodu dlouhou $j - i$ s předperiodou délky i .

Když známe délku periody p a předperiody q , číslo K -tého čípu spočítáme hloupejším algoritmem výše, který spustíme pro $K' = q + (K - q) \bmod p$.

Zhývá si uvědomit, že K' , tedy ani počet kroků výpočtu předperiody, nemůže přesáhnout N^3 už jen díky tomu, že více různých trojic se nemůže objevit. To zároveň slouží jako důkaz, že perioda se v posloupnosti vždy objeví.

Algoritmus, který najde periody tedy poběží v case $O(N^3)$ a spotřebuje $O(N^3)$ paměti. Může se to zdát hodně, ale při rozumně velkém N , které silbovalo zadání, je to velké zrychlení oproti $O(K)$. Takové řešení mohlo být za plný počet bodů.

```
Program (C):
http://ksp.mff.cuni.cz/viz/28-Z1-6.c
```

Stále to jde ale rychleji.

Upozorňujeme čtenáře, že následující text obsahuje zrychlené množství matematiky. Doporučujeme!

Podívejme se, jak se hodnoty proměnných x, y a z změní během prvního kroku (staré hodnoty označíme indexem 1 a nové indexem 2):

$$x_2 = y_1, \quad y_2 = z_1, \quad z_2 = (a \cdot x_1 + b \cdot y_1 + c \cdot z_1) \bmod N$$

Rozezpíšeme si tyto rovnice do tvaru, ve kterém výsledné proměnné jsou vázány součet vstupů:

$$\begin{aligned} x_2 &= (0 \cdot x_1 + 1 \cdot y_1 + 0 \cdot z_1) \bmod N \\ y_2 &= (0 \cdot x_1 + 0 \cdot y_1 + 1 \cdot z_1) \bmod N \\ z_2 &= (a \cdot x_1 + b \cdot y_1 + c \cdot z_1) \bmod N \end{aligned}$$

Tyto lineární rovnice, které x (x_1, y_1, z_1) počítají (x_2, y_2, z_2), se dají taky vyjádřit jako maticové násobení těchto vektorů zleva. Pokud vám matice a vektory nejsou známé, můžete si pro naše účely představit vektory jako skupinky čísel, se kterými budeme pracovat najehtou (třeba (x_1, y_1, z_1)).

Matice jsou jenom zkratka za postup tohoto typu: z jednoho vektoru udělám jiný vektor, a složky nového vektoru jsou nějaké vážené součty složek prvního, kde váhy jsou konstanty. Právě tyto konstanty v pořadí jako ve zvorčích číslo výše tvoří matici. Naš případ se dá maticově zapist jako takovéto násobení:

$$\begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix} = \begin{pmatrix} 0 \cdot x_1 + 1 \cdot y_1 + 0 \cdot z_1 \\ 0 \cdot x_1 + 0 \cdot y_1 + 1 \cdot z_1 \\ a \cdot x_1 + b \cdot y_1 + c \cdot z_1 \end{pmatrix} \bmod N = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ a & b & c \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} \bmod N$$

Tato matice se ještě bude hodit, tak si ji označíme:

$$M = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ a & b & c \end{pmatrix}$$

Když potom z (x_2, y_2, z_2) počítáme (x_3, y_3, z_3), provádíme stejné operace: $x_3 = y_2, y_3 = z_2, z_3 = (a \cdot y_2 + b \cdot y_2 + c \cdot z_2)$, neboli zapíšeme maticově:

$$\begin{pmatrix} x_3 \\ y_3 \\ z_3 \end{pmatrix} = \begin{pmatrix} M \cdot \begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix} \end{pmatrix} \bmod N = \begin{pmatrix} M \cdot \begin{pmatrix} M \cdot \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} \end{pmatrix} \end{pmatrix} \bmod N$$

A podobně to vypadá i další posunutí proměnných x, y, z o jedno místo znamená jejich vynásobení zleva maticí M . Obecně pro hodnoty x, y, z po K krocích platí:

$$\begin{pmatrix} x_K \\ y_K \\ z_K \end{pmatrix} = M^{K-1} \cdot \begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix}$$

Co to ale vlastně znamená násobit matice? Násobení matic je trochu divná operace. Je vyvinutá tak, aby bylo asociativní, tedy aby pro libovolné čtvercové matice A a B a každý vektor x platilo $A \cdot (B \cdot x) = (A \cdot B) \cdot x$. Matice $(A \cdot B)$ tedy reprezentují stejnou operaci, jako nejdřív vektor „prohnan“ maticí B a pak maticí A .

Na vektorech o dvou dimenzích a maticích velkých 2×2 si z toho snadno můžeme odvodit, jak se musí násobit. Mějme pár matic a libovolný vektor:

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}, \quad B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}, \quad x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

Potom:

$$B \cdot x = \begin{pmatrix} b_{11} \cdot x_1 + b_{12} \cdot x_2 \\ b_{21} \cdot x_1 + b_{22} \cdot x_2 \end{pmatrix}$$

Po vynásobení A dostaneme:

$$\begin{aligned} A \cdot (B \cdot x) &= \begin{pmatrix} a_{11}(b_{11}x_1 + b_{12}x_2) + a_{12}(b_{21}x_1 + b_{22}x_2) \\ a_{21}(b_{11}x_1 + b_{12}x_2) + a_{22}(b_{21}x_1 + b_{22}x_2) \end{pmatrix} = \\ &= \begin{pmatrix} x_1(a_{11}b_{11} + a_{12}b_{21}) + x_2(a_{11}b_{12} + a_{12}b_{22}) \\ x_1(a_{21}b_{11} + a_{22}b_{21}) + x_2(a_{21}b_{12} + a_{22}b_{22}) \end{pmatrix} \end{aligned}$$

Odsud si můžeme hned předčíst, jak musí matice $A \cdot B$ obecně vypadat. Pro rozměr 3 je koeficientů víc, ale dáji se odvodit stejným způsobem.

