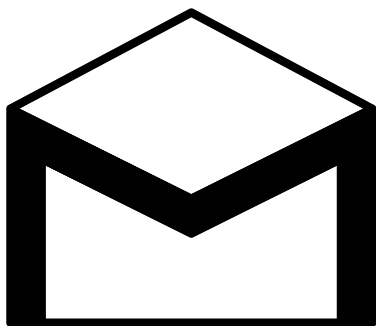
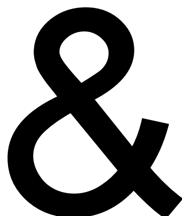
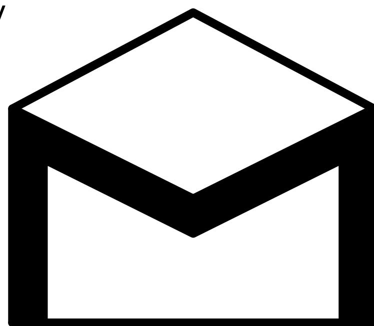
**STUDENTSKÝ ČASOPIS A KORESPONDENČNÍ SEMINÁŘ**

Ročník XXV
Číslo 1

# MATEMATIKA FYZIKA INFORMATIKA

Uvnitř najdete několik témat a s nimi souvisejících úloh. Zamyslete se nad nimi a pošlete nám svá řešení. My vám je opravíme, pošleme zpět s dalším číslem a ta nejzajímavější z nich otiskneme. Nejlepší řešitele zveme na podzim a na jaře na soustředění.

## Milí řešitelé,

v rukou držíte časopis M&M, určený pro zvídavé středoškoláky. Stejně jako každý rok se můžete prostřednictvím našich teoreticky i experimentálně zaměřených tématek dozvědět a naučit mnoho zajímavých věcí z různých oblastí matematiky, fyziky a informatiky.

S prvním číslem nového ročníku přichází i několik změn. Letos v M&M objevíte více tématek, než bylo dosud obvyklé. O klasické úlohy ovšem nepřijdete! V každém čísle budou u právě aktuálních tématek nové úlohy s nimi provázané. Úloh se však nemusíte bezpodmínečně držet. V tématkách je povoleno experimentovat, tvořit vlastní teorie a realizovat své nápady, proto se toho nebojte. Svoje úvahy sepište a pošlete nám je tak, jak jste byli zvyklí (pro detailnější informace viz sekce „Jak řešit“ na protější straně a na našem webu). Jakákoliv řešení týkající se daných tématek jsou vítána a budou adekvátně bodově hodnocena.

Nemusíte celý rok setrvat jen u těch tématek, která jste si vybrali na začátku. Další tématka se objeví v následujících číslech a budou trvat několik měsíců nebo i celý ročník, takže není problém přecházet od jednoho ke druhému. Doporučujeme jich řešit i více najednou, začít nebo přestat řešit další či jiné tématko můžete kdykoliv se vám zachce.

V průběhu ročníku budeme zveřejňovat vaše nejlepší řešení spolu s těmi organizátorskými. Takto můžete sledovat průběh tématka v případě, že jej nebudete chtít řešit již od začátku.

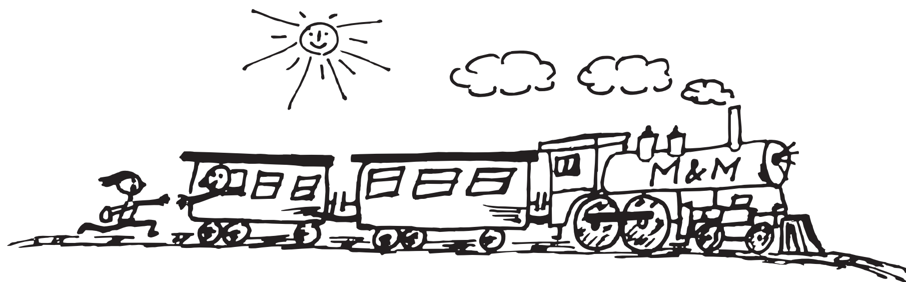
Na řešení tématek nemusíte být sami. O každém tématku můžete s dalšími zájemci komunikovat pomocí e-mailové konference, jejíž adresa je u tématka uvedena.

V prvním čísle se vám představí tématka, ve kterých budete objeviteli neznámého měsíce, kryptografy, návrháři algoritmů nebo si můžete vyzkoušet, co je pravdy na tom, že teplá voda mrzne rychleji než studená.

Pokud máte jakékoliv dotazy, nezdráhejte se nám napsat (kontakt najdete na zadní straně).

Přejeme vám při řešení hodně zábavy a bystrou mysl.

*Vaši organizátoři*



# Jak řešit

V každém z čísel v tomto školním roce zveřejníme několik textů, které nazýváme témata či tématka. Tyto texty nejen z oblasti matematiky, fyziky a informatiky popisují nějaký problém a jsou doprovázeny návodnými úlohami. Vaším úkolem je zamyslet se nad daným problémem a sepsat vaše úvahy ve formě krátkého textu. Zde je pár tipů, jak na to:

- Pročtěte si zadání všech tématek a vyberte si ta, která jsou vám nejbližší. Doporučujeme jich řešit víc najednou.
- Nespěchejte s řešením, málokdy se stane, že vás hned něco napadne.
- Buďte kreativní! Objevíte-li problém týkající se tématka, kterému se zrovna nevěnuje žádná z úloh, neznamená to, že se jím při řešení nesmíte zabývat. Naopak, vaše vlastní nápady vítáme a oceňujeme. Budeme rádi, pokud se při řešení nebudete držet jen námi položených otázek, ale budete si klást i nějaké vlastní.
- Během roku můžete samozřejmě řešit různá tématka a naopak z každého můžete poslat jen některé úlohy. Tématka budou mít různou délku a jejich počet se budeme snažit držet v průběhu ročníku stejný v každém čísle.
- Bodují se i úprava a struktura příspěvku, proto pište srozumitelně, vyjadřujte se jednoznačně a pokud možno se vyvarujte hrubých gramatických chyb. Pokud obdržíme nerozluštitelný text, projeví se to na bodování. Na druhou stranu nemusíte sepisovat vědeckou práci o desítkách stran. Pamatujte, že někdy je jedna strana lepší než pět.
- Neostýchejte se sáhnout po odborné literatuře či se poradit s on-line zdroji. Body vám za to rozhodně neubudou, jen nezapomeňte ve vašem řešení zmínit zdroj informací.
- Pokud si nevíte rady, zkuste dát hlavy dohromady a zamyslet se nad nějakým tématem či úlohou s dalšími řešiteli. U každého tématka je adresa e-mailové konference, pomocí které můžete komunikovat s řešiteli, kteří se o tématko zajímají. Pro přidání do konference napište organizátorovi zodpovědnému za dané téma. Pokud sepíšete řešení společně, počet bodů, které každý z vás dostane, vypočteme podle vzorce $\frac{3b}{n+2}$, kde $b$ je počet bodů, které by příspěvek získal, pokud by měl jediného autora, a $n$ je počet autorů příspěvku.
- Jak již bylo řečeno, za řešení dostáváte body. Po obdržení určitého počtu bodů a za dobré umístění na konci ročníku od nás obdržíte věcné ceny, avšak největší odměnou, kterou za svou snahu získáte, je účast na soustředění. Ty nejlepší z vás totiž zveme na týdenní soustředění, které se koná dvakrát

do roka. Avšak pokud se vám nepovede dostat na soustředění, nesmutněte, protože s ostatními řešiteli se můžete potkat i na víkendovém setkání.

