*Milí řešitelé!*

Jaro se již krůček po krůčku blíží k nám. Vlastně se spíš plíží plůžek po plůžku… Pro ty z Vás, kteří i v tomto jarním čase mají chuť k řešení KSP, je tady další a v tomto roce již poslední série. S ní by Vám měla přijít i ročenka minulého ročníku.

K blížícímu se jaru patří jarní úklid. Ten se nevyhne ani našim webovým stránkám, kde brzy kromě zadání v HTML najdete i verzi pro tisk, která je ve stejném formátu jako ta, kterou právě držíte v ruce. A kromě jarního úklidu nám jaro přineslo také nového kamaráda ⟶

Odesílání řešení po Internetu alias Poštovský panáček zatím vypadá slibně, takže bude připraven přijímat i Vaše řešení čtvrté série. Ovšem pozor na to, že je pedant a řešení přijímá jen do dne odeslání včetně.

Aktuální informace o KSP můžete nalézt na Internetu na *http://ksp.mff.cuni.cz/*, dotazy organizátorům je možno posílat e-mailem na adresu *ksp@mff.cuni.cz*.

**Zadání čtvrté série šestnáctého ročníku KSP**

## 16-4-1 Mnichova posedlost aneb Hanoi strikes back — 10 bodů

Všichni jistě víte, co jsou to Hanojské věže. Jedná se o tři kolíky, na kterých je nasazeno celkem $N$ disků různých velikostí. Na začátku jsou všechny disky nasazeny na krajním kolíku a uspořádány podle velikosti (dole leží největší disk). Vaším úkolem je přemístit všechny disky na jiný kolík a přitom dodržet pravidlo, že vždy musí menší disk ležet na větším.

Mnichům v Tibetu se v časech dávno minulých zjevil sám Bůhdha a slíbil jim, že až přesunou v Hanojských věžích 64 disků z jednoho kolíku na jiný, svět bude u konce. A mnichové se snaží jím zadaný úkol splnit již po tisíciletí, protože k tomu potřebují (jak jistě víte) $2^{64} - 1$ přesunů.

Vzhledem k tomu, že je to práce velmi jednotvárná, stalo se jednou, že do mnicha Askejťáka vstoupil démon Kazisvět, který nechce, aby svět skončil (jinak by ho už nemohl dál kazit). Mnich pod vlivem Kazisvěta přesouval disky sice podle pravidla, že menší disk musí ležet na větším, avšak vůbec nedodržoval již tisíce let stanovený postup. Poté, co Kazisvět odešel, mniši zjistili, že vůbec nevědí, jak dál. Ale Vy jistě vědět budete.

Na vstupu dostanete počet disků $N$. Předpokládejte, že disky jsou očíslovány podle velikosti od největšího (1) k nejmenšímu ($N$). Dále dostanete popis situace na Hanojských věžích (jaké disky jsou na kterých kolících; uvědomte si, že pořadí disků na kolíku je určeno vždy jednoznačně). Vaším úkolem je říci, na jaký kolík bude pro mnichy nejvýhodnější všechny disky nakonec nasadit a kolik k tomu budou potřebovat kroků. Můžete pro jednoduchost předpokládat, že počet *potřebných* kroků bude menší než $2^{64} - 1$.

*Příklad:* Pokud $N = 7$ a na prvním kolíku jsou disky $6, 7$, na druhém kolíku disk 5 a na třetím kolíku disky $1, 2, 3, 4$, je pro mnichy nejvýhodnější přemístit všechny disky na kolík číslo tři. Dokážou to pomocí čtyř přesunů.

## 16-4-2 Technologické trable — 8 bodů

Jedna nejmenovaná počítačová společnost nedávno zveřejnila, že hodlá vrhnout na trh naprosto originální novinku mezi úložnými zařízeními. Její naprostá nepřekonatelná moderní nedostižná a prostě skvělá přednost tkví v tom, že bude mít nekonečnou kapacitu.

Toto úložné zařízení slouží k uchování *nekonečně* mnoha celých čísel. Jednotlivá čísla jsou uložena v buňkách, které jsou očíslovány přirozenými čísly, a navíc platí, že obsahy paměťových buněk jsou uloženy ve vzestupném pořadí.

Společnost se ovšem natolik vyčerpala několikatýdenní letákovou, novinovou, televizní a billboardovou propagandou, že již není schopná napsat program, který bude s propagovaným zařízením zacházet.

Vaším úkolem je napsat program, který najde zadanou hodnotu v nekonečně velkém vzestupně uspořádaném poli se začátkem. Pozor na to, že jednotlivé hodnoty se mohou opakovat. Protože toto pole je nekonečně velké, nemůžete ho dostat na vstupu. Hodnoty toho pole budete zjišťovat tak, že se budete zařízení ptát na hodnotu v konkrétní paměťové buňce (pro nás to znamená, že se budete ptát uživatele). Výsledkem programu má být číslo paměťové buňky, kde se hledaná hodnota vyskytuje, případně $-1$, pokud se hledaná hodnota v zařízení nevyskytuje vůbec.

Důležité na Vašem programu je to, kolik hodnot si z popisovaného zařízení vyžádá. Zkuste také dokázat, že Vaše řešení je optimální (pokud opravdu je).

*Příklad:* Hledaná hodnota je 15, odpovědi na dotazy jsou tyto: v buňce 1 je uložena hodnota $-8$, v buňce 3 je 14 a v buňce 4 je hodnota 16. Výsledek programu je $-1$.

## 16-4-3 Stávka programátorů — 10 bodů

*Pondělí ráno. Ústředí televize TeleNovela. Telefon zvoní. „Prosím?“ „Pane řediteli, hrozná věc! Programátoři stávkují, prý že i programátor má právo na lidská práva!“ „Ale my jsme televize TeleNovela. Tohle nás vůbec nezajímá.“*

*To samé pondělí ráno. Ústředí televize CNN. Telefon zvoní. „Prosím?“ „Pane řediteli, hrozná věc! Programátoři stávkují, prý že i programátor má právo na lidská práva!“ „To je ale hrůza! O tom musíme natočit reportáž!“*

Programátoři ve známém městě *Jsi-li con, válej* se rozhodli uspořádat stávku. Chtějí mít všechna lidská práva, hlavně právo dostat za práci přiměřenou odměnu. Většina z nich je totiž neúnosně přeplacena. A oni chtějí spravedlnost.

Město si můžeme představit jako pravoúhlou síť ulic, kterých je ve vodorovném i svislém směru $N$. Ulice si očíslujeme, ty vodorovné odspoda, ty svislé zleva. Každou křižovatku můžeme označit dvojicí $(x, y)$, kde $x$ je číslo ulice svislé a $y$ je číslo ulice vodorovné.

Programátoři budou postupně stávkovat na $M$ různých křižovatkách. A televize CNN chce o všech stávkách natočit reportáž. K tomu, aby natočili reportáž o stávce na křižovatce $(x, y)$ však nemusí být přímo na této křižovatce, ale stačí, když jsou ve svislé ulici $x$ nebo ve vodorovné ulici $y$ (mají totiž velmi dobré objektivy).

Televize CNN má ústředí na křižovatce $(cnn_x, cnn_y)$. Má k dispozici bohužel jediný vůz, který na začátku vyjede

z ústředí, nafilmuje všech $M$ stávek v pořadí, v jakém se budou konat, a vrátí se zpět do ústředí. Vaším úkolem je naplánovat jeho cestu tak, aby nafilmoval všechny stávky (jak už bylo řečeno, nemusí být přímo u stávky, stačí, když bude na ulici procházející křižovatkou stávky) a jeho cesta byla nejkratší možná.

*Příklad:* Pro $N = 4$, $M = 3$, stávky na křižovatkách v pořadí $(1, 4)$, $(4, 1)$, $(3, 4)$ a pro centrum CNN $(2, 2)$ je nejkratší možná cesta filmového vozu ←↓→→↑←.

## 16-4-4 Dlouhoprstova zapeklitá hra — 10 bodů

Dlouhoprstova hrabivost byla sice jeho vítězstvím nad piráty na chvilku utlumena, ale jeho sebevědomí poněkud stouplo. A tak si řekl, že když už je jednou na cestě do pekla, aby upsal ďáblu duši za pár stříbrňáků, tak tam dojde.

V pekle byl Dlouhoprst Satanem pěkně uvítán, ale návrh, že upíše svou duši, mu neprošel. Satan tvrdil, že Dlouhoprstova duše skončí tak jako tak v pekle, takže proč by mu za to měl ještě platit. Nicméně souhlasil, že si s Dlouhoprstem zahraje karetní hru.

Hraje se na jedné straně o Dlouhoprstův život, o truhlu stříbrňáků na straně druhé. Hry prší se samozřejmě v horoucím pekle bojí jako kříže a ani s mariášem Satan nesouhlasil. Viděl totiž Dlouhoprstovy rukávy nadité falešnými kartami. A tak Satan navrhl hru vlastní.

Hraje se se speciálními pekelnými kartami, každá karta je označená buď malým nebo velkým písmenem nebo číslicí. Satan před sebe položí dvě řady karet, obě v nějakém konkrétním pořadí. V jedné řadě je karet $N$, v druhé $M$. V každé řadě otočí nějaké karty vzhůru rubem a nějaké lícem. Každou řadu tedy můžeme zaznamenat jako posloupnost malých a velkých písmen a číslic (to jsou konkrétní karty), otazníků (práve jedna libovolná karta) a hvězdiček (0 a více libovolných karet).

Protože Satan ví o Dlouhoprstových falešných kartách, dostal Dlouhoprst za úkol vytáhnout z útrob svého obleku takovou *nejkratší* posloupnost karet, že může být stejná jako obě Satanovy řady karet. Případně má říci, že to není možné. Pokud je nejkratších posloupností více, stačí libovolná z nich.

*Příklad:* Pokud jsou Satanovy řady karet `t?2*f?` a `t*fg`, Dlouhoprst má vytáhnout například karty `tX2fg`.

## 16-4-5 Obchodníci s deštěm — 10 bodů

Pro svou práci na novém operačním systému potřebujete spolehlivý generátor pseudonáhodných čísel. A to takový, který bude generovat nelokální posloupnosti náhodných čísel. To jsou takové posloupnosti, jejichž členy jsou rozptýlené na celém používaném intervalu. Jinak řečeno, nejmenší vzdálenost mezi dvěma libovolnými prvky je pokud možno co největší.

Bohužel jediná firma, která generátory tohoto typu nabízí, je O. Š. Kubal a synové. Tu musíte určitě znát. Je to jediná společnost, která dokázala prodat tisíc kusů ledniček značky „Mrazík“ Eskymákům, patentovala si v USA perpetuum mobile a dokázala si prodat vlastní nos mezi svýma očima.

Ale protože znáte O. Š. Kubala i jeho syny a chcete, aby generátor doopravdy fungoval, rozhodli jste se ho nejprve vyzkoušet. A k tomuto účelu si potřebujete napsat následující program.

Na vstupu dostanete $N$ a $K$. Pak budete postupně načítat $N$ různých náhodných čísel. *Hned* po načtení jednoho náhodného čísla (kromě prvního) vypíšete, jaký je nejmenší rozdíl mezi libovolnými různými dvěma z posledních $K$ načtených náhodných čísel.

*Příklad:* Pro $N = 6$, $K = 3$ má vypadat vstup a výstup programu následovně:

| *náhodné číslo* | *aktuální nejmenší rozdíl* |
|---|---|
| 5 | |
| 7 | 2 |
| 4 | 1 |
| 15 | 3 |
| 6 | 2 |
| 20 | 5 |

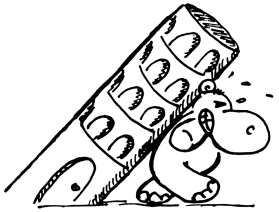
## 16-4-6 Šikmá věž v Kocourkově — 10 bodů

V Kocourkově se radní buřtipáni rozhodli, že si postaví šikmou věž. Všichni s jejich návrhem souhlasili a všichni chtěli pomoci. Silák se hned nabídl, že vykopá do země díru, aby byla věž šikmá, Bohemína tvrdila, že z hromady vajec dokáže udělat skvělou maltu, Zpívarotti slíbil, že za trochu piva zazpívá a všem půjde práce pěkně od ruky.

Brzy (po několika postavených věžích) ale zjistili, že to nebude tak jednoduché. Některé úkoly bylo totiž nutné vyřešit před tím, než jiné začnou. Silák bohužel nedokázal pod již postavenou věží vykopat dost velkou díru, aby věž byla šikmá. A i když Lamželezo byl silák a s velkým rozběhem zvládl vybourat dveře v již postavené zdi, postavit zeď i s dveřmi bylo přeci jen lepší.

Dále zjistili, že nějaké práce jsou *klíčové*. Zpoždění takových klíčových prací totiž vede ke zpoždění celé stavby. A protože chtějí mít Kocourkovští věž postavenou co nejdříve, poslali si pro Vás, abyste jim pomohli.