Tedy už víme, co dělá a jak se dělá násobení matic a že je asociativní. Proto $(K-1)$ -násobení vynásobení x_1 zleva maticí M se dá napsat jako $M^{K-1} \cdot x_1$ – nezalží na užávorování.

Naše první řešení se dá popsat jako „vezmi $x_1, (K-1)$ -krát ho zleva vynásob M a pak vrať x_1 “. Tohle znamená i -krát promásobit třísložkový vektor maticí 3×3 (a promodulit výsledek N).

Rychlejší řešení to oběje tím, že *rychle spočítá* M^K (modulo N) a potom vynásobí x zleva M^K (a zase vymodulí). Bude nám na to stačit $O(\log K)$ násobení matic 3×3 a jedno násobení vektoru. Násobení matic matice sice potřebuje více operací než násobení matic vektoru, ale obě operace trvají konstantní čas; jenom násobíme a sčítáme konstantní počet čísel na vstupu.

Díky asociativitě násobení matic se můžeme spoletnout na to, že pro libovolná g a h platí $M^{g+h} = M^g \cdot M^h$. Číslo $(K-1)$ si vyjádříme jako součet mocnin dvojky (neboli ho převedeme do dvojkové soustavy) a přislaňme si přenspořádané výpočet M^{K-1} . Například pro $K = 44$ tak dostaneme $M^{43} = M^{32} \cdot M^8 \cdot M^2 \cdot M^1$. Takhle si rozložíme M^{K-1} na nejvýše $\lceil \log_2 i \rceil$ násobení matic, které jsou „dvojkové mocniny M^i “.

Tyhle „dvojkové mocniny M^i “ budeme efektivně generovat: v prvním kroku budeme držet $M = M^1$. Jejím vynásobením samy se sebou dostaneme M^2 , ze kterého po dalším umocnění dostaneme M^4 , a tak dále.

V matci M^i si budeme postupně stavět potřebnou matici M^{i+1} . Počítání hodnoty M^i bude maticový ekvivalent jedničky. Takové matice se říká jednotkovou, značí se ve třech rozměrech I_3 , a je to ta jediná, která se chová v násobení matic jako jednička v násobení čísel: když je T libovolná matice 3×3 , potom $I_3 \cdot T = T \cdot I_3 = T$. V vlastnosti násobení se dá uhotnout, jak I_3 vypadá. Má nuly všude kromě diagonály, kde má jedničky:

$$I_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Budeme se posouvat v binárním zápisu čísla $(K-1)$ zprava dolava. Na začátku kroku číslo i budeme mít uloženo prozatím maticí M^i a maticí M^{2^i} . Podíváme se na i -té číslo zprava v binárním zápisu, a ještě tam uvidíme jedničku, přensobíme M^i maticí M^{2^i} .

Ať vidíme jedničku nebo nulu, umocníme M^{2^i} na druhou, čímž si spočítáme pro další krok $M^{2^{i+1}}$. Po každém kroku taky vymodulíme obsah $M^{2^{i+1}}$ i M^i číslem N . Tohle modulení musíme provádět, aby chom zajistili, že všechny použité proměnné budou omezené velké, konkrétně mezi 0 a $N-1$. Bez modulení by totiž mohly ríst hodnoty rychle a ukládat velká čísla nesrojnými konstantami mimo paměti.

Z výsledku nás zajímá jenom x_i mod N a to vyjde stejně ať počítáme přesně, nebo modulo N .

Na konci bude M^i součín těch mocnin M^i , které odpovídají valám jedniček v binárním zápisu $(K-1)$, tedy bude M^i přensé rovna M^{K-1} . Za každou číslici v binárním zápisu $(K-1)$ provedeme jedno nebo dvě násobení matic, což trvá konstantní čas, a protože čísle je logaritmičky mnoho, trvá náš algoritmus jenom $O(\log K)$. Dá se navíc naplnenem-tovat tak, že mu stačí jenom konstantní množství paměti.

Program (Python 3):
http://ksp.mff.cuni.cz/viz/28-Z1-6.py

Michal „Převlád“ Pokorný

⁵ Obecně se operaci „spočítat zbytek po dělení“ říká „modulo“ a často se dá v programovacích jazycích provést operátorem $\%$. Program navíc v praxi nejspíš i poběží rychleji: přístupy do paměti jsou rychlejší, když zapisujeme a čteme pořádk do stejného místa. Zatímco první program popisuje K různými místy v paměti, druhý program jenom tři.

⁷ To platí, protože $(a + b) \bmod N = [(a \bmod N) + (b \bmod N)] \bmod N$. x_i vznikne z (x_0, y_0, z_0) jenom jejich posloupností, proto můžeme všechny mezivýsledky včetně matic ukládat modulo N .