PRASK
IV. ročník, 2017/18
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

Vzorové riešenia 2. kola zimnej časti

1. Pokračovanie príbehu s pavúkom Jonášom

vzorák napísal(a) Žaba
(max. 15 b za riešenie)

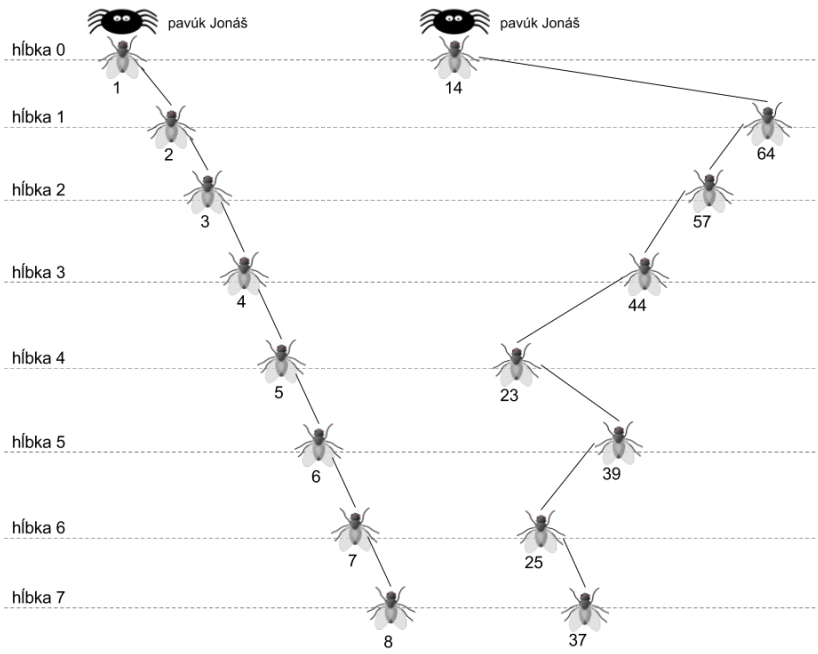
Tak ako v predchádzajúcej úlohe, aj tu sme pracovali s binárnymi vyhľadávacími stromami. Tentokrát sme si však mali všímať trochu inú vlastnosť – hĺbku. Poďme si najskôr ukázať ako sa riešili zadané úlohy a na konci vzorového riešenia si povieme, čo má táto úloha spoločné s reálnym použitím takýchto stromov.

Podúloha a.

V tejto podúlohe bolo treba čo najrýchlejšie vytvoriť pavučinu s hĺbkou 7. Najlepšie bolo otvoriť si priloženú stránku a vytvárať nejaké siete. Nie je ťažké prísť na to, že ak chceme použiť čo najmenej mušiek, musí každá nová muška zväčšiť hĺbku siete. Takáto sieť potom bude mať 8 mušiek (netreba zabudnúť na hĺbku 0), čo je naozaj najmenší možný počet.

Ostáva už len nájsť vhodných 8 čísiel – váhy mušiek. Jedno možné riešenie je pridávať vždy mušku, ktorá je ťažšia ako všetky ostatné mušky v sieti. Pri pridávaní takejto mušky totiž Jonáš ide po sieti vždy doprava, pri každom kroku o úroveň hlbšie. Zároveň navštívi všetky mušky a novú pridá napravo od nich, samú do novej hĺbky.