Další výhoda, která vám z řešení M&M vyplyne, jsou odpuštěné přijímačky na Matfyz, a to pokud získáte za ročník minimálně 65 bodů.

Řešení posílejte na e-maily organizátorů uvedené pod daným tématkem. Pokud sepíšete řešení papírově, můžete ho čitelně naskenovat, nebo poslat poštou (adresu najdete na konci čísla). Pokud posíláte řešení poprvé, napište nám zároveň i své jméno, adresu, e-mail, školu a rok maturity.

# Zadání témat

**Termín odeslání úloh 1. série: 23. 10. 2018**
**(25. 9. 2018 pro účast na podzimním soustředění)**

## Téma 1 – Paradoxní výsledky

V tomto spíše experimentálním tématku se budeme zabývat ověřováním a následně interpretací dat, která jste naměřili. Nebude nijak početně složité, chceme po vás, abyste se naučili přemýšlet o datech, která získáte, vyvozovat z nich hypotézy a umět si na jejich základě pokládat otázky, které vás dovedou k dalším úvahám a experimentům. Začněme takovým malým příběhem, který nám nastíní, o co se bude v tomto tématku jednat.

Bylo nebylo, kdesi daleko žil jeden malý kluk. Jmenoval se Bartoloměj a měl rád zmrzlinu. Vyráběl si ji spolu s kamarády ve škole skoro každý den. Vždy smíchali mléko a cukr, tuto směs zahřáli v hrnci, nechali vychladnout a pak dali zmrznout do mrazáku. Jelikož kluků bylo hodně a míst v mrazáku málo, ne vždy se dostalo na všechny. Platilo, že kdo dřív přijde, ten dřív mele, takže Bartoloměj jednoho dne v zápalu boje o poslední místo v mrazáku zariskoval, nepočkal, až mu směs na zmrzlinu vychladne, a zavřel ji do mrazáku horkou. Když se po chvíli

vrátil, bylo mu opravdu divné, že jeho směs už ztuhla a proměnila se ve zmrzlinu, avšak zmrzliny ostatních kamarádů ještě neztuhly. Vždyť byly studenější než ta jeho! V Bartolomějovi to vzbudilo zájem, ale ať přemýšlel, jak chtěl, nemohl přijít na to, proč tomu tak bylo. Když se zeptal učitelů, tak mu nevěřili a dělali si z něj legraci, že si vymyslel své vlastní fyzikální zákony. Jenže Bartoloměj věděl, co viděl. Rozhodl se dokázat ve školní laboratoři, že měl pravdu. Připravil si dvě navlas stejné nádoby a poté do jedné nalil horkou vodu a do druhé o něco chladnější. Obě poté zavřel do školního mrazáku.

**Problém**: *Co si myslíte vy, jaký byl výsledek jeho pokusu? Proveďte podobný pokus a ověřte, zda měl Bartoloměj pravdu. Doporučujeme to vyzkoušet nejprve s vodou, ta má víc konzistentní strukturu než Bartolomějova zmrzlina (ale nikdo vám samozřejmě nebrání něco podobného vyzkoušet). Zkuste si graficky zaznamenat závislost doby tuhnutí na počáteční teplotě jednotlivých měření, zkoumejte i závislost teploty na čase pro jednotlivá měření. Jaké parametry určují, zda je voda zmrzlá? Stanovte si okamžik, kdy prohlásíte jednotlivá měření za ukončená.*

Tip: Experimenty bývají obecně časově náročnější, proto pokud se rozhodnete pro toto tématko, nezačínejte s měřeními na poslední chvíli.

*Pája a Matej;* pavla.trembulakova@seznam.cz
*e-mailová konference:* paradoxni@mam.mff.cuni.cz

# Téma 2 – Principy kryptografie

## Substitučně-permutační sítě a AES

V on-line světě každý den odešleme i přijmeme celou řadu šifrovaných zpráv. Obvykle se tak děje bez toho, abychom si to jakkoli uvědomovali. Zatímco ještě před několika lety byla i nešifrovaná komunikace na internetu celkem běžná, dnes, pokud si chceme zobrazit webovou stránku nebo někomu zavolat, obvykle šifrování využíváme. Za způsoby, jakými jsou data šifrována, přitom často nejsou schovány nijak složité úvahy. Cílem tohoto tématu bude tyto postupy přiblížit a trochu si je osahat.

V první části se budeme věnovat substitučně-permutačním sítím (substitution-permutation network, SPN), na kterých je založená mimo jiné v dnešní době nejrozšířenější bloková šifra AES.

### Historické metody

Potřeba utajovat zasílané vzkazy je stará tisíce let. Například z dob antiky pochází slavná Caesarova šifra, která spočívá v posunutí každého písmena v abecedě o tři, tedy v nahrazení (neboli substituci) jednoho písmena druhým podle pevně stanovené tabulky.

Později byla vymyšlena Vigenèrova šifra, u které vznikne zašifrovaný text postupným přičtením jednotlivých písmen klíče k jednotlivým písmenům původního textu. Pokud se tedy v původním textu (budeme mu říkat *otevřený*) objevuje

stejné písmeno vícekrát, může být v zašifrovaném textu zakódované pokaždé pomocí jiného znaku.

Dalším tradičním způsobem je proházení (permutace) pozic písmen obsažených v textu.

Substitučně-permutační sítě v podstatě nepřinášejí nic nového. Pouze všechny tyto tři dávno známé metody šikovným způsobem propojují.

### Moderní blokové šifry

Aby matematici mohli šifry lépe popisovat a zkoumat, zavedli pojem „bloková šifra“. Tu si můžeme představit jako sadu dvou černých krabiček. První slouží pro šifrování – pokud do ní vložíme otevřený text a šifrovací klíč, dostaneme zašifrovaný text. Druhá krabička pak umožňuje s pomocí zašifrovaného textu a šifrovacího klíče vytvořit zpátky otevřený text.

Šifrovacím klíčem je vždy nějaká posloupnost znaků („heslo“) předepsané délky. Bloková šifra vždy očekává vstup pevně zadané délky a stejně dlouhý je i její výstup. Této délce se říká *velikost bloku.* Například AES používá bloky o velikosti 128 bitů neboli 16 znaků. Pokud chceme zašifrovat delší text, můžeme ho například rozdělit na části odpovídající velikosti bloku a ty zašifrovat zvlášť.

### Dvojková soustava

Než budeme pokračovat, musíme udělat krátkou odbočku. Moderní kryptografie předpokládá využití počítačů, a tak jim přizpůsobuje využívané prostředky. Čísla jsou obvykle zapisována ve dvojkové soustavě. Při početních operacích se typicky počítá s bloky dat (proměnnými) omezené velikosti.