Váš program dostane na vstupu $N$, což je počet pracovních úkonů, které je třeba k postavení věže vykonat. U každého úkonu víte dobu, jakou trvá, a dále seznam jiných úkonů, které musí být dokončeny před tím, než začne práce na tomto úkonu. Výstupem programu by mělo být buď slovo „nelze“, pokud věž postavit vůbec nejde (například Zpívarotti zpívá jen za pivo a hospodský chce dát pivo jen za doušek pěkného zpěvu). Pokud věž postavit jde, měl by Váš program vypsat nejmenší dobu, za jakou mohou Kocourkovští věž postavit, a ty úkony, které jsou klíčové.



*Příklad:* $N = 5$, úkon 1 trvá 4 časové jednotky, úkon 2 trvá 2 časové jednotky, úkon 3 trvá 5 časových jednotek, úkon 4 trvá 10 časových jednotek a úkon 5 trvá 2 časové jednotky. Úkon 2 je možné začít, až když jsou hotové úkony 1 a 3. Úkon 4 může začít až po skončení úkonů 5 a 2.

Tehdy lze věž postavit za 17 časových jednotek a klíčové jsou úkony 2, 3 a 4.

Svá řešení nám zasílejte do 31. května 2004 na adresu:

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

## Výsledková listina šestnáctého ročníku KSP po třetí sérii

| | | škola | ročník | 1631 | 1632 | 1633 | 1634 | 1635 | 1636 | suma | celkem |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1. | Miroslav Cicko | GJGTajov | 3 | 10 | 10 | 8 | 8 | 9 | | 45 | 126 |
| 2. | Peter Perešíni | GJGTajov | 2 | 9 | 10 | 10 | | 10 | | 39 | 125 |
| 3. | Petr Škoda | GÚstavní | 4 | 6 | 10 | 8 | 7 | | 9 | 40 | 119 |
| 4. | Jan Bulánek | G Klatovy | 3 | 8 | | 10 | 4 | 9 | | 31 | 111 |
| 5. | Kryštof Hoder | GKptJaroš | 4 | 9 | | 10 | | 10 | | 29 | 94 |
| 6. | David Matoušek | GZborov | 4 | 6 | 10 | 10 | 10 | 5 | | 41 | 93 |
| 7. | Marek Jančuška | G Nitra | 4 | | | | | | | 0 | 88 |
| 8. | Michal Repovský | | 4 | 6 | | 10 | 4 | 2 | | 22 | 83 |
| 9. | Jana Kravalová | G VKlobou | 4 | 6 | 2 | 1 | 6 | 6 | | 21 | 79 |
| 10. | Miroslav Klimoš | G Lanškr | 0 | 6 | | 7 | 7 | 3 | | 23 | 78 |
| 11. | Pavel Motloch | GPBezruče | 1 | 6 | | 8 | | | | 14 | 77 |
| 12. | Ondřej Bílka | G Zlín | 2 | 6 | 8 | | | 8 | | 22 | 73 |
| 13. | Ondřej Garncarz | G Příbor | 3 | 5 | 2 | 7 | 1 | 5 | 1 | 21 | 65 |
| 14. – 16. | Jana Fabriková | GKptJaroš | 4 | | | 5 | 7 | 6 | | 18 | 62 |
| | Jan Hrnčíř | GFXŠaldy | 2 | 2 | 1 | 2 | 1 | 5 | | 11 | 62 |
| | Pavel Klavík | G Chrudim | 1 | 2 | 5 | 8 | 8 | 4 | | 27 | 62 |
| 17. | Zbyněk Falt | GNeumannov | 3 | 6 | | | | | | 6 | 57 |
| 18. | Martin Podloucký | G Strážnic | 3 | 5 | | 5 | | | | 10 | 55 |
| 19. | Marek Blahuš | G UHradi | 3 | 3 | | 7 | | 10 | | 20 | 54 |
| 20. | Stanislav Haviar | G Klatovy | 3 | 3 | | 8 | | 6 | | 17 | 50 |
| 21. – 22. | Martin Koníček | G UBrod | 3 | 4 | | 7 | | | | 11 | 49 |
| | Peter Šufliarsky | G NZámky | 4 | | | 1 | 6 | 6 | | 13 | 49 |
| 23. | Michal Bečka | G MTřebová | 4 | 6 | | | | | | 6 | 48 |
| 24. | Peter Černo | GĽŠtúra | 3 | | | | | | | 0 | 46 |
| 25. – 28. | Martin Čech | G UBrod | 3 | 2 | | 1 | | 4 | | 7 | 43 |
| | Tomáš Gavenčiak | G Bílovec | 4 | | | | | | | 0 | 43 |
| | Eva Schlosáriková | G Piešťany | 3 | 4 | | 2 | 6 | 9 | | 21 | 43 |
| | Filip Šauer | G Klatovy | 3 | 2 | | 2 | | 5 | | 9 | 43 |
| 29. | Daniel Marek | GZborov | 1 | | | | | | | 0 | 42 |
| 30. | Martina Tomisová | GZborov | 4 | 4 | | 4 | | | | 8 | 40 |
| 31. | Petr Soběslavský | GJHeyrovs | 3 | 1 | 0 | 2 | | | | 3 | 38 |
| 32. – 33. | Petr Kortánek | G Sedlča | 2 | 6 | | | | | | 6 | 37 |
| | Martin Křivánek | GKptJaroš | 2 | | | | | | | 0 | 37 |
| 34. | Ján Záhornadský | GZborov | 3 | | | | | | | 0 | 36 |
| 35. | Petr Švec | G Beroun | 4 | | | | | | | 0 | 35 |
| 36. | Jindřich Flídr | G Lanškr | 4 | | | | | | | 0 | 33 |
| 37. | Jaroslav Havlín | G Sedlča | 4 | | | | | | | 0 | 32 |
| 38. – 40. | Petr Kratochvíl | | 1 | 5 | | 7 | | 4 | | 16 | 30 |
| | Marek Ludha | GJGTajov | 4 | | | | | | | 0 | 30 |
| | Adam Přenosil | GSladkNám | 2 | | | | | | | 0 | 30 |
| 41. | Cyril Hrubiš | G Bílovec | 2 | 3 | 3 | | 6 | 1 | | 13 | 27 |
| 42. | Martin Dobroucký | G MTřebová | 3 | 6 | | 8 | 3 | 9 | | 26 | 26 |
| 43. – 45. | Stanislav Basovník | G Kroměříž | 3 | | | | | | | 0 | 24 |
| | Jan Křetínský | GMLercha | 4 | | | | | | | 0 | 24 |
| | Martin Kupec | GMendel | 2 | 2 | | 7 | 3 | 1 | | 13 | 24 |
| 46. | Benjamin Vejnar | G Nymburk | 4 | | | | | | | 0 | 20 |
| 47. | Petr Musil | G MBuděj | 2 | 5 | 0 | | | | | 5 | 17 |
| 48. | Milan Dvořák | G NMnMor | 1 | | | | | | | 0 | 16 |
| 49. – 50. | Jiří Bělohradský | | 2 | | | | | | | 0 | 13 |
| | Jan Richter | G Příbor | 3 | | | | | | | 0 | 13 |
| 51. | Kristýna Knapová | G Jičín | 4 | | | | | | | 0 | 12 |
| 52. – 53. | Miroslav Kratochvíl | G Čáslav | 2 | 2 | | 6 | | 2 | | 10 | 10 |
| | Michal Potfaj | G NMnVáh | 4 | | | | | | | 0 | 10 |
| 54. – 55. | Zbyněk Konečný | GKptJaroš | 1 | | | | | 9 | | 9 | 9 |
| | Martin Schmid | G ČTřebová | 0 | | | | | | | 0 | 9 |
| 56. | Jindřich Pergler | G Klatovy | 3 | | | | | 5 | | 5 | 8 |
| 57. – 59. | David Irschik | G Ledeč | 3 | | | | | | | 0 | 7 |
| | Petr Paščenko | GDašická | 4 | | | | | | | 0 | 7 |
| | Jaromír Vojíř | G Ledeč | 3 | | | | | | | 0 | 7 |
| 60. | Radoslav Sopoliga | G Svidník | 4 | | | | | | | 0 | 6 |
| 61. | Tomáš Herceg | | 1 | | | | | | | 0 | 4 |
| 62. | Aleš Razým | | 3 | | | | | | | 0 | 2 |



## Recepty z programátorské kuchařky

V nedávném vydání programátorské kuchařky jsme se zabývali tříděním dat, tentokráte si povíme, jak v uspořádaných datech něco efektivně najít a jak si data udržovat stále uspořádaná. K tomu se nám bude hodit zejména binární vyhledávání a různé druhy vyhledávacích stromů.

**Binární vyhledávání.** Představte si, že jste k narozeninám dostali obrovské pole setříděných záznamů (to je, pravda, trochu netradiční dárek, ale proč ne – může to být třeba telefonní seznam). Záznamy mohou vypadat libovolně a to, že jsou setříděné, znamená jen a pouze, že $x_1 < x_2 < \ldots < x_N$, kde $<$ je nějaká relace, která nám řekne, který ze dvou záznamů je menší (pro jednoduchost předpokládáme, že žádné dva záznamy nejsou stejné).

Co si ale s takovým polem počneme? Zkusíme si v něm najít nějaký konkrétní záznam $z$. To můžeme udělat třeba tak, že si nalistujeme prostřední záznam (označíme si ho $x_m$) a porovnáme s ním naše $z$. Pokud $z < x_m$, víme, že se $z$ nemůže vyskytovat „napravo“ od $x_m$, protože tam jsou všechny záznamy větší než $x_m$ a tím spíše než $z$. Analogicky pokud $z > x_m$, $z$ se nemůže vyskytovat v první polovině pole. V obou případech nám zbude jedna polovina a v ní budeme pokračovat stejným způsobem. Tak budeme postupně zmenšovat interval, ve kterém se $z$ může nacházet, až buďto $z$ najdeme nebo vyloučíme všechny prvky, kde by mohlo být.

Tomuto principu se obvykle říká *binární vyhledávání* nebo také *hledání půlením intervalu* a snadno ho naprogramujeme buďto rekursivně nebo pomocí cyklu, v němž si budeme udržovat interval $\langle l, r\rangle$, ve kterém se hledaný prvek může nacházet:

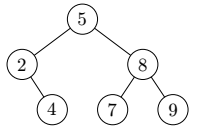
```
function BinSearch(z : integer):integer;
var l,r,m : integer;
begin
  l := 1;   { interval, ve kterém hledáme }
  r := N;
  while l <= r do begin { ještě není prázdný }
    m := (l+r) div 2;   { střed intervalu }
    if z < x[m] then
      r := m-1          { je vlevo }
    else if z > x[m] then
      l := m+1          { je vpravo }
    else begin          { Bingo! }
      hledej := m;
      exit;
    end;
  end;
  hledej := -1;         { nebyl nikde }
end;
```

Všimněte si, že průchodů cyklem `while` může být nejvýše $\lceil \log_2 N \rceil$, protože interval $\langle l, r\rangle$ na počátku obsahuje $N$ prvků a v každém průchodu jej zmenšíme na polovinu (ve skutečnosti ještě o jedničku, ale tím lépe pro nás), takže po $k$ průchodech bude interval obsahovat nejvýše $N/2^k$ prvků. Jenže pokud by $k$ bylo větší než $\log_2 N$, bude $N/2^k < 1$ a tehdy se algoritmus určitě musí zastavit. Časová složitost binárního vyhledávání tedy bude $O(\log N)$. [Základ logaritmu nemusíme psát, protože $\log_a b = \log_c b / \log_c a$, čili logaritmy o různých základech se liší jen konstantou, která se „schová do $O$-čka.“]

Hledání půlením intervalu je tedy velmi rychlé, pokud máme možnost si data předem setřídit. Jakmile ale potřebujeme za běhu programu přidávat a odebírat záznamy, potážeme se se zlou: buďto budeme mít záznamy uložené v poli, a pak nezbývá než při zatřiďování nového prvku ostatní „rozhrnout“, což může trvat až $N$ kroků, a nebo si je budeme udržovat v nějakém seznamu, do kterého dokážeme přidávat v konstantním čase, jenže pak pro změnu nebudeme při vyhledávání schopni najít tolik potřebnou polovinu.