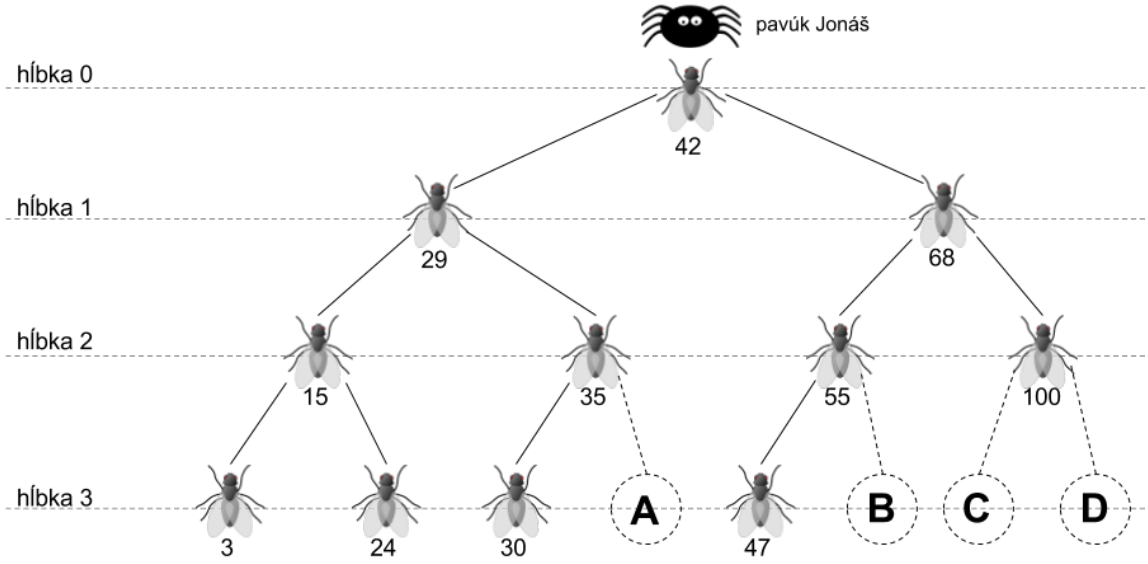
Možná postupnosť mušiek, ktoré má Jonáš do siete pridávať je napríklad: 1, 2, 3, 4, 5, 6, 7 a 8. Samozrejme funguje aj pridávanie mušky, ktorá je ľahšia od ostatných a dokonca sa dá nájsť aj veľa iných riešení. Na obrázku nižšie si môžete pozrieť aj sieť, ktorú by Jonáš dostal, keby mušky pridával v poradí 14, 64, 57, 44, 23, 39, 25 a 37.



Podúloha b.

V tejto podúlohe mal Jonáš presne opačnú úlohu – pridať do siete čo najviac mušiek tak, aby výsledná sieť nemala hĺbku väčšiu ako 3. Na začiatok sme si mohli zobrať obrázok zo zadania. Toto je totiž sieť hĺbky 3, ktorá obsahuje 11 mušiek. A dokonca poznáme aj postupnosť mušiek, ktorých pridanie túto sieť vytvorí.

Je to však najviac mušiek, ktoré v takejto sieti vedia byť? Na obrázku predsa ľahko môžeme vidieť zopár prázdnych miest, do ktorých by sa ešte nejaké mušky zmestili. Presnejšie, v hĺbke 3 by mohli byť ešte 4 ďalšie mušky. Ostáva zistiť, akú váhu by mali mať.



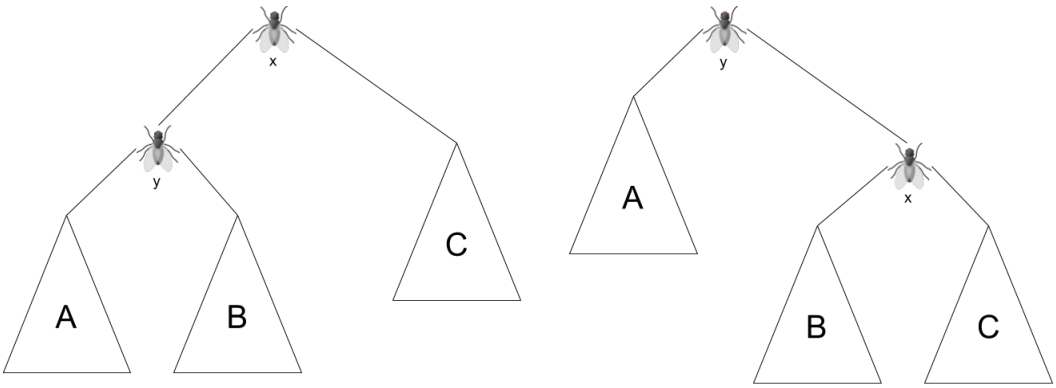
Začnime prvým voľným miestom, ktoré je na obrázku označené A. Čo vieme povedať o váhe mušky, ktorá by mohla byť na tomto mieste? Keďže je naľavo od najvyššej mušky, musí byť ľahšia ako 42. Je však napravo od mušky 29, takže od nej musí byť ťažšia. A musí byť ťažšia aj od mušky 35. Ľubovoľné číslo v intervale $(35, 42)$ by preto malo vyhovovať. A ľahko overíme, že tomu tak naozaj je.