Každé číslo je možné v desítkové soustavě zapsat jako součet jednociferných násobků mocnin deseti, například $2945 = 2 \cdot 10^3 + 9 \cdot 10^2 + 4 \cdot 10^1 + 5 \cdot 10^0$. Podobně ve dvojkové soustavě je možné každé číslo zapsat jako součet mocnin dvojky.

Pokud chceme převést číslo z desítkové soustavy do dvojkové, stačí najít vhodné mocniny dvou, jejichž součet odpovídá původnímu číslu. Například:

$$(42)_{10} = 4 \cdot 10^1 + 2 \cdot 10^0 = 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = (101010)_2$$

Převod z dvojkové soustavy do desítkové probíhá podobně.

Protože čísla ve dvojkové soustavě jsou dlouhá a jejich zápis pro člověka není příliš dobře čitelný, často se místo dvojkové soustavy používá soustava šestnáctková. Způsob převodu je opět úplně stejný. Abychom měli pro šestnáctkovou soustavu dostatečné množství číslic, přidávají se za číslici 9 ještě písmena A až F, například $(42)_{10} = (2\mathrm{A})_{16}$. Všimněte si, že 16 je mocninou 2, převod mezi dvojkovou a šestnáctkovou soustavou je proto velmi jednoduchý. Každá hexadecimální číslice totiž odpovídá čtveřici binárních číslic, např. $(2\mathrm{A})_{16} = (10\ 1010)_2$.

Pro čísla zapsaná ve dvojkové soustavě můžeme snadno definovat takzvané bitové operace, tedy početní operace, které provádíme na jednotlivých bitech (tj. pozicích ve dvojkové soustavě) čísla nezávisle na ostatních. Obvyklými operacemi pro dvě čísla jsou:

and & z čísel vybere pouze ty bity, na kterých je v obou číslech 1 (ostatní bity nastaví na 0), tedy např.

$$\begin{array}{rr} & 10100110 \\ \& & 01101011 \\ \hline & 00100010 \end{array}$$

or $\|$ vybírá ze zadaných čísel všechny bity, na kterých je jednička alespoň v jednom z nich, např.

$$\begin{array}{rr} & 10100110 \\ \| & 01101011 \\ \hline & 11101111 \end{array}$$

xor $\oplus$ na výstup vypíše 1 u těch bitů, kde se zadaná čísla liší, např.

$$\begin{array}{rr} & 10100110 \\ \oplus & 01101011 \\ \hline & 11001101 \end{array}$$

Kromě těchto operací se ještě využívá klasické sčítání (značíme +) na číslech omezené délky, ve kterém zanedbáváme přenos přes nejvyšší řád. Pokud tedy budeme pracovat s 8-bitovými čísly, tak $10100110 + 01101011 = 00010001$. Hodnota odpovídá běžnému sčítání v desítkové soustavě, při kterém nás ale zajímá ze součtu pouze zbytek po dělení $2^8$ (obecně $2^n$ pro $n$-bitová čísla).

Navíc ještě definujeme operace, které provádíme vždy pouze s jedním číslem:

not $\sim$ je operace, která prohazuje, na kterých bitech jsou jedničky a na kterých nuly, např. $\sim 10100110 = 01011001$.

shift left $\ll$ posouvá všechny bity o jednu pozici vlevo a na konec přidá nulu. Bit nejvíce vlevo se ztratí. Např. $\ll 10100110 = 01001100$. Tato operace odpovídá násobení 2.

shift right $\gg$ posouvá všechny bity o jednu pozici vpravo a na první pozici přidá nulu. To odpovídá dělení 2, např. $\gg 10100110 = 01010011$.

rot left $\lll$ neboli rotace vlevo funguje obdobně jako shift vlevo, ale bit z nejlevější pozice přesune na konec čísla, např. $\lll 10100110 = 01001101$.

rot right $\ggg$ neboli rotace vpravo opět připomíná shift vpravo, ale poslední bit přesune na začátek čísla, např. $\ggg 10100110 = 01010011$, $\ggg 01101011 = 10110101$.

V tomto dílu využijeme jen některé z těchto operací, ale v průběhu tématu se nám mohou hodit i další.

**Úloha 1** [1b]: *Mějme libovolná čísla a a b. Dokážete získat výsledky operací or, xor a rotace vpravo pouze pomocí opakovaného využití operací and a not?*

## Šifra OCTA

Vybaveni potřebným matematickým aparátem si můžeme konečně zadefinovat naši vlastní jednoduchou šifru OCTA založenou na substitučně-permutační síti a trochu ji prozkoumat.

OCTA využívá blok o velikosti osmi bitů a očekává klíč, jehož délka v bitech je násobkem osmi. Jak je u moderních blokových šifer obvyklé, bude dělat několikrát po sobě operace zapsané pomocí stejného předpisu. Při tvorbě šifrového textu tedy proběhne několik kol šifrování (těm říkáme *rundy*).

Na začátku každé rundy přixorujeme k aktuálnímu textu klíč. Abychom pořád nexorovali stejnou hodnotu, použijeme poprvé prvních osm bitů klíče (označíme $K_1$), v druhé rundě dalších osm bitů ($K_2$) a tak dále. Jakmile dojdeme až na konec klíče, začneme opět od začátku. Pokud by tedy měl klíč například 16 bitů, tak $K_1 = K_3 = K_5 = \dots$.

Druhou operací v rámci rundy bude využití takzvaných S-boxů neboli substituce podle pevně stanovené tabulky. Text (tj. osm bitů) rozdělíme na čtyři skupiny po dvou bitech a hodnotu každé z nich změníme podle tabulky:

| $x$ | 00 | 01 | 10 | 11 |
|---|---|---|---|---|
| $S(x)$ | 10 | 00 | 01 | 11 |

Tedy například text 01110101 změníme na 00110000.

Třetí operací pak bude třikrát po sobě rotace celého textu vpravo (tedy rotace vpravo o tři bity).

Takových rund proběhne pět. Na závěr ještě jednou přixorujeme klíč.

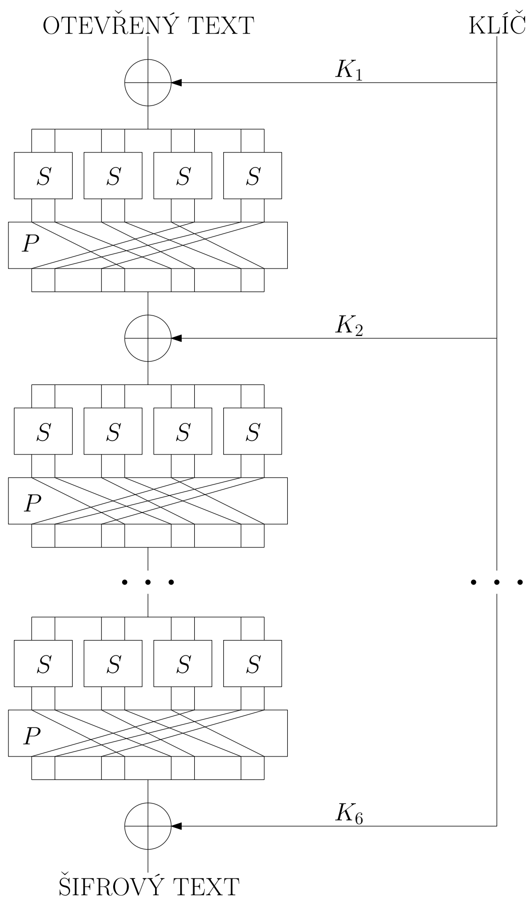
Schéma celé šifry je zachyceno na Obrázku 1.