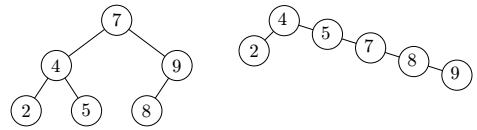
Zkusme ale provést jednoduchý myšlenkový pokus:

**Vyhledávací stromy.** Představme si, jakými všemi možnými cestami se může v našem poli binární vyhledávání ubírat. Na začátku porovnáváme s prostředním prvkem a podle výsledku se vydáme jednou ze dvou možných cest (nebo rovnou zjistíme, že se jedná o hledaný prvek, ale to není moc zajímavý případ). Na každé cestě nás zase čeká porovnání se středem příslušného intervalu a to nás opět pošle jednou ze dvou dalších cest atd. To si můžeme přehledně popsat pomocí stromu:




Jeden vrchol stromu prohlásíme za kořen a ten bude odpovídat celému poli (a jeho prostřednímu prvku). K němu budou připojené vrcholy obou polovin pole (opět obsahující příslušné prostřední prvky) a tak dále. Ovšem jakmile známe tento strom, můžeme náš půlící algoritmus provádět přímo podle stromu (ani k tomu nepotřebujeme vidět původní pole a umět v něm hledat poloviny): začneme v kořeni, porovnáme a podle výsledku se buďto přesuneme do levého nebo pravého podstromu a tak dále. Každý průběh algoritmu bude tedy odpovídat nějaké cestě z kořene stromu do hledaného vrcholu $z$.

Teď si ale všimněte, že aby hledání hodnoty podle stromu fungovalo, strom vůbec nemusel vzniknout přesně půlením intervalu – stačilo, aby v každém vrcholu platilo, že všechny hodnoty v levém podstromu jsou menší než tento vrchol a naopak hodnoty v pravém podstromu větší. Hledání v témže poli by tedy popisovaly následující stromy (např.):




Hledací algoritmus podle jiných stromů samozřejmě už nemusí mít pěknou logaritmickou složitost (když bychom hledali podle „degenerovaného“ stromu z pravého obrázku, trvalo by to dokonce lineárně). Důležité ale je, že takovéto stromy se dají poměrně snadno modifikovat a že je při troše šikovnosti můžeme udržet dostatečně podobné ideálnímu půlení intervalu. Pak bude hloubka stromu stále $O(\log N)$ (a tedy i časová složitost hledání a, jak za chvilku uvidíme, i dalších operací).

**Definice.** Zkusme si tedy pořádně nadefinovat to, co jsme právě vymysleli:

*Binární vyhledávací strom* (po domácku BVS) je buďto prázdná množina nebo *kořen* obsahující jednu hodnotu a mající dva *podstromy* (levý a pravý), což jsou opět BVS,

ovšem takové, že všechny hodnoty uložené v levém podstromu jsou menší než hodnota v kořeni, a ta je naopak menší než všechny hodnoty uložené v pravém podstromu.

*Úmluva:* Pokud $x$ je kořen a $L_x$ a $R_x$ jeho levý a pravý podstrom, pak kořenům těchto podstromů (pokud nejsou prázdné) budeme říkat *levý a pravý syn* vrcholu $x$ a naopak vrcholu $x$ budeme říkat *otec* těchto synů. Pokud je některý z podstromů prázdný, pak vrchol $x$ příslušného syna nemá. Vrcholu, který nemá žádné syny, budeme říkat *list* vyhledávacího stromu. Všimněte si, že pokud $x$ má jen jediného syna, musíme stále rozlišovat, je-li to syn levý nebo pravý, protože potřebujeme udržet správné uspořádání hodnot. Také si všimněte, že pokud známe syny každého vrcholu, můžeme již rekonstruovat všechny podstromy.

Každý BVS také můžeme popsat velmi jednoduchou strukturou v paměti:

```
type pvrchol = ^vrchol;
     vrchol = record
       l, r : pvrchol;  { levý a pravý syn }
       x    : integer;  { hodnota }
     end;
```

Pokud některý ze synů neexistuje, zapíšeme do příslušné položky hodnotu *nil*.

**Find.** V řeči BVS můžeme přeformulovat náš vyhledávací algoritmus takto:

```
function TreeFind(v:pvrchol; x:integer):pvrchol;
{ Dostane kořen stromu a hodnotu. Vrátí vrchol,
  kde se hodnota nachází, nebo nil, není-li. }
begin
  while (v<>nil) and (v^.x<>x) do begin
    if x<v^.x then
      v := v^.l
    else
      v := v^.r
  end;
  TreeFind := v;
end;
```

Funkce `TreeFind` bude pracovat v čase $O(h)$, kde $h$ je hloubka stromu, protože začíná v kořeni a v každém průchodu cyklem postoupí o jednu hladinu níže.

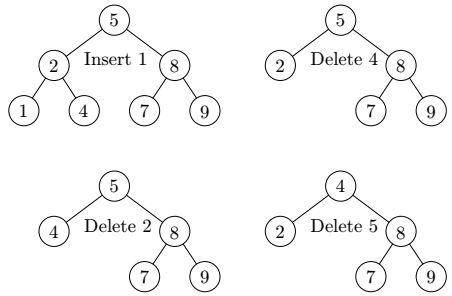
**Insert.** Co kdybychom chtěli do stromu vložit novou hodnotu (aniž bychom se teď starali o to, zda tím strom nemůže degenerovat)? Stačí zkusit hodnotu najít a pokud tam ještě nebyla, určitě při hledání narazíme na odbočku, která je *nil*. A přesně na toto místo připojíme nově vytvořený vrchol, aby byl správně uspořádán vzhledem k ostatním vrcholům (že tomu tak je, plyne z toho, že při hledání jsme postupně vyloučili všechna ostatní místa, kde nová hodnota být nemohla). Naprogramujeme opět snadno, tentokráte si ukážeme rekurzivní zacházení se stromy:

```
function TreeIns(v:pvrchol; x:integer):pvrchol;
{ Dostane kořen stromu a hodnotu ke vložení,
  vrátí nový kořen. }
begin
  if v=nil then begin
    { prázdný strom => založíme nový kořen }
    new(v);
    v^.l := nil;
    v^.r := nil;
    v^.x := x;
  end else if x<v^.x then  { vkládáme vlevo }
    v^.l := TreeIns(v^.l, x)
  else if x>v^.x then      { vkládáme vpravo }
    v^.r := TreeIns(v^.r, x);
  TreeIns := v;
end;
```

**Delete.** Mazání bude o kousíček pracnější, musíme totiž rozlišit tři případy: Pokud je mazaný vrchol list, stačí ho vyměnit za *nil*. Pokud má právě jednoho syna, stačí náš vrchol $v$ ze stromu odstranit a syna přepojit k otci $v$. A pokud má syny dva, najdeme největší hodnotu v levém podstromu (tu najdeme tak, že půjdeme jednou doleva a pak pořád doprava), umístíme ji do stromu namísto mazaného vrcholu a v levém podstromu ji pak smažeme (což už umíme, protože má 1 nebo 0 synů). Program následuje:

```
function TreeDel(v:pvrchol; x:integer):pvrchol;
{ Parametry stejně jako TreeIns }
var w:pvrchol;
begin
  TreeDel := v;
  if v=nil then exit         { prázdný strom }
  else if x<v^.x then
    v^.l := TreeDel(v^.l, x) { ještě hledáme x }
  else if x>v^.x then
    v^.r := TreeDel(v^.r, x)
  else begin                 { našli jsme }
    if (v^.l=nil) and (v^.r=nil) then begin
      TreeDel := nil;        { mažeme list }
      dispose(v);
    end else if v^.l=nil then begin
      TreeDel := v^.r;       { jen pravý syn }
      dispose(v);
    end else if v^.r=nil then begin
      TreeDel := v^.l;       { jen levý }
      dispose(v);
    end else begin           { má oba syny }
      w := v^.l;             { hledáme max(L) }
      while w^.r<>nil do w := w^.r;
      v^.x := w^.x;          { prohazujeme }
      { a mažeme původní max(L) }
      v^.l := TreeDel(v^.l, w^.x);
    end;
  end;
end;
```

Když do stromu z našeho prvního obrázku zkusíme přidávat nebo z něj odebírat prvky, dopadne to takto:




Jak vkládání, tak mazání opět budou trvat $O(h)$. Ale pozor, jejich používáním může $h$ nekontrolovatelně růst – sami zkuste najít nějaký příklad, kdy $h$ dosáhne až $N$.



**Úloha 16-3-5 – Botanikova setba – program**

```
#include <stdio.h>

#define N 4
#define M 5
int p[N][M];                                        /* p[y][x]  */
int c[M];                                           /* součty sloupečků  */
int ay, by;
int max=-100000, maxax, maxay, maxbx, maxby, maxobsah;

void
na_primce (void)
{
        int ax=0, bx, h=0;
        for (bx=0; bx<M; bx++) {
                int obsah = (bx-ax)*(by-ay);
                h+=c[bx];
                if ( (h > max) || ( (h == max) && (obsah<maxobsah))) {
                        maxax=ax; maxay=ay; maxbx=bx; maxby=by; max=h;
                        maxobsah = obsah;
                }
                if (h<=0)
                        h=0, ax=bx+1;
        }
}

int
main (void)
{
        int i;
        for (i=0; i<N*M; i++)
                scanf ("%d␣", &p[0][i]);
        for (ay=0; ay<N; ay++) {
                for (i=0; i<M; i++) c[i] = 0;
                for (by=ay; by<N; by++) {
                        for (i=0; i<M; i++) c[i] += p[by][i];
                        na_primce ();
                }
        }
        printf ("Hledaná podmatice má levý horní roh v %d. sloupci, %d. řádku"
                "a pravý dolní v %d. sloupci a %d. řádku.\n", maxax+1, maxay+1, maxbx+1, maxby+1);
        return 0;
}
```

**Úloha 16-3-4 – Ekonomova odměna – program**

```
#include <stdio.h>
#define MAX_P 100

int coins[MAX_P];                                   /* toto jsou "jistoty" pirátů */
int unsure_thing[MAX_P];                            /* zda má pirát naději na víc než na "jistotu" */
int freqs[3];                                       /* kolik pirátů s jistotami -1,0,1 existuje */
int S, P;

int main (void) {
  int i, j, paid;                                   /* paid – tolik musí Dlouhoprst platit */
  int needed, have;                                 /* kolik potřebuji a mám pro mě hlasujících pirátů */
  int whom_to_pay;                                  /* kterým pirátům platím */
  int not_paid_to;                                  /* kolika pirátům chtějícím whom_to_pay jsem neplatil */

  printf ("Zadejte počet Pirátů a Stříbrňáků:");
  scanf ("%d %d", &P, &S);

  if (S >= (2* ( (P+1)/2) - 1)) {                   /* S je dostatečné – známe řešení */
    switch (P) {
    case 0:printf ("Nesmysl, žádný pirát neexistuje.\n"); break;
    case 1:printf ("Dlouhoprst zemře.\n"); break;
    case 2:printf ("Návrh je nedat nikomu nic, zisk je %d.\n", S); break;
    case 3:printf ("Návrh je dát prvním dvěma pirátům 1 stříbrňák, zisk je %d.\n", S-2); break;
    default:printf ("Návrh je dát prvním %d pirátům dva stříbrňáky,"
                    "pirátovi %d jeden a ostatním nic. Zisk je %d.\n", (P+1)/2-1, P-1, S-2* ( (P+1)/2)+1);
    }
    return 0;
  }

  for (i=0; i<=P; i++) {                            /* S je malé – simuluj */
    needed= (i+1)/2;
    have=paid=0;
    whom_to_pay=-1;
    while (whom_to_pay<2) {
      if (have+freqs[whom_to_pay+1]>=needed) {
        paid+= (needed-have)* (whom_to_pay+1);
        not_paid_to=have+freqs[whom_to_pay+1]-needed;
        break;
      }
      have+=freqs[whom_to_pay+1];
      paid+=freqs[whom_to_pay+1]* (whom_to_pay+1);
      whom_to_pay++;
    }
    if (paid > S) whom_to_pay=2;                    /* nemám na to dost peněz */
    if (whom_to_pay<2) {                            /* šlo to uplatit */
      for (j=0; j<i; j++) {
        if (coins[j]<2) freqs[coins[j]+1]--;
        if (coins[j]>whom_to_pay) coins[j]=unsure_thing[j]=0;
        else if (! (coins[j]==whom_to_pay && not_paid_to)) coins[j]++, unsure_thing[j]=0;
        else unsure_thing[j]=1;
        if (coins[j]<2) freqs[coins[j]+1]++;
      }
      coins[i]=S-paid;
    } else {                                        /* nešlo to uplatit */
      coins[i]=-1;
    }
    if (coins[i]<2) freqs[coins[i]+1]++;
  }
  if (coins[P]==-1) puts ("Dlouhoprst přežít nemůže.");
  else {
    printf ("Dlouhoprst přežít může a vydělá si %d stříbrňáků.\nNávr je tento:", S-paid);
    not_paid_to=needed-have;
    for (i=0; i<P; i++) printf ("%d␣", (unsure_thing[i] && not_paid_to>0) ?
                                (not_paid_to--, coins[i]+1) : (unsure_thing[i]) ?0:coins[i]);
  } return 0; }
```