Pre pozíciu B dostaneme interval $(55, 68)$, pre pozíciu C interval $(68, 100)$ a pre pozíciu D interval $(100, \infty)$. Ľubovoľné čísla z týchto intervalov budú vyhovovať. Treba ich už len doplniť do postupnosti, teda povedať, kedy má tieto mušky Jonáš pridať.

Môžeme ich dať na začiatok tejto postupnosti? Asi nie. Ak je totiž sieť prázdna, tak pridanie mušky 105 ju zaradí na vrch siete, odkiaľ sa už potom neposunie. Vidíme teda, že ak chce byť muška x zavesená pod muškou y, tak muška y musí byť do siete pridaná skôr ako muška x. Naše nové čísla teda musíme zaradiť za čísla, pod ktorými visia. Pri dodržaní tohto pravidla je však jedno kam ich dáme. Jedna možná postupnosť teda je: 42, 29, 15, 68, 3, 35, A=**39**, 55, 47, 24, 30, B=**60**, 100, D=**105**, C=**84**.

Podúloha c.

V podúlohe c. sme mali načrtnúť postup, ktorý by nám umožnil presunúť mušku, ktorá je zavesená priamo pod najvyššou muškou na vrch siete. Cestou k správnemu riešeniu je nakresliť si dostatočne všeobecne situáciu, ktorá nastáva. Poďme presúvať mušku naľavo na vrch siete. Čo si potrebujeme zaznačiť? Určite mušku x, ktorá je na vrchu, mušku y, ktorá je naľavo od x a je teda ľahšia. Všetky zvyšné mušky potom vieme rozdeliť do troch skupín – ľahšie ako y (naľavo od mušky y), ťažšie ako y ale ľahšie ako x (napravo od mušky y) a ťažšie ako x (napravo od mušky x). Mušky v jednej takejto skupine sú vzhľadom na mušky x a y úplne rovnaké. Preto aj vo výslednom strome musia všetky skončiť na rovnakej strane od x a y. Nebudeme preto kresliť všetky tieto mušky zvlášť, ale jednu skupinu si predstavíme ako samostatnú sieť, ktorá bude zavesená pod x alebo y. Tieto siete si označíme A, B a C.



Vieme, že vo výslednom obrázku musí byť muška y navrchu. Tak si ju tam nakreslime. Naľavo od nej budú mušky ľahšie. Všetky ľahšie mušky sú však v sieti A. Naľavo od y preto bude zavesená sieť A. Všetky zvyšné mušky (vrátane x) sú ťažšie ako y a preto musia byť od y napravo.

Navyše vieme, že mušky v sieti C sú ťažšie ako x a teda budú napravo od tejto mušky. Ostáva sieť B. O tej vieme, že sú v nej mušky ťažšie ako y ale ľahšie ako x. Od mušky y teda musí byť B napravo a od mušky x naľavo. A práve tieto podmienky spĺňa v našom novom obrázku miesto naľavo od x.

Uvedommte si naviac, že muška x sa presunula hneď napravo od y. Takže ak by sme chceli vyriešiť opačnú úlohu, kde máme mušku napravo presunúť na vrch, tak stačí spraviť uvedený postup spätne.

Podúloha d.

Jonáš chce na vrch pavučiny presunúť tú mušku v sieti, pri ktorej aktuálne stojí. V predchádzajúcej úlohe sme si ukázali ako dostať mušku susednú s vrchnou mušku na samý vrch. Muška pri ktorej stojí Jonáš však nemusí susediť s vrchnou muškou.

Predstavme si, že k prvému obrázku z príkladu vyššie pridáme mušku z, pod ktorou je na ľavej strane muška x. Ak spravíme uvedené zmeny, muška z ostane na svojom mieste, akurát na jej ľavej strane bude zrazu muška y. A to isté, ak by boli tieto mušky napravo od z. Pri postupe z podúlohy c. totiž nemeníme nič, čo je vyššie ako mušky, ktoré vymieňame.

Ak si teda zoberieme mušku y, tak tá sa iba posunula na vyššiu úroveň. No a našou úlohou je dostať ju na najvyššiu úroveň – hĺbku 0. Mohli by sme teda tento postup opakovať. Jonáš zakaždým zoberie mušku, ktorú chce dostať na samý vrchol a presunie ju na pozíciu o jedno vyššie pomocou postupu z podúlohy c.. Muška bude postupne stúpať až sa objaví na samom vrchu pavučiny.

Podúloha e.