Abychom pomocí naší šifry mohli šifrovat reálné texty, využijeme v souladu se skutečnou praxí reprezentaci znaků pomocí ASCII tabulky[1], která převádí základní tisknutelné znaky na osmibitové hodnoty. Pokud budeme chtít mít v otevřeném textu písmeno A, tak do něj tedy napíšeme $(41)_{16} = (01000001)_2$. Každou osmici bitů (tedy každé písmeno) pak budeme šifrovat zvlášť.

Zbývá otázka, jak bude vypadat funkce pro dešifrování zpráv. Můžeme si ale všimnout, že všechny kroky, které jsme v průběhu šifrování provedli, lze provést i zpětně (všimněte si, že přixorování je samo sobě inverzní operací). Dešifrovat zprávy tedy můžeme pomocí stejného algoritmu, ale pozpátku.

**Úloha 2** [2b]: *Dostali jste zprávu 89 19 2C 8C F9 BC FA CC 6D 6C FC BD 19 59 19 C8 29, o které víte, že je zašifrovaná šifrou OCTA pomocí klíče 4D 41 4E 44 4D (tj. MANDM). Jaká byla původní zpráva?*

[1] viz `https://www.asciitable.com/`




**Obrázek 1:** schéma šifry OCTA

## Využití šifry

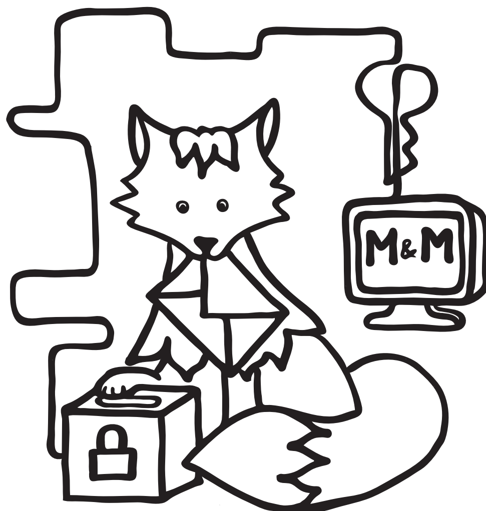
**Problém** [2b+]: *Představme si reálné využití šifry. Chcete mít aktuální přehled o tom, kolik máte doma tabulek hořké čokolády. A tak si každou hodinu posíláte do školy zprávu o jejich počtu. Protože tato informace je ale důvěrná, posíláte ji zašifrovanou pomocí šifry OCTA s klíčem, který znáte jenom vy.*

*Je k tomuto účelu OCTA vhodná? Pokud někdo bude odposlouchávat vaše zprávy, dokáže z nich něco zjistit? Jak by bylo možné zasílání zprávy vylepšit?*

Zkuste se zamyslet i nad případnými dalšími nedostatky prezentované šifry. Kde všude by mohla v praxi narazit? Za pěkný rozbor lze získat pěkné bodové ohodnocení.

**Úloha 3** [2b+]: *Můžete též zkusit zauvažovat nad volbou šifrovacích klíčů. Existuje klíč, který vámi zvolený otevřený text zašifruje tak, že otevřený i šifrový text budou stejné? A existuje klíč, který zašifruje libovolný otevřený text na šifrový text, který bude s otevřeným textem shodný?*

*Najdete otevřený text a klíč, který tento text zašifruje na samé 1, tj. FF?*




## AES

Zatím jsme si hráli s jednoduchou šifrou OCTA. Jak to ale vypadá s reálnými šiframi jako třeba s AES? Od naší šifry se až tak moc neliší. AES používá 128-bitové bloky a klíče o velikosti 128, 192, nebo 256 bitů. Podle velikosti klíčů je využito 10, 12, nebo 14 rund.

Na začátku dojde k přixorování rundovního klíče. Samotná runda pak obsahuje využití S-boxů v kroku nazvaném *SubBytes* a potom dvě permutace *ShiftRows* a *MixColumns*. Runda je zakončena opět přixorováním rundovního klíče.

Detailní popis šifry můžete nalézt třeba na Wikipedii[2].

**Úloha 4** [2b]: *V detailním popisu AES si můžete všimnout, že v poslední rundě je vynechaná permutace MixColumns. Proč tomu tak je? A proč není vynechaná i ShiftRows?*

*Pokud bychom i v naší šifře OCTA vynechali v poslední rundě permutaci, změnili bychom tím obtížnost jejího prolomení?*

[2] `https://en.wikipedia.org/wiki/Advanced_Encryption_Standard`

### Poznámka na závěr

Protože se toto téma zabývá v praxi využívanými postupy, na část otázek a úloh může být jednoduché najít odpovědi na internetu. Takový postup je zcela regulérní a nebojte se jej využívat. Kdykoli budete něco tvrdit, tak ale nezapomínejte svá tvrzení zdůvodňovat. Argument „je to pravda, protože to psali na Wikipedii“ uznávat nebudeme. Ale pokud si na Wikipedii přečtete zdůvodnění a to potom využijete ve svém řešení včetně citace zdroje, je to naprosto v pořádku.

*Kuba, Káťa a Lenka;* jakub.topfer@matfyz.cz
*e-mailová konference:* krypto@mam.mff.cuni.cz

## Téma 3 – Neznámý měsíc

Byli jste vysláni do jedné z hvězdných soustav objevených výzkumníky institutu M&M, specificky prozkoumat soustavu M&M-25 (podle jednoho ze standardních značení „objevitel-pořadové číslo objevené hvězdy-planeta měsíc“). Nejzajímavější se vám zdál měsíc M&M-25-d I. Tento měsíc má velikost i hustotu atmosféry přibližně stejnou jako Země a nemá vázanou rotaci. Planeta, kterou obíhá, je velikostí podobná Uranu, má ale výrazněji viditelné bouře – podobné těm, jaké jsou na Jupiteru. Při vstupu do atmosféry měsíce se vaše loď začala rozpadat, tak jste ji opustili v únikovém modulu a následně úspěšně přistáli. Máte prostředky pro přežití, ale přišli jste o veškeré pokročilé vybavení. Než se jej vydáte hledat, bylo by dobré zjistit o planetě i měsíci vše, co půjde zjistit pomocí improvizovaného vybavení vyrobeného z vám dostupných předmětů – např. máte jídlo, igelitové balíčky, nějaké ulomitelné části lodi… Bude také dobré zavést nějaký navigační systém – jak definovat souřadnice, podle čeho určovat pozici. Pokud to nevymyslíte jinak, zamyslete se alespoň jak to udělat, kdybyste měli stopky a metr.