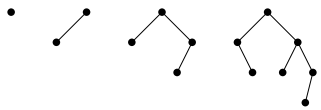
**Procházení stromu.** Pokud bychom chtěli všechny hodnoty ve stromu vypsat, stačí strom rekursivně projít a sama definice uspořádání hodnot ve stromu nám zajistí, že hodnoty vypíšeme ve vzestupném pořadí: nejdříve levý podstrom, pak kořen a pak podstrom pravý. Časová složitost je, jak se snadno nahlédne, lineární, protože strávíme konstantní čas vypisováním každého prvku a těch je právě $N$. Program opět snadný:

```
procedure TreeShow(v:pvrchol);
begin
  if v=nil then exit;  { není co dělat }
  TreeShow(v^.l);
  writeln(v^.x);
  TreeShow(v^.r);
end;
```

**Vyvážené stromy.** S binárními stromy lze dělat všelijaká kouzla. Ale prakticky všechny stromové algoritmy mají společné to, že jejich časová složitost je lineární v hloubce stromu. (Pravda, právě ten poslední byl výjimka, ale všechny prvky opravdu rychleji než lineárně s $N$ nevypíšeme.) Jenže jak jsme viděli, neopatrným insertováním a deletováním prvků mohou snadno vznikat všelijaké degenerované stromy, které mají lineární hloubku. Abychom tomu zabránili, musíme stromy *vyvažovat*. To znamená definovat si nějaké šikovné omezení na tvar stromu, aby hloubka byla vždy $O(\log N)$. Možností je mnoho, my uvedeme jen ty nejdůležitější:

*Dokonale vyvážený* budeme říkat takovému stromu, ve kterém pro každý vrchol platí, že počet vrcholů v jeho levém a pravém podstromu se liší nejvýše o jedničku. Takové stromy kopírují dělení na poloviny při binárním vyhledávání, a proto (jak jsme již dokázali) mají vždy logaritmickou hloubku. Jediné, čím se liší, je, že mohou zaokrouhlovat na obě strany, zatímco náš půlící algoritmus zaokrouhloval polovinu vždy dolů, takže levý podstrom nemohl být nikdy větší než pravý. Z toho také plyne, že se snadnou modifikací půlícího algoritmu dá dokonale vyvážený BVS v lineárním čase vytvořit ze setříděného pole. Bohužel se ale při Insertu a Deletu nedá v logaritmickém čase strom znovu vyvážit.

**AVL stromy.** Zkusíme tedy vyvažovací podmínku trochu uvolnit a vyžadovat pouze, aby se u každého vrcholu lišily o jedničku nikoliv velikosti podstromů, nýbrž jejich hloubky. Takovým stromům se říká *AVL stromy* a mohou vypadat třeba takto:



Každý dokonale vyvážený strom je také AVL stromem, ale jak je vidět na předchozím obrázku, opačně to platit nemusí. To, že hloubka AVL stromů je také logaritmická, proto není úplně zřejmé a zaslouží si to trochu dokazování:

*Věta:* AVL strom o $N$ vrcholech má hloubku $O(\log N)$.

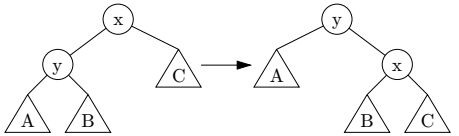
*Důkaz:* Označme $A_d$ nejmenší možný počet vrcholů, jaký může být v AVL stromu hloubky $d$. Snadno vyzkoušíme, že $A_1 = 1$, $A_2 = 2$, $A_3 = 4$ a $A_4 = 7$ (příslušné minimální stromy najdete na předchozím obrázku). Navíc platí, že $A_d = 1 + A_{d-1} + A_{d-2}$, protože každý minimální strom hloubky $d$ musí mít kořen a 2 podstromy, které budou opět minimální, protože jinak bychom je mohli vyměnit za minimální a tím snížit počet vrcholů celého stromu. Navíc alespoň jeden z podstromů musí mít hloubku $d-1$ (protože jinak by hloubka celého stromu nebyla $d$) a druhý hloubku $d-2$ (podle definice AVL stromu může mít $d-1$ nebo $d-2$, ale s menší hloubkou bude mít evidentně méně vrcholů).

Spočítat, kolik přesně je $A_d$, není úplně snadné, ale nám bude stačit dokázat, že $A_d \geq 2^{n/2}$. To provedeme indukcí: Pro $d < 4$ to plyne z ručně spočítaných hodnot. Pro $d \geq 4$ je pak $A_d = 1 + A_{d-1} + A_{d-2} > 2^{(d-1)/2} + 2^{(d-2)/2} = 2^{d/2}\cdot(2^{-1/2}+2^{-1}) > 2^{d/2}$ (součet čísel v závorce je $\approx 1.207$).

Jakmile ale víme, že $A_d$ roste s $d$ alespoň exponenciálně, tedy že $\exists c : A_d \geq c^d$, důkaz je u konce: Máme-li AVL strom $T$ na $N$ vrcholech, najdeme si nejmenší $d$ takové, že $A_d \leq N$. Hloubka stromu $T$ může být maximálně $d$, protože jinak by $T$ musel mít alespoň $A_{d+1}$ vrcholů, ale to je více než $N$. A jelikož $A_d$ rostou exponenciálně, je $d \leq \log_c N$, čili $d = O(\log N)$. *Q.E.D.*

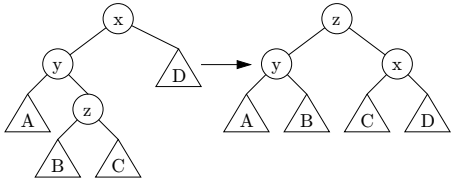
AVL stromy tedy vypadají nadějně, ale ještě stále nevíme, jak provádět Insert a Delete tak, aby AVL vyváženost zachovávaly. Nemůžeme si totiž dovolit strukturu stromu měnit libovolně – stále musíme dodržovat správné uspořádání hodnot. K tomu se nám bude hodit zavést si nějakou množinu operací, o kterých dokážeme, že jsou korektní, a pak strukturu stromu měnit vždy jen pomocí těchto operací. Budou to:

**Rotace.** Rotací binárního stromu (respektive nějakého podstromu) nazveme jeho „překořenění“ za některého ze synů kořene. Místo formální definice ukažme raději obrázek:




Strom jsme překořenili za vrchol $y$ a přepojili jednotlivé podstromy tak, aby byly vzhledem k $x$ a $y$ opět uspořádané správně (všimněte si, že je jen jediný způsob, jak to udělat). Jelikož se tím okolí vrcholu $y$ „otočilo“ po směru hodinových ručiček, říká se takové operaci *rotace doprava*. Inverzní operaci (tj. překořenění za pravého syna kořene) se říká *rotace doleva* a na našem obrázku odpovídá přechodu zprava doleva.

**Dvojrotace.** Také si nakreslíme, jak to dopadne, když provedeme dvě rotace nad sebou lišící se směrem (tj. jednu levou a jednu pravou nebo opačně). Tomu se říká *dvojrotace* a jejím výsledkem je překořenění podstromu za vnuka kořene připojeného „cikcak“. Raději opět předvedeme na obrázku:




**Znaménka.** Při vyvažování se nám bude hodit pamatovat si u každého vrcholu, v jakém vztahu jsou hloubky jeho podstromů. Tomu budeme říkat *znaménko* vrcholu a bude buďto 0, jsou-li oba stejně hluboké, – pro levý podstrom

hlubší a + pro pravý hlubší. V textu budeme znaménka, respektive vrcholy se znaménky značit $\oplus$, $\ominus$ a $\odot$.

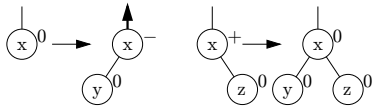
Pokud celý strom zrcadlově obrátíme (prohodíme levou a pravou stranu), znaménka se změní na opačná ($\oplus$ a $\ominus$ se prohodí, 0 zůstane). Toho budeme často využívat a ze dvou zrcadlově symetrických situací popíšeme jenom jednu s tím, že druhá se v algoritmu zpracuje symetricky.

Často také budeme potřebovat nalézt otce nějakého vrcholu. To můžeme zařídit buďto tak, že si do záznamů popisujících vrcholy stromu přidáme ještě ukazatele na otce a budeme ho ve všech operacích poctivě aktualizovat, a nebo využijeme toho, že jsme do daného vrcholu museli někudy přijít z kořene, celou cestu z kořene si zapamatujeme v nějakém zásobníku a postupně se budeme vracet.

Tím jsme si připravili všechny potřebné ingredience, tož s chutí do toho:

**Vyvažování po Insertu.** Když provedeme Insert tak, jak jsme ho popisovali u obecných vyhledávacích stromů, přibude nám ve stromu list. Pokud se tím AVL vyváženost neporušila, stačí pouze opravit znaménka na cestě z nového listu do kořene (všude jinde zůstala zachována). Pakliže porušila, musíme s tím něco provést, konkrétně ji šikovně zvolenými rotacemi opět opravit. Popíšeme algoritmus, který bude postupovat od listu ke kořeni a vše potřebné zařídí:

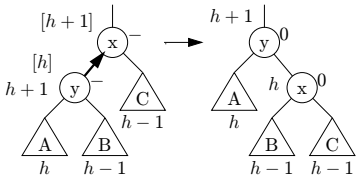
Nejprve přidání listu samotné:



Pokud jsme přidali list (bez újmy na obecnosti levý, jinak vyřešíme zrcadlově) vrcholu se znaménkem $\odot$, změníme znaménko na $\ominus$ a pošleme o patro výš informaci o tom, že hloubka podstromu se zvýšila (to budeme značit šipkou). Přidali-li jsme list k $\oplus$, změní se na $\odot$ a hloubka podstromu se nemění, takže můžeme skončit.

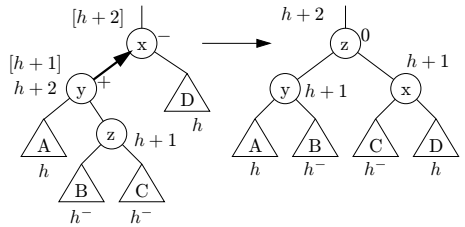
Nyní rozebereme případy, které mohou nastat na vyšších hladinách, když nám z nějakého podstromu přijde šipka. Opět budeme předpokládat, že přišla zleva; pokud zprava, vyřešíme zrcadlově. Pokud přišla do $\oplus$ nebo $\odot$, ošetříme to stejně jako při přidání listu:

Pokud ale vrchol $x$ má znaménko $\ominus$, nastanou potíže: levý podstrom má teď hloubku o 2 vyšší než pravý, takže musíme rotovat. Proto se podíváme o patro níž, jaké je znaménko vrcholu $y$ pod šipkou, abychom věděli, jakou rotaci provést. Jedna možnost je tato ($y$ je $\ominus$):



Tehdy provedeme jednoduchou rotaci vpravo. Jak to dopadne s hloubkami jsme přikreslili do obrázku – pokud si hloubku podstromu $A$ označíme jako $h$, $B$ musí mít hloubku $h-1$, protože $y$ je $\ominus$, atd. Jen nesmíme zapomenout, že v $x$ jsme ještě $\ominus$ nepřepočítali (vede tam přeci šipka), takže ve skutečnosti je jeho levý podstrom o 2 hladiny hlubší než pravý (původní hloubky jsme na obrázku naznačili v [závorkách]). Po zrotování vyjdou u $x$ i $y$ znaménka $\odot$ a celková hloubka se nezmění, takže jsme hotovi.

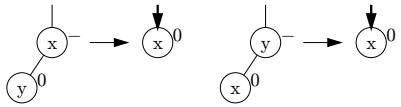
Další možnost je $y$ jako $\oplus$:



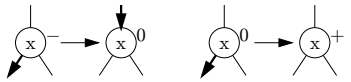
Tehdy se podíváme ještě o hladinu níž a provedeme dvojrotaci. (Nemůže se nám stát, že by $z$ neexistovalo, protože jinak by v $y$ nebylo $\oplus$.) Hloubky opět najdete na obrázku. Jelikož $z$ může mít libovolné znaménko, jsou hloubky podstromů $B$ a $C$ buďto $h$ nebo $h-1$, což značíme $h^-$. Podle toho pak vyjdou znaménka vrcholů $x$ a $y$ po rotaci. Každopádně vrchol $z$ vždy obdrží $\odot$ a celková hloubka se nemění, takže končíme.

Poslední možnost je, že by $y$ byl $\odot$, ale tu vyřešíme velmi snadno: všimneme si, že nemůže nastat. Kdykoliv totiž posíláme šipku nahoru, není pod ní $\odot$. (Kontrolní otázka: jak to, že $\oplus$ může nastat?)