**Úloha 1**: *Zaveďte navigační systém a navrhněte nástroje a metody pro navigaci.*

**Úloha 2**: *Zjistěte gravitační zrychlení, velikost, parametry rotace a oběžné dráhy měsíce a planety.*

**Problém**: *Můžete určit ještě něco jiného pomocí obecně dostupných předmětů – např. kdybyste měli lékárničku či nějakou kosmetiku?*

*Kuba a Kubo;* kusnir.jk@gmail.com
*e-mailová konference:* mesic@mam.mff.cuni.cz

# Téma 4 – Algoritmy od nuly (do $n$)

## Stavíme počítač

### Slovo úvodem

Jak je již z názvu patrné, v našem tématku se budeme zabývat algoritmy. Co je to takový algoritmus? Žádná oficiální definice neexistuje. Dá se třeba říct, že je to přesný popis postupu, jak něco udělat. V tomto pojetí by algoritmem mohl být třeba dobře popsaný recept na smažení palačinek nebo návod na složení židle z IKEA. My se ale omezíme pouze na algoritmy, které jsou ve výsledku určeny pro zpracování počítačem. Po počítači nemůžeme chtít, aby nám usmažil palačinky (tedy, chtít to samozřejmě můžeme). Co po něm naopak chtít budeme, je například sečíst dvě čísla nebo určit, jestli se v posloupnosti čísel vyskytuje dané číslo. Náš algoritmus bude mít vždy vstup (data, se kterými má počítat) a výstup (výsledek). V případě prvního algoritmu by tedy vstupem byla dvě čísla a výstupem jejich součet, vstupem pro druhý algoritmus by byla posloupnost $n$ čísel a hledané číslo $k$ a výstupem informace, zda se číslo $k$ v posloupnosti vyskytuje, případně na které pozici.

Během řešení tématka se budeme snažit pro různé problémy vymyslet algoritmy, které vždy nejen vypíší správný výsledek, ale udělají to v co nejkratším čase. Algoritmus, který odpovídá vždy správně, ale výpočet by mu na dnešních počítačích již pro krátké vstupy trval několik miliard let, pro nás, jak jistě uznáte, není příliš užitečný (a brzy uvidíte, že omylem vymyslet takový algoritmus není vůbec složité).

Při vymýšlení algoritmů budeme, jak název napovídá, začínat úplně od nuly. **Body tedy dáváme za to, že jste vymysleli řešení, a ne za sepsání algoritmu, který jste již dobře znali.** Také věříme, že M&Mko řešíte hlavně proto, abyste se něco nového naučili (a ne jen kvůli bodům). Pokud tedy řešení některých zde zmíněných problémů znáte, pusťte se raději do jiného tématka nebo jiné úložky tohoto tématka, které pro vás budou větší výzvou, nebo si vhodnou úložku k řešení sami vymyslete.

### Počítač

Algoritmy nebudeme spouštět na skutečném počítači – museli bychom brát ohled na spoustu technických detailů, které by nám zabraňovaly soustředit se na samotný algoritmus. Místo toho si tedy vymyslíme vlastní teoretický model počítače. Budeme od něj chtít, aby byl co nejjednodušší a aby zároveň jeho fungování co nejlépe odpovídalo skutečnému počítači (tedy aby algoritmy, které jsou na něm rychlé, byly rychlé i na skutečném počítači a naopak).

Protože zavést takový model od nuly dá trochu práci, ukážeme vám nejdříve jeho verzi, která bude obsahovat o trochu pokročilejší konstrukce, zato bude jednodušší na pochopení a používání. Nemusíte se ale bát, že bychom vás o skutečně minimalistický model ochudili – naleznete ho na konci tohoto textu.

Náš počítač má paměť, do které si můžeme ukládat proměnné. *Proměnná* má vždy nějaké jméno a je v ní uloženo jedno celé číslo. Počítač umí počítat (tedy provádět operace $+$, $-$, $\cdot$, $/$) s čísly uloženými v proměnných a přiřazovat je do jiných proměnných. Protože umíme do proměnných ukládat jen celá čísla, funguje dělení jako dělení se zbytkem, ale zbytek se nikam neukládá (tedy $8/3 = 2$). Na první pohled by se mohlo zdát, že je velmi omezující, že umíme pracovat pouze s celými čísly. Můžete si ale zkusit rozmyslet, že třeba text nebo racionální čísla se dají jednoduše zakódovat pomocí celých čísel.

Když chceme na našem modelu něco spočítat, rozepíšeme náš algoritmus jako jednotlivé instrukce, které má počítač provést, dáme mu je a on je postupně jednu po druhé provádí. Takovému seznamu instrukcí říkáme *kód* nebo *program*. Může vypadat například takto:

1. `a` ← `4` (do proměnné `a` přiřaď číslo 4)
2. `b` ← `5` (do proměnné `b` přiřaď 5)
3. `soucet` ← `a` + `b` (do proměnné `soucet` přiřaď součet `a` + `b`, tedy číslo 9)

Náš program sčítá čísla. Sčítá ale vždy jen čísla 4 a 5, což není moc užitečné. Potřebovali bychom, aby náš algoritmus (sečtení dvou čísel) šlo spustit pro libovolný vstup (tedy pro libovolná 2 čísla) a nemuseli jsme kvůli tomu psát pokaždé nový program. V praxi bychom museli vstup nějak načíst (třeba z klávesnice či souboru). My budeme předpokládat, že máme instrukci na načtení čísla ze vstupu a vypsání čísla nebo textu na výstup. Program tedy bude vypadat následovně:

1. `a` ← přečti číslo
2. `b` ← přečti číslo
3. `soucet` ← `a` + `b`
4. Vypiš `soucet`

Vezměme si nyní další problém z úvodu tohoto textu. Chceme najít určené číslo v posloupnosti délky $n$. Abychom mohli problém vyřešit, museli bychom jako vstup dostat kromě čísla $n$ a čísla, které hledáme, ještě dalších $n$ proměnných – prvky posloupnosti. Stejně tak na výstup bychom někdy mohli potřebovat vypsat třeba $n$ proměnných. Pojmenovávání $n$ proměnných by bylo poněkud nepraktické, proto si zavedeme konstrukt, kterému budeme říkat *pole*. Pole je prostě pojmenovaná posloupnost celých čísel, která má předem určenou délku. V informatice je zvykem číslovat prvky pole od nuly a my budeme tento zvyk dodržovat. K prvkům pole budeme přistupovat tak, že za název pole napíšeme pozici příslušného prvku, tedy např. `seznam[8]` je devátý prvek v poli `seznam` (protože číslujeme od nuly). Jednotlivé prvky pole se pak chovají jako proměnné, a můžeme s nimi tedy dělat to, co s proměnnými – sčítat, odčítat, násobit, dělit (celočíselně), přiřazovat do nich, ale nejde třeba naráz přičíst jedničku ke všem číslům v poli. Pole navíc umí oproti proměnným to, že ho můžeme *indexovat* proměnnou. Tedy pokud je třeba v proměnné `a` hodnota 8, pak `seznam[a]` je to samé, co `seznam[8]`. Také

nesmíme zapomenout, že u pole musí být určena maximální délka, nemůžeme tedy pole rovnou používat (jak to děláme u proměnných), ale musíme počítači napřed říct, aby vyrobil pole dané délky, tedy např. „vyrob pole `seznam` délky $n$“ a až poté „`seznam[i]` $\leftarrow$ 13“.

Chybí nám ještě poslední dvě instrukce – podmínka a cyklus.

*Podmínka* obsahuje nějaký výraz, u kterého lze určit, zda je, či není pravdivý. Výraz má tvar „proměnná nebo číslo $<$, $>$, $\geq$, $\leq$, $=$ jiná proměnná nebo číslo“ (např. `a` $>$ `0`). Můžeme také použít negaci a spojit více výrazů pomocí `OR`, nebo `AND`[3]. Pod podmínku pak napíšeme část kódu, která se má provést, pokud podmínka platí, a volitelně i část kódu, která se má provést, pokud podmínka neplatí. Můžeme tedy napsat:

```
1. a ← vstup
2. Pokud (a/2)·2 = a:
3.     Vypiš „Vstup je sudý“
4. Jinak:
5.     Vypiš „Vstup je lichý“
```

Všimněme si, že u lichých čísel se podíl při dělení dvojkou zaokrouhlí dolů, a tedy opravdu nenastane v podmínce rovnost.

Nyní si zavedeme další konstrukt. Budeme mu říkat *cyklus* a vypadá následovně: Dokud platí „podmínka“, tak dělej „část kódu, která se má provádět“. Cyklus můžeme použít například k napsání programu, který násobí dvě přirozená čísla, aniž by použil operaci násobení.

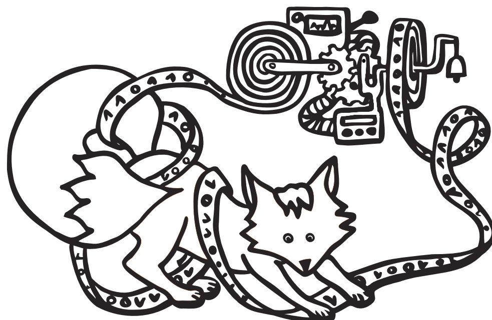
```
1. a ← vstup
2. b ← vstup
3. pocetScitani ← 0
4. soucin ← 0
5. Dokud (pocetScitani < b) tak:
6.     soucin ← soucin + a
7.     pocetScitani ← pocetScitani + 1
8. Vypiš soucin
```

To je zatím k našemu modelu vše. Pojďme si teď zkusit nějaký užitečný algoritmus napsat.

[3]AND a OR jsou logické spojky, které jste pravděpodobně brali ve škole v rámci výrokové logiky pod názvy konjunkce a disjunkce. Složený výrok se spojkou AND je pravdivý právě tehdy, když jsou pravdivé oba výroky, a se spojkou OR právě tehdy, když je pravdivý alespoň jeden z nich.

### Algoritmus – hledání čísla v poli

Nyní si ukážeme jednoduchý algoritmus, který zjistí, zda se v posloupnosti čísel nachází číslo 42. (Zatím nás nebude zajímat, na které pozici je. Můžete si rozmyslet, jak algoritmus upravit, abychom pozici zjistili.) Budeme předpokládat, že víme, kolik čísel dostaneme. Pomocí cyklu si pak ze vstupu uložíme čísla do pole. (Tento technický detail nebudeme pokaždé zmiňovat – dále už budeme vždy předpokládat, že vstup máme na začátku uložen v paměti.)



Pomocí cyklu budeme postupně procházet celé pole. Vytvoříme si proměnné `pozice` a `nalezeno`. V proměnné `pozice` si budeme pamatovat pozici v poli, na které zrovna hledáme číslo 42 a kterou po každém průchodu cyklem zvýšíme o jedna. Takovou proměnnou potřebujeme, abychom věděli, jak v podmínce cyklu otestovat, zda máme pokračovat. V proměnné `nalezeno` bude na začátku 0 symbolizující, že jsme 42 ještě nenalezli, a když ji najdeme, nastavíme tuto proměnnou na hodnotu 1. V každém průchodu cyklem pak podmínkou zkontrolujeme, zda na pozici dané proměnnou `pozice` není číslo 42. Pokud je, tak proměnnou `nalezeno` nastavíme na 1, jinak jí necháme původní hodnotu. Po doběhnutí programu tedy můžeme díky proměnné `nalezeno` zjistit, zda jsme číslo 42 našli, nebo nenašli.

Celý algoritmus by tedy vypadal asi následovně:

Vstup: posloupnost čísel v poli `posloupnost` a jejich počet v proměnné `n`

```
1. pozice ← 0
2. nalezeno ← 0
3. Dokud pozice < n:
4.    Pokud posloupnost[pozice] = 42:
5.       nalezeno ← 1
6.    pozice ← pozice + 1
```

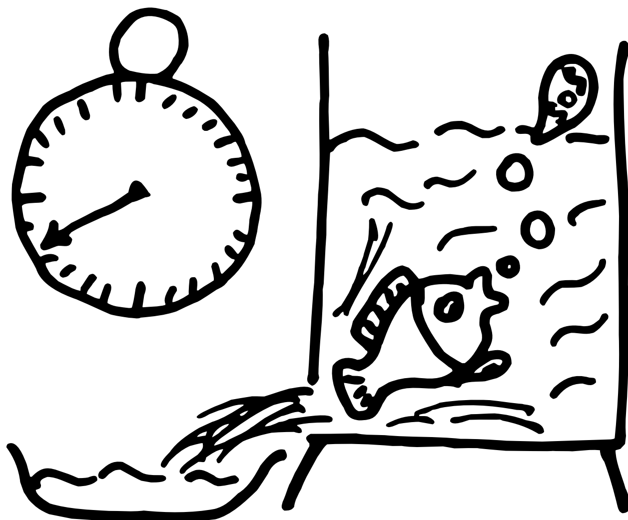
Výstup: Proměnná `nalezeno`, ve které je uložena 1, právě pokud je číslo 42 v posloupnosti obsaženo

Můžeme si navíc všimnout, že jsme k ničemu nepoužili to, že jsme hledané číslo znali předem, a že jsme na řádce 4 mohli kontrolovat rovnost s proměnnou, ve které by bylo uložené hledané číslo. Umíme takto tedy zjistit, zda je v poli libovolné číslo uložené v proměnné.

### Časová složitost

V úvodu jsme tvrdili, že se budeme snažit vymýšlet algoritmy, které spočtou výsledek v co nejkratším čase. Co to ale na našem novém počítači vlastně znamená? Protože na teoretickém modelu jen těžko můžeme měřit čas běhu programu, budeme místo toho počítat provedené instrukce. Vztah ke skutečným počítačům je zde zřejmý – skutečné procesory opravdu mají sadu instrukcí (do kterých je každý program přeložen) a vykonat jednu instrukci trvá nějakou krátkou dobu. Počet vykonaných instrukcí tedy nějakým způsobem odpovídá době běhu programu.

Za instrukci považujeme některou z operací $\leftarrow$ (přiřazení), $+$, $-$, $\cdot$, $/$ nebo určení pravdivostní hodnoty jednoduchého výrazu.