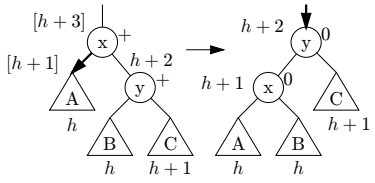
**Vyvažování po Deletu.** Vyvažování po Deletu je trochu obtížnější, ale také se dá popsat pár obrázky. Nejdříve opět rozebereme základní situace: odebíráme list (BÚNO levý) nebo vnitřní vrchol stupně 2 (tehdy ale musí být jeho jediný syn listem, jinak by to nebyl AVL strom):



Šipkou dolů značíme, že o patro výš posíláme informaci o tom, že se hloubka podstromu snížila o 1. Pokud šipku dostane vrchol typu $\ominus$ nebo $\odot$, vyřešíme to snadno:



Problematické jsou tentokráte ty případy, kdy šipku dostane $\oplus$. Tehdy se musíme podívat na znaménko *opačného* syna a podle toho rotovat. První možnost je, že opačný syn má $\oplus$:





```
    while (queue_head < queue_tail) {
        s = queue[queue_head++];
        j = D[s.tx][s.ty][s.mx][s.my] + 1;
        s = m_move (s);
        if (s.mx == s.tx && s.my == s.ty)          /* unhappy end */
            continue;
        if (s.tx == Ex && s.ty == Ey)              /* happy end */
            break;
        add (s, j, −1, 0);
        add (s, j, 1, 0);
        add (s, j, 0, −1);
        add (s, j, 0, 1);
    }
    if (queue_head < queue_tail) {                 /* happy end */
        i = D[s.tx][s.ty][s.mx][s.my] + 1;
        printf ("Cesta nalezena (v opacnem poradi):\n");
        while (i−−) {
            printf ("(%d,%d)\n", s.tx, s.ty);
            s = P[s.tx][s.ty][s.mx][s.my];
        }
    } else
        printf ("Cesta neexistuje\n");
    return 0;
}
```

**Úloha 16-3-3 – Genetikova evoluce – program**

```
#include <values.h>
#include <stdio.h>
#define M 30                                       /* Max vertices */

int used[M], minI[M], gens[M];                     /* in span?; nearest vertex in span; data */

int bits (int x, int y){
  int b=0, i, c=gens[x]^gens[y];
  for (i=0; i<30; i++) b+=c%2, c/=2;
  return b;
}                                                  /* bits */

int main (){
  int j, i, b, N, mutat=0;                         /* N=vertices; # mutations */
  int minI, minV=MAXINT;                           /* nearest vertex index; min edge val */
  used[0]=1;                                       /* root of tree */
  scanf ("%d", &N); for (i=0; i<N; i++) scanf ("%d", &gens[i]);   /* input */

  for (j=0; j<N−1; j++) {                          /* spanning tree has n-1 edges */
    for (i=0, minV=MAXINT; i<N; i++)               /* nearest edge */
      if (!used[i]) b=bits (i, minI[i]), minV= b<minV ?minI=i, b:minV;

    used[minI]=1; mutat+=bits (minI, minI[minI]);  /* add it to span */
    printf ("%d->%d;", gens[minI[minI]], gens[minI]);

    for (i=0; i<N; i++)                            /* update nearest vertices */
      if (!used[i]) if (bits (i, minI)<bits (i, minI[i])) minI[i]=minI;
  }                                                /* for j */

  printf ("\nCelkove %d mutaci\n", mutat);
}                                                  /* main */
```

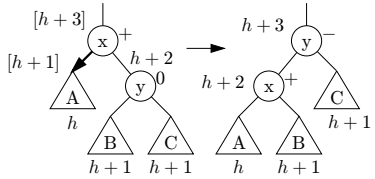
**Úloha 16-3-2 – Historikova past – program**

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

#define INFTY 999999
#define MAX 20

struct state {
        int tx, ty;                                     /* Theseus */
        int mx, my;                                     /* Minotaurus */
};

struct state queue[MAX*MAX*MAX*MAX];
int queue_head, queue_tail;

int N, M, K;
int Ex, Ey;                                             /* cíl */
int B[MAX][MAX];                                        /* bludiště */
int D[MAX][MAX][MAX][MAX];                              /* stavový prostor */
struct state P[MAX][MAX][MAX][MAX];                     /* předchůdci */

struct state m_move (struct state s)                    /* pohni s Minotaurem a vrat nový stav */
{
        int i;

        for (i=0; i<K; i++) {
                if (s.my > s.ty && B[s.mx][s.my-1] == '.')
                        s.my--;
                if (s.my < s.ty && B[s.mx][s.my+1] == '.')
                        s.my++;
                if (s.mx > s.tx && B[s.mx-1][s.my] == '.')
                        s.mx--;
                if (s.mx < s.tx && B[s.mx+1][s.my] == '.')
                        s.mx++;
        }
        return s;
}

void add (struct state s, int step, int dx, int dy)     /* zařaď "rozlévací" stav do fronty */
{
        struct state t = s;

        t.tx += dx;
        t.ty += dy;
        if (t.tx>=0 && t.tx<N && t.ty>=0 && t.ty<M && B[t.tx][t.ty] == '.' && D[t.tx][t.ty][t.mx][t.my] > step) {
                queue[queue_tail++] = t;
                D[t.tx][t.ty][t.mx][t.my] = step;
                P[t.tx][t.ty][t.mx][t.my] = s;
        }
}

int main (void)
{
        int i, j, x, y, c;
        struct state s;