Můžete se pokusit spočítat, kolik instrukcí provede výše popsaný algoritmus (nezapomeňte, že některé instrukce uvnitř cyklu bude počítač provádět $n$-krát). Už jen to, že se autoři tohoto textu nejsou schopni shodnout, zda to vyjde $4n+4$, nebo $6n+3$, napovídá, že spočítat přesný počet instrukcí není vůbec jednoduché a u složitějších algoritmů téměř nemožné. Navíc když spustíme program na skutečném počítači, závisí počet instrukcí na tom, jak přesně funguje procesor daného stroje. Počítače jsou také ve skutečnosti různě rychlé, takže stejná instrukce bude trvat na různých počítačích různě dlouho. Co s tím? Místo toho, abychom počítali přesný počet instrukcí, budeme ho počítat jen zhruba. Konkrétně nebudeme řešit konstanty, kterými $n$ násobíme a které k němu přičítáme. Důvodem, proč to není zas tak špatné řešení, je, že když bude vstup dostatečně dlouhý (a tedy $n$ dostatečně velké), budou konstanty zanedbatelně velké vůči něčemu, co závisí na $n$. Například pokud porovnáme časové složitosti $20n$ a $3n^2$, tak vidíme, že již pro poměrně malá $n$ budou konstanty zanedbatelné vzhledem k rozdílu způsobenému kvadratickou funkcí.

V našem algoritmu závisí počet instrukcí na $n$ lineárně – počet provedených instrukcí je nejvýše přímo úměrný počtu čísel na vstupu. Není těžké si uvědomit, že rychlejší algoritmus nemůže existovat – na každé z $n$ zadaných čísel se totiž musíme alespoň jednou podívat a jen na to potřebujeme alespoň $n$ instrukcí.

Abychom si tedy ulehčili práci a nemuseli jsme počet instrukcí počítat přesně, zavádíme takzvanou „óčkovou notaci“[4]. Óčková notace je horní odhad na počet instrukcí, které algoritmus provede, pokud zanedbáme multiplikativní konstanty a spustíme ho pro dostatečně velký vstup. [5] Pokud má tedy algoritmus časovou složitost $\mathcal{O}(n^2)$, znamená to, že pro dostatečně velká $n$ existuje konstanta $k$ taková, že algoritmus provede maximálně $k \cdot n^2$ instrukcí (například pro každé $n$ větší než 100 provede algoritmus maximálně $42 \cdot n^2$ instrukcí). Také si můžeme všimnout, že například algoritmus, který vždy provede $5 + 2n + 3n^4$ instrukcí běží v čase $\mathcal{O}(n^4)$, jelikož $5 + 2n + 3n^4 < 5n^4$ pro každé $n > 1$ (ze stejného důvodu bychom mohli tvrdit, že tentýž algoritmus běží v čase $\mathcal{O}(2^n)$, tato informace by ale pro nás nebyla příliš cenná – vždy se snažíme o co nejtěsnější odhad.) O našem algoritmu na hledání čísla v poli tedy můžeme říct, že má časovou složitost $\mathcal{O}(n)$.

V předchozím textu jsme vám trochu zalhali. Náš algoritmus umíme drobně upravit tak, že pro něj půjde najít lepší odhad než $\mathcal{O}(n)$. Stačilo by po každém průchodu cyklem zkontrolovat, jestli už jsme číslo našli, a pokud ano, ukončit provádění smyčky. Pokud by tedy první z čísel v posloupnosti bylo 42, algoritmus by provedl jen konstantně mnoho instrukcí (počet instrukcí by byl nezávislý na $n$), běžel by tedy v čase $\mathcal{O}(1)$. Takový případ ale pro nás není příliš zajímavý, protože nikdy nevíme, jaká čísla na vstupu dostaneme, a nemůžeme se při návrhu algoritmů spoléhat na to, že budeme mít pokaždé štěstí. Pokud tedy není řečeno jinak, budeme vždy určovat časovou složitost v nejhorším případě. Budeme tedy předpokládat, že nepřítel nám zadal vstup tak, aby náš algoritmus běžel co nejdéle. Taky si to můžeme představit tak, že spustíme algoritmus pro všechny vstupy dané délky a budeme počítat ten vstup, na který jsme potřebovali nejvíce instrukcí. S touto definicí složitosti už tedy opravdu platí naše dřívější poznámka, že najít číslo v poli $n$ čísel obecně neumíme rychleji než v čase $\mathcal{O}(n)$.

**Úloha 1** [2b]: *Vymyslete algoritmus, který zjistí, kde se v poli nachází největší číslo a jaké číslo to je. Můžete předpokládat, že největší číslo je v poli jen jedno.*

Tato úložka by měla jít vyřešit podobně jako ukázkový problém výše. Kdykoliv po vás chceme vymyslet algoritmus, rádi bychom slyšeli i jeho časovou složitost. To k algoritmu patří, protože to je v určitém slova smyslu to, co nám říká, jak je daný algoritmus dobrý.

Když už umíme najít maximum v poli, tak není těžké pole setřídit.

[4] Přestože pojem „óčková notace“ nezní úplně odborně, anglicky se toto značení skutečně nazývá „Big O notation“.

[5] Formální definici si můžete přečíst třeba na Wikipedii: `https://en.wikipedia.org/wiki/Big_O_Notation`

**Úloha 2** [3b]: *Vymyslete algoritmus, který setřídí pole čísel. Jinými slovy, chtěli bychom čísla seřadit podle jejich velikosti.*

Tento problém se vám pravděpodobně nepodaří vyřešit v čase $\mathcal{O}(n)$. Pokud ano, určitě nám takové řešení pošlete. Jedná se totiž o slavný otevřený problém, který se informatici snaží vyřešit již po desetiletí. A pokud máte pomalejší řešení, také ho určitě pošlete, i když si myslíte, že by to mohlo jít rychleji.

Lze dokázat, že třídit nejde rychleji než v $\mathcal{O}(n \log n)$, ale tento důkaz předpokládá, že si zakážeme s čísly dělat cokoliv kromě porovnávání. Kdybyste tento důkaz viděli, tak si vzpomeňte na tento předpoklad, díky kterému si důkaz neprotiřečí s tím, že bychom chtěli umět třídit v čase $\mathcal{O}(n)$.

**Bonusová verze** [+2b]: *Vyřešte ten samý problém, ale jen s konstantní pamětí navíc. To znamená, že máte v paměti pole, chcete ho setřídit a smíte při tom využívat libovolné konstantní (tedy nezávislé na $n$) množství proměnných. Už si ale třeba nemůžete pořídit pole, do kterého budete ukládat setříděnou posloupnost. To konkrétně znamená, že setříděná čísla budou na konci muset být v poli, ve kterém jste je dostali (na výstup nic nevypisujte).*

Na závěr formálněji popíšeme a odůvodníme chování našeho modelu. Pokud následující část nechcete číst, nijak vám to nebrání v řešení tématka, vážou se k ní ale některé další problémy.