        scanf ("%d %d %d", &N, &M, &K);
        scanf ("%d %d %d %d %d %d",                     /* souřadnice Thesea, Minotaura a východu */
          &s.tx, &s.ty, &s.mx, &s.my, &Ex, &Ey);
        for (i=0; i<N; i++) for (j=0; j<M; j++) {
                while (isspace (c=getchar ()));
                B[i][j] = c;
                for (x=0; x<N; x++) for (y=0; y<M; y++)
                        D[i][j][x][y] = INFTY;
        }
        queue[0] = s;                                   /* počáteční stav zařaď do fronty */
        queue_tail = 1;
        D[s.tx][s.ty][s.mx][s.my] = 0;
```



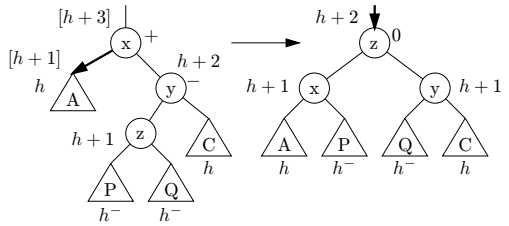
Tehdy provedeme rotaci vlevo, $x$ i $y$ získají nuly, ale celková hloubka stromu se sníží o hladinu, takže nezbývá, než poslat šipku o patro výš.

Pokud by $y$ byl $\odot$:




Opět rotace vlevo, ale tentokráte se zastavíme, protože celková hloubka se nezměnila.

Poslední, nejkomplikovanější možnost je, že by $y$ byl $\ominus$:




V tomto případě provedeme dvojrotaci ($z$ určitě existuje, jelikož $y$ je typu $\ominus$), vrcholy $x$ a $y$ obdrží znaménka v závislosti na původním znaménku vrcholu $z$ a celý strom se snížil, takže pokračujeme o patro výš.

**Happy end.** Jak při Insertu, tak při Deletu se nám podařilo strom upravit tak, aby byl opět AVL stromem, a trvalo nám to lineárně s hloubkou stromu (konáme konstantní práci na každé hladině), čili stejně jako trvá Insert a Delete samotný. Jenže o AVL stromech jsme již dokázali, že mají hloubku vždy logaritmickou, takže jak hledání, tak Insert a Delete zvládneme v logaritmickém čase (vzhledem k aktuálnímu počtu prvků ve stromu).

**Další typy stromů.** AVL stromy samozřejmě nejsou jediný způsob, jak zavést stromovou datovou strukturu s logaritmicky rychlými operacemi. Další jsou třeba:

- *Červeno-Černé stromy* – ty si místo znamének vrcholy barví, každý je buďto červený nebo černý a platí, že nikdy nejsou dva červené vrcholy pod sebou a že na každé cestě z kořene do listu je stejný počet černých vrcholů. Opět je hloubka stromu logaritmická, po Insertu a Deletu barvy opravujeme přebarvováním na cestě do kořene a rotováním, ovšem je potřeba rozebrat podstatně více případů než u AVL stromů. (Za to jsme ale odměněni tím, že nikdy nedělame více než 2 rotace.)
- *2-3-stromy* – v jednom vrcholu nemáme uloženu jednu hodnotu, nýbrž jednu nebo dvě (a synové jsou pak 2 nebo 3, odtud název). Hloubka opět logaritmická, vyvažování řešíme pomocí spojování a rozdělování vrcholů.
- *Splay stromy* – nezavádíme žádnou vyvažovací podmínku, nýbrž definujeme, že kdykoliv pracujeme s nějakým vrcholem, vždy si jej vyrotujeme do kořene a pokud to jde, preferujeme dvojrotace. Takové operaci se říká Splay a dají se pomocí ní definovat operace ostatní: Find hodnotu najde a poté na ni zavolá Splay. Insert si nechá vysplayovat předchůdce nové hodnoty a vloží nový vrchol mezi předchůdce a jeho pravého syna. Delete vysplayuje mazaný prvek, pak uvnitř pravého podstromu vysplayuje minimum, takže bude mít jen jednoho syna a můžeme jím tedy nahradit mazaný prvek v kořeni. Jednotlivé operace samozřejmě mohou trvat až lineárně dlouho, ale dá se o nich dokázat, že jejich *amortizovaná* složitost (co to je, najdete v kuchařce k předchozí sérii) je vždy $O(\log N)$. To u většiny použití stačí (datovou strukturu obvykle používáte uvnitř nějakého algoritmu a zajímá vás, jak dlouho běží všechny operace dohromady) a navíc je Splay stromy daleko snazší naprogramovat než nějaké vyvažované stromy. [Mimo to mají i jiné krásné vlastnosti: přizpůsobují svůj tvar četnostem hledání, takže často hledané prvky jsou pak blíž ke kořeni, snadno se dají rozdělovat a spojovat atd.]
- *Treapy* – randomizovaně vyvažované stromy: něco mezi stromem (tree) a haldou (heap). Každému prvku přiřadíme *váhu,* což je náhodné číslo z intervalu $\langle 0, 1\rangle$. Strom pak udržujeme uspořádaný stromově podle hodnot a haldově podle vah (všimněte si, že tím je jeho tvar určen jednoznačně). Insert a Delete opravují haldové uspořádání velmi jednoduše pomocí rotací. Časová složitost v průměrném případě je $O(\log N)$.
- *BB-$\alpha$ stromy* – zobecnění dokonalé vyváženosti jiným směrem: zvolíme si vhodné číslo $\alpha$ a vyžadujeme, aby se velikost podstromů každého vrcholu lišila maximálně $\alpha$-krát (prázdné podstromy nějak ošetříme, abychom nedělili nulou; dokonalá vyváženost odpovídá $\alpha = 1$ [až na zaokrouhlování]). V každém vrcholu si budeme pamatovat, kolik vrcholů obsahuje podstrom, jehož je kořenem, a po Insertu a Deletu přepočítáme tyto hodnoty na cestě zpět do kořene a zkontrolujeme, jestli je strom ještě stále $\alpha$-vyvážený. Pokud ne, najdeme nejvyšší místo, ve kterém se velikosti podstromů příliš liší, a vše od tohoto místa dolů znovu vytvoříme algoritmem na výrobu dokonale vyvážených stromů. Ten, pravda, běží v lineárním čase, ale čím větší podstrom přebudováváme, tím to děláme méně často, takže vyjde opět amortizovaně $O(\log N)$ na operaci.

**Cvičení.** Několik věcí, které se do kuchařky už nevešly, ale můžete si je zkusit vymyslet:

1. jak konstruovat dokonale vyvážené stromy
2. jak pomocí toho naprogramovat BB-$\alpha$ stromy
3. algoritmus, který k prvku ve stromu najde jeho následníka, což je prvek s nejbližší vyšší hodnotou (zde předpokládejte, že ke každému prvku máte uložený ukazatel na jeho otce)
4. jak vypsat celý strom tak, že začnete v minimu a budete postupně hledat následníky (i když nalezení následníka může trvat až $O(h)$, všimněte si, že projití celého stromu přes následníky bude lineární)
5. jak do vrcholů stromu ukládat různé pomocné informace, jako třeba počet vrcholů v podstromu každého vrcholu, a jak tyto informace při operacích se stromem udržovat (při Insertu, Deletu, rotaci)
6. že libovolný interval $\langle a, b\rangle$ lze rozložit na logaritmicky mnoho intervalů odpovídajících podstromům
7. a že zkombinováním předchozích dvou cvičení lze odpovídat i na otázky typu „kolik si strom pamatuje hodnot ze zadaného intervalu“ v logaritmickém čase…

**Několik poznámek na závěr.**

- Pokud záznamy můžeme jenom porovnávat, je binární vyhledávání nejlepší možné. Je totiž vždy popsáno nějakým binárním stromem a binární strom s $N$ vrcholy musí mít vždy hloubku alespoň $\lfloor \log N \rfloor$.
- Pokud bychom ale předpokládali, že se záznamy můžeme zacházet i jinak, dají se některé operace provádět i v konstantním čase (alespoň průměrně). K tomu se hodí například *hashování,* a to si popíšeme v některé z kuchařek v příštím ročníku KSP. Jeho nevýhodou ovšem je, že udržuje jenom množinu prvků, nikoliv uspořádání na ní, takže například nelze najít k zadanému prvku nejbližší vyšší.
- Pokud bychom připustili, že se mohou vyskytnout dva stejné záznamy, bude vše také fungovat, jen si musíme dát o něco větší pozor na to, co všechno při operacích se stromem může nastat.
- Jakpak přišly AVL stromy ke svému jménu? Inu, podle svých objevitelů pánů Adeľsona-Velského a Landise.
- Rekurenci $A_d = 1 + A_{d-1} + A_{d-2}$, $A_1 = 1$, $A_2 = 2$ pro velikosti minimálních AVL stromů je samozřejmě možné vyřešit i přesně. Žádné překvapení se nekoná, objeví se totiž stará známá Fibonacciho čísla: $A_n = F_{n+2} - 1$.

Dnešní menu vám servírovali
*Martin Mareš a Tomáš Valla*

## Vzorová řešení třetí série šestnáctého ročníku KSP

### 16-3-1 Fyzikova blecha

Tak jak to všechno dopadlo... U některých řešení by se blecha urazila nebo dokonce přímo unudila, než by dostala řešení. Tím myslím především algoritmy s exponenciální časovou složitostí, už daleko lepší byly kvadratické a nejlepší algoritmy byly se složitostí $O(N \log N)$. A takový si zde ukážeme.

Jak tento bleší problém vyřešíme? Začneme tím, že si plošinky utřídíme podle $y$-ové souřadnice. Předpokládejme, že u konců každé plošinky víme, na jakou jinou plošinku z tohoto konce blecha spadne. Budeme probírat plošinky podle stoupající $y$-ové souřadnice a u každé plošinky si budeme u obou konců počítat nejkratší cestu na podlahu. To provedeme tak, že zkusíme ze zpracovávaného konce plošinky spadnout na nižší plošinku (víme na kterou). Protože plošinka, na kterou dopadneme, je níž než zpracovávaná, už u ní známe nejkratší cestu z obou konců – vybereme si, zda jít doleva nebo doprava, aby byla cesta co nejkratší.

Celý tento postup zvládneme v čase $O(N)$, protože u každé plošinky uděláme jen konstantně mnoho operací (zjistíme na kterou plošinku spadneme, jak bude dlouhá cesta když po dopadu zahneme nalevo, jak bude dlouhá cesta když po dopadu zahneme napravo, vybereme minimum).

Jak tedy budeme u plošinky určovat, na jakou nižší blecha z jejího konce spadne? Použijeme k tomu *statický intervalový strom*. To je struktura, která si pro každý prvek s indexem 1 až $P$ pamatuje nějaké číslo, přičemž $P$ musí být pevné po celou dobu běhu programu. Intervalový strom umí dvě operace: *zjisti hodnotu prvku $i$* a *nastav hodnotu prvků* v intervalu $i \dots j$ na *co*, obě v čase $O(\log N)$.

Předpokládejme, že už takovou strukturu známe. Použijeme ji tímto způsobem: Jednotlivé prvky intervalového stromu budou použité $x$-ové souřadnice plošinek (je jich nanejvýš $2N$) a hodnota prvku $i$ ($i$-tá nejmenší $x$-ová souřadnice) je číslo nejvýše umístěné plošinky, která se na této $x$-ové souřadnici vyskytuje. Abychom mohli $x$-ové souřadnice očíslovat, musíme si je za začátku opět setřídit.

Na začátku dáme do intervalového stromu jen podlahu. Probereme si plošinky opět podle vzrůstající $y$-ové souřadnice a u každého konce (souřadnice $left_x, right_x$; předpokládejme, že po očíslování mají indexy $left_i, right_i$) se intervalového stromu zeptáme, jaká je hodnota prvku $left_i$ a $right_i$ (0 znamená podlaha, jiné číslo je pořadové číslo plošinky). Tím jsme zjistili, na kterou plošinku spadne blecha z levého a pravého konce plošinky. Poté zpracovávanou plošinku „přidáme“ do intervalového stromu, čili (pokud zpracováváme $i$-ou odspoda) do intervalového stromu zapíšeme hodnotu $i$ do prvků v intervalu $left_i \dots right_i$.

Pokud tedy zvládneme naimplementovat popsaný intervalový strom, máme řešení s časovou složitostí $O(N \log N)$, protože třídění nás stojí $O(N \log N)$ a dále zpracováváme $N$ plošinek a každou v čase $O(\log N)$. Paměťová složitost je jako obvykle $O(N)$.

*Statický intervalový strom* si můžeme představit jako dokonale vyvážený binární strom. Jednotlivé vrcholy odpovídají intervalům z rozmezí 1 až $P$ tak, že listy tohoto stromu jsou jednotlivé prvky (odpovídají intervalům $i \dots i$) a každý vnitřní vrchol odpovídá intervalu, který je roven sjednocení intervalů synů tohoto vrcholu. Čili vrchol celého stromu odpovídá intervalu $1 \dots P$, jeho levý syn intervalu $1 \dots \lfloor P/2 \rfloor$ a pravý syn intervalu $(\lfloor P/2 \rfloor + 1) \dots P$.

U každého vrcholu si budeme pamatovat jednak hodnotu $h_i$ a jednak informaci $p_i$, zda hodnota $h_i$ odpovídá všem prvkům na intervalu, který tento vrchol reprezentuje (u listů je to vždy *true*). Zjištění hodnoty nějakého prvku potom provedeme následovně: začneme ve vrcholu. Pokud je $p_v$ *true*, vrátíme hodnotu $h_v$. Jinak si vybereme levého nebo pravého syna (podle indexu prvku, jehož hodnotu zjišťujeme), a rekurzíme (určitě se zastavíme, listy mají $p_i$ na *true*).

Jak dopadne nastavení hodnoty prvků na intervalu $i \dots j$? Opět začneme ve vrcholu. Pokud interval, který zkoumaný vrchol $v$ pokrývá, je podinterval $i \dots j$, nastavíme $p_v$ na *true* a $h_v$ na nastavovanou hodnotu. Jinak se spustíme na toho syna (případně na oba), jehož interval má neprázdný průnik s intervalem $i \dots j$. (*Pozor:* Bylo-li $p_v$ *true*, je třeba nejprve rozdělit vrcholem reprezentovaný interval synům.)

Protože strom je dokonale vyvážený, má logaritmickou výšku. Obě operace závisí na výšce stromu (u nastavování intervalu je si to třeba rozmyslet - někdy se sice spustíme na pravého i levého syna, ale když to nastane, jednoho syna pokryjeme celého – nebudeme se z něj spouštět níže), mají tedy logaritmickou složitost.

Zbývá vyřešit jedinou drobnost – jak si rozumně vytvořit takový binární strom? Použijeme pro to pole o velikosti $2N$. Řekneme, že prvek s indexem 1 odpovídá intervalu $1 \dots P$. Dále řekneme, že synové prvku $i$ pokrývajícího interval $left_i \dots right_i$ jsou $2i$ levý a $2i + 1$ pravý. Levý syn bude pokrývat interval $left_i \dots \lfloor left_i + right_i)/2 \rfloor$, pravý syn bude pokrývat $\dots (\lfloor left_i + right_i)/2 \rfloor + 1) \dots right_i$. Tímto trikem můžeme chápat pole jako strom, přičemž jednotlivé intervaly, které vrcholy pokrývají, si počítáme až při průchodu tímto stromem/polem.

*Tomáš Vyskočil a Milan Straka*



```
}
on = j;                                              /* počet různých x-ových hodnot */

for (k=1; on<k; k*=2);
for (i=1; i<k; i++) p[i] = 0;                        /* p-čka vnitřních vrcholů 0 */
for (i=k; i<2*k; i++) { p[i] = 1; value[i] = 0; }    /* p-čka listů 1 */

add (1, 1, on, d[0][0], d[0][1], 0);
for (i=1; i<n; i++){                                 /* najdeme, na které plošinky dopadneme */
  lpred[i] = find (1, 0, on, d[i][0]);
  if (d[i][2] - d[lpred[i]][2] > v) lpred[i] = n;
  rpred[i] = find (1, 0, n, d[i][1]);
  if (d[i][2] - d[rpred[i]][2] > v) lpred[i] = n;
  add (1, 0, on, d[i][0], d[i][1], i);
}
lsum[n] = rsum[n] = INF;
lsum[0] = rsum[0] = 0;
for (i=1; i<n; i++){                                 /* hledáme nejkratší cestu */
  left = lpred[i], right = rpred[i];
  if (left && lsum[left] + abs (d[i][0] - d[left][0]) < rsum[left] + abs (d[i][0] - d[left][1]))
    lsum[i] = lsum[left] + abs (d[i][0] - d[left][0]);
  else if (left)
    lsum[i] = rsum[left] + abs (d[i][0] - d[left][1]);
  else lsum[i] = 0;

  right = lpred[i], right = rpred[i];
  if (right && lsum[right] + abs (d[i][0] - d[right][0]) > rsum[right] + abs (d[i][0] - d[right][1]))
    rsum[i] = lsum[right] + abs (d[i][0] - d[right][0]);
  else if (right)
    rsum[i] = rsum[right] + abs (d[i][0] - d[right][1]);
  else rsum[i] = 0;
}
for (max=0, maxi=0, i=0; i<n; i++){                  /* najdeme první plošinku */
  if (d[i][2] > max && yb >= d[i][2] && d[i][0] < xb && d[i][1] > xb){
    max = d[i][2];
    maxi = i;
  }
}
i = maxi, j = 0;                                     /* a vysledujeme zpáteční cestu */
while (!lpred[i] && !rpred[i] && i != n){
  if (xb - lsum[maxi] < rsum[maxi] - xb){
    i = lpred[maxi];
    result[j] = 'l';
  } else {
    i = rpred[maxi];
    result[j] = 'r';
  }
  j++;
}
if (i == n) printf ("Chudak blecha asi se nam urazi!\n");
else {
  for (i=0; i<j; i++){
    printf ("%c\n", result[j]);
  }
  if (xb - lsum[maxi] < rsum[maxi] - xb)
    printf ("%d\n", xb+lsum[maxi]);
  else
    printf ("%d\n", xb+rsum[maxi]);
}

return 0;
}
```

### Úloha 16-3-1 – Fyzikova blecha – program

```
#include <stdio.h>

#define MAX 1024
#define INF 0xffffff

int n, xb, yb, v, o[2*MAX], value[4*MAX];          /* value - hodnoty int. stromu */
int p[4*MAX],                                      /* p - vrchol int. stromu je pokryt */

int cmp (const void *a, const void *b)             /* porovnání dle y */
{ return ( (int *)a)[2] - ( (int *)b)[2]; }

int cmp2 (const void *a, const void *b)            /* a dle x */
{ return * (int *)a - * (int *)b; }

int find (int node, int lnode, int rnode, int x)   /* najdi nejbližší nižší plošinku */
{
  int m = (lnode + rnode) / 2;

  if (p[node])                                     /* máme koncový vrchol */
    return value[node];
  if (x <= o[m])                                   /* jinak jdeme vlevo nebo vpravo */
    return find (2 * node, lnode, m, x);
  return find (2 * node + 1, m+1, rnode, x);
}

void add (int node, int lnode, int rnode, int from, int to, int index)   /* přidej plošinku do int. stromu */
{
  int m;

  if (o[lnode] >= from && o[rnode] <= to){
    p[node] = 1;
    value[node] = index;
  } else {
    if (p[node] && lnode < rnode){                 /* rozdělíme interval na dva menší */
      p[2 * node] = p[2 * node + 1] = 1, p[node] = 0;
      value[2 * node] = value[2 * node + 1] = value[node];
    }
    m = (lnode + rnode) / 2;                       /* a pokryjeme zbytek */
    if (from <= o[m]) add (2 * node, lnode, m, from, to, index);
    if (o[m] < to) add (2 * node, m + 1, rnode, from, to, index);
  }
}

int main (void)
{
  int i, on, j, k, d[MAX][3], lpred[MAX], rpred[MAX], lsum[MAX], rsum[MAX];
  int left, right, maxi, max, result[MAX];

  bzero (o, MAX*sizeof (int));

  scanf ("%d", &n);                                /* načítáme */
  scanf ("%d %d %d", &xb, &yb, &v);
  for (i=0; i<n; i++){
    scanf ("%d %d %d", &d[i][0], &d[i][2], &d[i][1]);
    d[i][1] += d[i][0];
  }
  d[n][0] = -INF, d[n][1] = INF, d[n++][2] = 0;

  qsort (d, n, sizeof (int)*3, cmp);               /* třídíme podle výšky */
  for (i=0; i<2*n; i++) o[2*i] = d[i][0], o[2*i+1] = d[i][1];

  qsort (o, 2*n, sizeof (int), cmp2);              /* třídíme si x-ovou osu */
  i=0, j=0;
  while (i < 2*n){
    while (i < 2*n && o[i] == o[i+1])
      i++;
    o[j++] = o[i++];
```



### 16-3-2 Historikova past

Nejprve se zamysleme nad tím, jak bychom úlohu řešili, kdyby v bludišti nepřekážel Minotauros. V tomto jednoduchém případě potřebujeme pouze najít nejkratší cestu ze startu k cíli.

Využijeme při tom algoritmus *vlny*. Představme si, že v prvním kroku vylijeme vodu na startovní políčko, což si na něm poznačíme například číslem 1. V druhém kroku nám voda přeteče do políček sousedících se startovním (samozřejmě těch, kde není zeď) a to si na nich poznačíme číslem 2. A tak dále, obecně v $i$-tém kroku označíme číslem $i$ všechny dosud nezaplavené sousedy políček s číslem $i-1$. Zjevně číslo přiřazené políčku udává délku nejkratší cesty do něj. Samotnou cestu vypíšeme zpětným průchodem od cíle tak, že z políčka s číslem $j$ popojdeme do libovolného souseda s číslem $j-1$, případně si můžeme pamatovat matici předchůdců, odkud do políčka voda natekla.

Vlastně se probíráme grafem, kde vrcholy (stavy bloudění) jsou všechny možné přípustné polohy Thesea, hrana mezi dvěma vrcholy vede pokud pozice v bludišti sousedí a hledáme v něm nejkratší cestu ze startu do cíle. Jak podobný přístup použít i za přítomnosti žravého Minotaura?

Stav bloudění je tentokrát určen jak polohou Thesea, tak polohou Minotaura. Všechny přípustné stavy jsou tedy dvojice (poloha Thesea, poloha Minotaura), kde ani jeden zrovna nestojí ve zdi a Minotaurus nežere Thesea. Jakmile se Theseus pohne, umíme jednoznačně určit v čase $O(k)$, jak na to zareaguje Minotaurus. Opět tedy můžeme hledat nejkratší cestu v grafu, jehož vrcholy budou všechny stavy bloudění a hrana povede z $(T, M)$ do $(T', M')$, pokud $T'$ je sousedem $T$ a Minotaurus na přesun Thesea do polohy $T'$ zareaguje posunem z $M$ na $M'$. Na tento graf (stavový prostor), kde cílové stavy jsou takové, ve kterých Theseus stojí v cíli, stačí pouze vypustit algoritmus vlny. (Graf si samozřejmě nemusíme pamatovat žádným z populárních způsobů uchovávání grafu v paměti, přecházet mezi vrcholy umíme jednoduše i bez konstrukce hran.)

Zbývá si pouze rozmyslet, jak vlnu efektivně naprogramovat. Zavedeme si frontu na vrcholy, ze který se bude rozlévat voda. Vždy vyzvedneme prvek ze začátku fronty, zaplavíme jeho dosud nezaplavené sousedy a zařadíme je na konec fronty. Tím si zaručíme, že na každý stav (kterých je nyní nejvýše $(NM)^2$) při vlně sáhneme jen konstantněkrát. Přechod mezi dvěma stavy umíme vypočítat v čase $O(k)$, zpětný výpis zvládneme v čase lineárním k počtu stavů, celkový čas výpočtu tudíž bude $O(N^2M^2k)$. Potřebujeme si zapamatovat matici $N \times M$ s bludištěm, stavový prostor velikosti $(NM)^2$) reprezentovaný čtyřrozměrným polem, kam si ukládáme čas zaplavení, stejně mnoho předchůdců pro výpis cesty a opět stejně velkou frontu. Celková spotřebovaná paměť tedy bude $O(N^2M^2)$.

*Tomáš Valla*

### 16-3-3 Genetikova evoluce

Většina z vás správně uhodla, že hledaná evoluce je vlastně minimální kostrou grafu, jehož vrcholy jsou jednotlivé druhy, hrany mezi nimi možné evoluční skoky a ohodnocení hran (vzdálenost mezi vrcholy) odpovídá počtu mutací mezi jednotlivými druhy.

Stačilo pak opsat kuchařku a řešení mžouralo na svět. Žel, jak při rodinné večeři u Blátošlapů vyšlo najevo, nikoliv optimální. Hned poté, co se Blátošlapovi podařilo umlčet nejmladšího Blátomarcka, (jsa genetikou dosud neinfikován, neustále cosi drmolil o povstávání z bláta), musel zažehnávat hádku s pubertálním vzdorem Blátoťapky - jeho jedinou zábavou toho roku bylo neustále rozvracet genetiků svatý svět (jde přeci o vývoj toho, kdo text čte, nikoliv jen o vývoj textu samotného...). Nejstarší Blátotlačka byla jediná, s kým byla ten den rozumná řeč - a jaká ! Ihned si všimla dvou detailů.

Časová složitost v kuchařce je tak veliká, hlavně kvůli setřídění všech hran a nenápadná zmínka o maximálním počtu sledovaných znaků, ji přivedla na myšlenku třídících algoritmů, které za určitých podmínek pracují rychleji (RadixSort, CountedSort, atd). Při troše péče se pak nechá časová složitost zmenšit až na kvadratickou vzhledem k počtu vrcholů.

Druhý podstatný detail - nepracujeme s obecným grafem, ale jen se speciální podtřídou grafů, kde je každý vrchol spojen s každým, čili počet hran je vždy $O(n^2)$. Díky tomu si při použití DFU můžeme dovolit investovat do slití dvou tříd čas $O(n)$ (je to strom, slévat budu n-1 krát), tak abychom byli schopni zjistit reprezentanta třídy vždy v $O(1)$ - nepotřebujeme pak žádné zrychlovací finty, které zachrání amortizovaný čas. Tím se můžeme vyhnout nabobtnalým zdrojákům, ve kterých se vrší chyba na chybě.

A jaké bude naše řešení: víme, že hran je $O(n^2)$ , takže je nepravděpodobné, že se někdy dostaneme pod tuto složitost. Nenecháme se zmást narafičenou kuchařkou a vzpomeneme/vyhledáme Prim-Jarníkův algoritmus, který lze implementovat v $O(n^2)$.

Jaká je jeho myšlenka: Budeme kostru budovat postupně, podle vrcholů (nikoliv hran). V každém kroku bude budovaný podgraf $G$ souvislý a nebude obsahovat kružnice, tj. bude to strom. Z toho plyne, že po přidání posledního vrcholu budeme mít v rukou kostru původního grafu.

Indukční krok: Další vrchol vybírám z množiny sousedů budovaného podgrafu (tj. takové vrcholy, které sousedí v původním grafu s $G$, ale dosud do $G$ nebyly zařazeny). Z této množiny vyberu ten, který je „nejbližší“ ke $G$, tj. vyberu vrchol, který v daném kroku zvýší celkový součet mutací v $G$ minimálním možným způsobem.

V našem případě začneme budovat $G$ vždy od vrcholu, který reprezentuje prvotního předka v evoluci, čímž zaručíme správné pořadí otec → syn při generování evoluce.

To, že je nalezená kostra vskutku minimální by se dokazovalo sporem, a zvídavější povahy nechť hledají např. letošní MOP, kde je důkaz proveden se vší parádou.

Implementační detaily: budeme zařazovat celkově $n$ vrcholů, tedy pro každé zařazení smím spotřebovat maximálně $O(n)$ času. Zavedu si pole *min*, které pro každý nezařazený vrchol, uchovává nejbližší zařazený(tj.již v podgrafu $G$). Vyhledání minima v tomto poli jistě v $O(n)$ zvládnu. Aktualizaci pole po zařazení, znamená zkontrolovat, nemá-li nějaký nezařazený vrchol blíže k právě zařazenému - a to v $O(n)$ zvládnu také.

Zjištění velikosti mutace mezi dvěma druhy (*bits*), je ekvivalentní zjišťování počtu jedniček v XORu znaků jednotlivých druhů. V našem řešení lineární vzhledem k počtu znaků - těch je ovšem konstantně mnoho, i ono proto trvá konstantní čas (existuje však také algoritmus logaritmický vzhledem k počtu znaků). Zjišťování vzdálenosti mezi dvěma vrcholy se bude vícekrát opakovat, takže by bylo možné výpočet urychlit zavedením pomocného pole pro spočtené

výsledky. Odnesli bychom to však zhoršením paměťové složitosti ze současných $O(n)$ na $O(n^2)$.

*Pavel Šanda*

## 16-3-4 Ekonomova odměna

Většina z Vás sice neodsoudila Dlouhoprsta k smrti, ale zato ho odsoudila k chudobě. A ta je pro zloděje snad ještě horší než smrt. Avšak ne nadarmo se Dlouhoprst zove Dlouhoprstem – brzy přišel na to, jak piráty ožebračit.

Nejprve předpokládejme, že máme stříbrňáků nepočítaně, a zkusme zjistit, kolik nejméně jich potřebujeme k tomu, aby byl Dlouhoprstův návrh přijat. První důležité pozorování je to, že je-li $P \neq 1$, tak Dlouhoprst přežije. Pokud Dlouhoprst zaplatí dost, může na svou stranu získat všechny piráty kromě toho s číslem $P$.

Nejprve si ukažme, jak bude situace vypadat pro malá $P$:

| $P$ | | | | | | | |
|---|---|---|---|---|---|---|---|
| **0** | $S$ | | | | | | |
| **1** | $S$ | $-1$ | | | | | |
| **2** | 0 | 0 | $S$ | | | | |
| **3** | 1 | 1 | 0 | $S-2$ | | | |
| **4** | 0 | 2 | 1 | 0 | $S-3$ | nebo | |
| **4** | 2 | 0 | 1 | 0 | $S-3$ | | |
| **5** | 0 | 2 | 2 | 1 | 0 | $S-5$ | nebo |
| **5** | 2 | 0 | 2 | 1 | 0 | $S-5$ | nebo |
| **5** | 2 | 2 | 0 | 1 | 0 | $S-5$ | |

*Pozn.:* Poslední číslo na řádku je Dlouhoprstův zisk, $-1 = \dagger$.

Důležité je to, proč pirátům $1, 2$ stačí v případě $P = 5$ pouze 2 stříbrňáky. Vzhledem k tomu, že situace může v případě $P = 4$ dopadnout dvěma způsoby (buď dostane zaplaceno 1. nebo 2. pirát), nemají tito piráti **jistotu**, že ony 2 stříbrňáky získají. Proto se při $P = 5$ spokojí s dvěma stříbrňáky (ovšem jeden by jim nestačil - mají **naději** na dva).

Pokud budeme pokračovat v rozboru dále, zjistíme, že kromě počátačních případů ($P \leq 3$) je optimální řešení takovéhle: chceme, aby pro Dlouhoprsta hlasovalo $\lceil P/2 \rceil$ pirátů. Pirátovi $P$ nedáme nic, pirátovi $P-1$ dáme 1 stříbrňák a ze zbylých pirátů $1 \ldots P-2$ vybereme libovolných $\lceil P/2 \rceil - 1$ a dáme každému z nich 2 stříbrňáky (pro jednoduchost vybereme např. prvních $\lceil P/2 \rceil - 1$ pirátů). Celkem použijeme $2\,(\lceil P/2 \rceil - 1) + 1 = 2\lceil P/2 \rceil - 1$ stříbrňáků.

Pokud je tedy $S \geq 2\lceil P/2 \rceil - 1$, umíme úlohu vyřešit dokonce v konstantním čase a prostoru. Případ $P \leq 3$ vyřešíme tabulkou a pro ostatní hodnoty $P$ použijeme popsané rozdělení (pozor na to, že nesmíme vypsat počet stříbrňáků pro každého piráta – to by mělo složitost $O(P)$).

Situace začne být složitější, je-li $S$ menší než $2\lceil P/2 \rceil - 1$. Není totiž pravda, že pak nikdy neexistuje řešení. Např:

| S = 1 | P | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | **0** | 1 | | | | | | | | |
| | **1** | 1 | $-1$ | | | | | | | |
| | **2** | 0 | 0 | 1 | | | | | | |
| | **3** | 0 | 0 | 1 | $-1$ | | | | | |
| | **4** | 0 | 1 | 0 | 0 | 0 | | | | |
| | **5** | 0 | 1 | 0 | 0 | 0 | $-1$ | | | |
| | **6** | 0 | 1 | 0 | 0 | 0 | $-1$ | $-1$ | | |
| | **7** | 0 | 1 | 0 | 0 | 0 | $-1$ | $-1$ | $-1$ | |