## Formální popis modelu

Náš počítač bude mít *paměť*, která se skládá z nekonečně mnoha paměťových *buněk* očíslovaných celými čísly (představujte si ji jako nekonečnou posloupnost čtverečků). V každé buňce může být uloženo libovolně velké celé číslo. Číslu buňky budeme říkat *adresa*. Buňka může být určena dvěma způsoby: *Přímou adresací* – napíšeme adresu buňky do hranatých závorek (buňku s adresou 3 tedy značíme `[3]`), nebo *nepřímou adresací* – do dvojitých hranatých závorek napíšeme adresu buňky, ve které je uložena výsledná adresa buňky (buňku, jejíž adresa je uložena v buňce s adresou 3, tedy značíme `[[3]]`).

Každý algoritmus, který chceme na našem počítači spustit, musíme zapsat pomocí *instrukcí*. Počítači tedy předáme seznam instrukcí (ekvivalent kódu napsaného v programovacím jazyce pro skutečné počítače) a do určených paměťových buněk napíšeme vstup. Na reálném počítači by vstup pocházel z nějakého souboru nebo by ho zadal uživatel, ale my budeme pro jednoduchost předpokládat, že ho už máme někde uložený. Počítač poté provede všechny instrukce v takovém pořadí, v jakém jsou zapsány. Až mu instrukce dojdou, skončí. Počítač navíc může pomocí instrukce `print` postupně vypisovat čísla na výstup. Výstupem je pak posloupnost čísel, které vypsal za dobu svého běhu.

Nyní již přímo k instrukcím našeho počítače. Typická instrukce bere dva argumenty – každý z nich je (přímo či nepřímo adresovaná) paměťová buňka, nebo celé číslo. Pokud je argumentem buňka, instrukce si vždy přečte číslo, které je v ní uložené, a dále počítá s ním. S takto určenými celými čísly instrukce něco provede a výsledek uloží do předem určené paměťové buňky.

Samotné instrukce budou následující:

- Instrukce přiřazení $\leftarrow$ bere jeden argument a jeho hodnotu uloží do určené buňky. Například `[1]` $\leftarrow$ `[2]` přečte číslo z druhé paměťové buňky a uloží ho do první paměťové buňky.

- Instrukce $+$, $-$, $\cdot$ a / berou vždy dva argumenty, spočítají jejich součet, rozdíl, součin nebo podíl (zaokrouhlený dolů na celé číslo) a výsledek uloží do dané buňky. Například `[[4]]` $\leftarrow$ `[1]` $+$ `2` vezme číslo uložené v první buňce, přičte k němu dvojku a výsledek uloží do buňky, jejíž adresa je uložena v buňce číslo 4.

- Instrukce skoku nebere žádný argument, ale „skočí“ v našem programu na označené místo. Někam mezi instrukce tedy napíšeme klíčové slovo (třeba „bagr“). Pokud na jiném místě napíšeme instrukci „skok na bagr“, náš počítač začne provádět instrukce, které následují za slovem „bagr“. Můžeme tím tedy způsobit, že se některé instrukce přeskočí, nebo naopak provedou víckrát.

- Instrukce podmíněného příkazu bere dva argumenty, podmínku, kterou mají splňovat, a libovolnou jinou instrukci s jejími argumenty. Podmínka je vždy některý z matematických operátorů ($<$, $>$, $\geq$, $\leq$, $=$, $\neq$). Pokud je podmínka splněna, provede se daná instrukce. Např. „`Pokud [1]` $>$ `1, [2]` $\leftarrow$ `3`“ uloží do druhé buňky číslo 3, pokud je v první buňce uloženo číslo větší než 1.

- Instrukce `print` vypíše argument na výstup.

Tím je náš model kompletní.

Model jsme se snažili zavést tak, aby byl co nejjednodušší, ale zároveň uměl spočítat vše, co umí dnešní počítače, a to se stejnou časovou složitostí. Výhodou našeho nového modelu je, že má méně instrukcí a lépe se s ním pracuje matematicky (třeba se v něm lépe dokazuje, že problém nelze algoritmicky vyřešit), zatímco v tom předchozím se zase lépe uvažuje o programech. Můžete si zkusit rozmyslet, že ve skutečnosti není o nic slabší než náš dříve zavedený uživatelsky přívětivější model…

**Problém**: *Zkuste dokázat, že jakýkoliv program, který funguje na modelu, který jsme si pro větší pohodlnost dříve zavedli, lze přepsat tak, aby fungoval i na tomto minimalistickém modelu se stejnou časovou složitostí. Můžete to samozřejmě dokázat jen pro nějakou část – například vysvětlit, jak v minimalistickém modelu vytvořit pole, podmínku či cyklus.*

Možná vám přijde zvláštní, že jsme schopni na našem modelu počítat s libovolně velkými čísly v konstantním čase. Říkáte si, že to asi neodpovídá realitě, protože určitě bude existovat vážně velké číslo, které na opravdový počítač ani nepůjde uložit, natož aby s ním počítače uměly pracovat v konstantním čase (ať už to u reálných počítačů znamená cokoliv). Nám to také přijde zvláštní. Je na vás, abyste tento problém vyřešili. Zkuste náš model nebo způsob počítání složitosti nějak upravit, nebo si vymyslet vlastní výpočetní model a počítat časovou složitost na něm. Můžete samozřejmě popsat více různých možností.

**Problém**: *Navrhněte výpočetní model nebo způsob počítání časové složitosti, který bude odpovídat realitě v tom, že operace s dlouhými čísly trvají déle než s krátkými.*

Ptáte se, zda je to opravdu třeba? Zatím jsme si neukázali žádný způsob, jak by šlo konstantní operace s dlouhými čísly zneužít. Jedním z problémů, při jehož řešení to jde, je například již zmíněné třídění čísel, které lze v našem modelu provést v $\mathcal{O}(n)$, a tedy není tak úplně pravda, že by to byl slavný otevřený problém, jak jsme psali. Ale v libovolném rozumném modelu tomu tak je. Další takový problém je násobení matic, které lze zrychlit tím, že zakódujeme $n$ čísel do jednoho a poté umíme provést aritmetickou operaci na $n$ číslech v čase $\mathcal{O}(1)$. Tyto konstrukce však nejsou úplně jednoduché, a nebudeme je zde tedy ukazovat. Pilný řešitel si je může vymyslet sám a pak o tom napsat pro ostatní.

**Těžký problém**: *Vymyslete nějaký algoritmus, který zneužívá operace s dlouhými čísly v konstantním čase. (Řešení tohoto problému nebudeme zveřejňovat, bude tedy možnost posílat k němu řešení po celý školní rok.)*

*Kuba a Tom;* `domestomas+mam@gmail.com`
*e-mailová konference:* `algoritmy@mam.mff.cuni.cz`

Časopis M&M je zastřešen Matematicko-fyzikální fakultou Univerzity Karlovy. S obsahem časopisu je možné nakládat dle licence CC BY 3.0. Autory textů jsou, není-li uvedeno jinak, organizátoři M&M.

## Kontakty:

M&M, OPMK, MFF UK
Ke Karlovu 3
121 16 Praha 2

E-mail: `mam@matfyz.cz`
Web: `mam.matfyz.cz`
FB: `casopis.MaM`