| | **8** | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Ačkoli se piráti dělí o jediný stříbrňák, pokud je jich 8, Dlouhoprst může přežít. Využije toho, že pro něj zadarmo budou hlasovat piráti $6, 7, 8$, kteří jinak zemřou.

Řešení situace, kdy je $S$ malé, založíme na dynamickém programování. Budeme postupně řešit situaci pro rostoucí $P$ až do zadaného. U každého piráta si budeme pamatovat, jaký počet stříbrňáků má „jistý“ a zda má šanci získat víc než svou „jistotu“. Všimněte si, že dále popsaný postup funguje pro libovolná $S$, mohli bychom tak řešit (byť pomaleji) i případ dostatečně velkého $S$.

Řekněme, že známe řešení pro $P-1$ pirátů a zajímá nás řešení situace s $P$ piráty. Hledáme $\lceil P/2 \rceil$ pirátů, kterým zaplatíme nejméně. Pokud budeme chtít, aby pro Dlouhoprsta hlasovali všichni piráti, kteří mají jistých $R$ stříbrňáků, nabídneme jim $R+1$ stříbrňáků a upravíme jejich „jistotu“. Pokud budeme chtít, aby hlasovali jenom někteří piráti z těch s „jistotou“ $R$ stříbrňáků, nabídneme jim taktéž $R+1$ stříbrňáků. Nezměníme ovšem jejich „jistotu“, jenom si u nich poznamenáme, že mají šanci získat více než svou „jistotu“.

Nyní si všimneme důležité věci: žádný pirát (kromě toho, který právě navrhuje rozdělení) nemůže dostat více než dva stříbrňáky, neboli „jistota“ každého piráta je menší než dva. To platí proto, že aby měl nějaký pirát jisté dva stříbrňáky, musel by někdy Dlouhoprst nabídnout dva stříbrňáky všem pirátům, kteří měli jistý jeden. Uvažujme nyní situaci, kdy Dlouhoprst poprvé nabízí dva stříbrňáky všem pirátům, kteří mají jistý jeden. V tuto chvíli jsou ještě „jistoty“ všech pirátů menší než dva. Když už by ovšem Dlouhoprst uplácel piráty s jedním jistým stříbrňákem, piráty s nižší „jistotou“ by už museli být uplaceni (jsou totiž levnější). Tedy piráti s „jistotou“ menší než jedna jsou již uplaceni, všichni piráti s „jistotou“ rovnou jedné jsou uplaceni a žádní jiní piráti neexistují. Čili Dlouhoprst by uplácel **všechny** piráty! To je ovšem spor s tím, že Dlouhoprst chce uplatit jen $\lceil P/2 \rceil$ pirátů.

Pokud tedy zjišťujeme řešení situace s $P$ piráty ze situace s $P-1$ piráty, nejprve zjistíme, kterým pirátům bude Dlouhoprst platit. Vzhledem k tomu, že „jistoty“ všech pirátů jsou menší než dva, dokážeme to v konstantním čase (pokud si pamatujeme, kolik pirátů má „jistotu“ rovnu $-1$ (smrt), kolik jí má nulovou a kolik pirátů má jistý 1 stříbrňák). Poté můžeme jedním průchodem nad piráty $1 \ldots P$ upravit jejich „jistoty“ a šance na získání většího počtu stříbrňáků. To celé zvládneme v čase $O(P)$.

Celý algoritmus bude fungovat tak, že bude zjišťovat, jak dopadnou návrhy pro $1, 2, \ldots, P$ pirátů a poslední z těchto návrhů vypíše. Při výpisu všem pirátům, kteří nemají šanci na nic víc než jejich „jistotu“, zaplatíme. Ovšem pirátům, kteří mají naději na víc, platíme o jeden stříbrňák víc a nemusíme jim platit všem, jenom tolika, abychom uplatili celkem $\lceil P/2 \rceil$ pirátů.

Celkem $P$-krát opakujeme postup o složitosti $O(P)$ a poté výsledky v čase $O(P)$ vypíšeme, čili celková časová složitost popsaného algoritmu je $O(P^2)$. Paměťová složitost je $O(P)$. A ačkoliv popsaný algoritmus používáme jen v případě, kdy $S < 2\lceil P/2 \rceil - 1$, funguje korektně pro všechna $S$.

*Milan Straka*

## 16-3-5 Botanikova setba

V jednorozměrném případě je úloha poměrně jednoduchá:

Udržujeme si dvě proměnné $(a_x, b_x)$, které nám říkají odkud a kam sahá zkoumaný úsek. Pokud úrodnost ve zkoumaném úseku je větší než prozatímní největší, upravíme ji. Pokud je naopak úrodnost menší nebo rovna nule, nevyplatí se nám tento úsek použít, a je lepší na něj zapomenout (tj. $a_x = b_x + 1$). Pokud je součet prvků aspoň jedna, vždy se nám vyplatí $a_x$ ponechat a tyto prvky přidat k naší zkoumané oblasti. Časová složitost tohoto je $O(n)$.

Nyní převedeme dvojrozměrný případ na jednorozměrný:

Projdeme všechny horní řádky $a_y$. Nyní budeme procházet všechny spodní řádky $b_y$, a zároveň si udržovat součet všech prvků ve sloupečcích v poli $c$. Časová složitost této části je $O(n^2)$. Na pomocné pole $c$ je nyní možné aplikovat jednorozměrné řešení. Celková složitost tak bude $O(n^3)$.

Při implementaci je trochu potřeba dát pozor na matici plnou záporných čísel, ale v této implementaci jsme to zvládli bez speciálního kódu. (Takový případ jde řešit v $O(n^2)$, ale kdo by sázel rostlinky se záporným ziskem?)

Vhodným předotočením matice je možné složitost vylepšit pro nečtvercové matice na $O(m * n * \min(m, n))$ (s díky Martinu Dobrouckému).

*Pavel Machek*

## 16-3-6 Matematikova sonda

Kdyby náš matematik byl věděl, jaké nesnáze řešitelům KSP způsobí, asi by kosmický výzkum pověsil na hřebík a raději by pořádal přednášky o pravděpodobnosti nebo okopával na zahrádce integrály. Přišla nám totiž necelá dvě řešení. My si ukážeme celé třetí. Držte se, jedeme s kopce!

Pokud se matematik rozhodne používat pravděpodobnostní strategii, bude jeho (pravda, místy poněkud jednotvárný) život probíhat takto: Osud si nejdříve rozmyslí, kolik pokusů se sondami kupovanými na místním jarmarku bude neúspěšných (to si označíme $t-1$, čili $t$-tá kupovaná sonda by už uspěla). Následně matematik spustí svůj algoritmus a ten se v $i$-tém tahu s pravděpodobností $p_i$ rozhodne, že nastal čas sondu si za $K$ grošů vyrobit, nebo naopak (tedy s pravděpodobností $1 - p_i$) si ji zajít na jarmark za 1 groš koupit. V průměrném případě tedy za sondy zaplatí

$$A_t = \sum_{i=1}^{t} p_i \cdot (K + i - 1) + \left(1 - \sum_{i=1}^{t} p_i\right) \cdot t \qquad [\spadesuit]$$

$$= \sum_{i=1}^{t} p_i \cdot (K + i - 1) + \sum_{i>t} p_i \cdot t \qquad [\heartsuit]$$

(první suma odpovídá případům, kdy si matematik koupí sondu v $i$-tém kroku, čili si ji $(i-1)$-krát koupí za 1, pak zaplatí $K$ za výrobu svépomocí a pak už nic; druhá suma popisuje případ, kdy si ji vždy koupil). Optimální strategie (kterou by použil každý správný věštec) by stála $O_t = \min(t, K)$ (pokud je $t < K$, vyplatí se sondy kupovat, jinak si ji hned na začátku za $K$ vyrobit).

Všimněte si, že každá volba čísel $p_1, p_2, p_3, \ldots$, kde $\forall i\ p_i \in \langle 0, 1 \rangle$ a $\sum_i p_i = 1$, odpovídá nějaké pravděpodobnostní strategii, a naopak každá p. s. se dá nějakou takovou posloupností popsat. My tedy chceme najít taková $p_i$, aby relativní cena naší strategie vůči optimální (ať už je na nás Osud sebevíc zlý) – tedy $C = \max_t A_t/O_t$ – byla nejmenší možná.

Nejdříve dokážeme, že pro $i > K$ se vyplatí $p_i$ ponechat nulové: Pokud $t > t' \geq K$, je také $A_t \geq A_{t'}$, protože při přechodu od $A_t$ k $A_{t'}$ přejdou v $\heartsuit$ některá $p_i$ (pro $t \geq i > t'$) z první sumy do druhé, takže se místo $K + i - 1$ budou násobit $t'$, což je číslo menší. Pokud bychom ale libovolné $p_{i>K}$ vynulovali a jeho původní hodnotu přičetli k $p_K$ (tím zůstane $\sum_i p_i$ zachována), všechna $A_{j<K}$ zůstanou nezměněna, $A_{j \geq i}$ se sníží a $A_j$ pro $K \leq j < i$ se sice zvýší, ale jelikož $A_j/O_j = A_j/K \leq A_i/K = A_i/O_i$, určitě si tím nepohoršíme. Takto můžeme postupně vynulovat všechna $p_{i>K}$ a získat stejně dobrou nebo dokonce lepší strategii.

Situace se tím nekonečněkrát zjednoduší – místo nekonečně mnoha $p_i$ teď stačí najít pouze $p_1, \ldots, p_K$ a také není třeba uvažovat $t > K$, jelikož pro taková $t$ se jak náš, tak optimální algoritmus chovají stejně jako pro $t = K$.

Pokud má naše strategie mít relativní cenu $C$, musí tedy pro všechna $t \leq K$ být $A_t \leq C \cdot t$, což si rozepíšeme pomocí $\spadesuit$ a místo $C$ použijeme $c = C - 1$, aby se nám lépe počítalo:

$$A_t - t = \sum_{i=1}^{t} p_i \cdot (K + i - 1) - t \cdot \sum_{i=1}^{t} p_i \leq c \cdot t.$$

Bez újmy na obecnosti budeme předpokládat, že pro všechna $t$ dokonce nastává rovnost (vyzkoušejte si, že kdyby nenastávala, mohli bychom $p_i$ trochu pozměnit a buďto zvětšit počet rovností, nebo dokonce snížit $c$). Pokud si pak do této rovnosti dosadíte $t = 1$, vyjde, že:

$$p_1 = c/(K - 1).$$

A pokud místo toho rovnost pro $t = j + 1$ odečtete od rovnosti pro $t = j$, dozvíte se, že:

$$p_{j+1} \cdot (K + j) - (j + 1) \cdot p_{j+1} - \sum_{i=1}^{j} p_i = c,$$

čili

$$p_{j+1} \cdot (K - 1) = c + \sum_{i=1}^{j} p_i.$$

Tuto rovnost opět odečteme od sebe samé pro $j$-čka lišící se o jedničku a konečně dostaneme rozumný rekurentní vztah:

$$p_{j+1} = p_j \cdot \frac{K}{K - 1},$$

z nějž plyne, že:

$$p_j = p_1 \cdot \left(\frac{K}{K - 1}\right)^{j-1}.$$

Tím jsou pro každé $c$ všechna $p_i$ jednoznačně určena, ovšem obvykle nebudou dávat korektní rozdělení pravděpodobností. Musíme proto $c$ nastavit tak, aby platilo:

$$1 = \sum_{i=1}^{K} p_i = \frac{c}{K - 1} \cdot \sum_{i=1}^{K} \left(\frac{K}{K - 1}\right)^{i-1} =$$

$$= \frac{c}{K - 1} \cdot \frac{\left(\frac{K}{K-1}\right)^K - 1}{\frac{K}{K-1} - 1} = c \cdot \left(\left(1 + \frac{1}{K - 1}\right)^K - 1\right).$$

Jenže jak každý správný matematik ví už od kolébky, výraz $(1 + 1/K)^K$ se pro rostoucí $K$ blíží k Eulerovu číslu $e \approx 2.718281828459045$ a stejně tak $(1 + 1/(K-1))^K$, kterýžto výraz je dokonce i pro malá $K$ zaručeně $\geq e$. Platí tedy, že $c \leq 1/(e-1)$, a tudíž $C = 1 + c \leq e/(e-1) \approx 1.5819767068693264$. Takže jsme našli pravděpodobnostní strategii pro pořizování kosmických sond, podstatně lepší, než je ta deterministická, a dokonce jsme dokázali, že taková strategie je nejlepší možná. (Tedy alespoň za předpokladu, že Osud a bohyně Fortuna stanovují své úradky „offline“, čili nezávisle na našich experimentech. Zkuste si rozmyslet, že nad online Osudem již neuhrajeme více než deterministický algoritmus.)

Tak končí naše malá sonda do života matematiků. A promiňte, už musím letět, právě mi přistává létající talíř výhodně pořízený od místního hokynáře...

*Martin Mareš*