V podúlohe e. máme dva obrázky sietí. Našou úlohou je navrhnúť postup, vďaka ktorému upravíme prvú sieť na tú druhú. Pritom predpokladáme, že obe siete obsahujú rovnaké mušky.

Vďaka podúlohe d. dokážeme na vrch siete dostať ľubovoľnú mušku. Určite teda vieme spraviť aspoň to, aby sa najvrchnejšie mušky v našich sieťach rovnali. Jednoducho zoberieme sieť, ktorú máme zmeniť, nájdeme v nej mušku, ktorá má byť navrchu a postupne ju presunieme na vrch, tak ako v predchádzajúcej podúlohe.

To však nemusí stačiť. Sieť, ktorú sme vytvorili, sa nemusí rovnať tej, ktorú máme predpísanú ako vzor. Určite sa rovnajú iba vo vrchnej muške. Uvedomme si však nasledovnú vec. V oboch sieťach sú naľavo tie isté mušky. A to isté platí aj o pravej strane. Na vrchu oboch sietí totiž sedí muška x. A všetky ľahšie mušky musia byť naľavo v oboch prípadoch, inak by predsa neplatili zadané podmienky.

Pozrime sa teda na mušky naľavo od x. Tie tvoria vlastnú, menšiu sieť, ktorá je potom pripojená na mušku x. Takže v podstate máme ten istý problém, akurát trochu menší. Máme dve siete (tá naľavo od x v prvom a tá naľavo od x v druhom obrázku) a máme ich preusporiadať tak, aby sa rovnali. Naviac si uvedomme, že to, čo robíme v podúlohách c. a d. nemení nič, čo je vyššie ako muška, ktorú presúvame. Takže túto menšiu sieť vieme meniť bez toho, aby sme ovplyvnili mušku x alebo jej pravú časť.

Tak ako predtým teda môžeme zopakovať náš postup. Pozrieme sa, ktorá muška je naľavo od mušky x vo vzorovej sieti a túto mušku presunieme na vrch ľavej siete aj v druhom obrázku. Takto docielime, že sa nám rovnajú už dve mušky. Rovnakým spôsobom môžeme však upraviť mušku, ktorá je napravo od mušky x. Takže naše siete sú rovnaké na hĺbke 0 a 1. A teraz môžeme jednoducho pokračovať s muškami v hĺbke 2, 3...

Pár slov na záver

Binárne vyhľadávacie stromy sú pre programátora silný nástroj. Aj tu sa však treba mať na pozore. Ak strom vyzerá tak ako v podúlohe a., všetky operácie na ňom trvajú dlho. Ak sa totiž vrátite k predchádzajúcemu riešeniu, väčšinou sa Jonáš pohyboval toľko krokov, ako hlboký bol zadaný strom. Preto by sa nám páčilo, aby naše stromy vyzerali viac ako stromy v podúlohe b..

To však nie je vždy také jednoduché. Pomôcť si však môžeme takzvanými rotáciami, ktoré ste vymysleli v podúlohe c.. Videli sme, že vďaka nim vieme preskupiť niektoré časti stromu a tak ich lepšie vyvážiť, aby sa nám nestalo, že jedna časť je oveľa hlbšia ako druhá.

To ako presne sa tieto rotácie využívajú je už na ďalšiu úlohu :) Ako však vidíte v podúlohe e., v konečnom dôsledku vďaka nim vieme dostať akýkoľvek tvar, aký si zažiadame.

2. Príbeh o Dášenke a minci

vzorák napísal(a) Baška
(max. 15 b za riešenie)

Pripomeňme si, že ráno dostala Dášenka dve rovnaké čísla p a d, ktoré sú väčšie ako nula. Následne niekoľkokrát hodila mincou a podľa toho, čo jej padlo spravila jednu z nasledovných operácií.

- **Z** (znak): pripočítame d ku p, takže vznikne dvojica čísel $p + d$ a d
- **H** (hlava): pripočítame p ku d, takže vznikne dvojica čísel p a $d + p$

Na konci dňa mala Dáša dvojicu čísel P a D. Našou úlohou bolo z týchto čísel zistiť, s akou hodnotou p začínala a tiež prísť na to, čo jej padalo na minci.

Na začiatok urobme niekoľko užitočných pozorovaní:

- Dáša **nikdy** nemohla mať **záporné číslo alebo nulu**.

  Obe čísla p, d sú ráno väčšie ako nula a operácie Z a H k nim hodnoty pričítavajú. No a súčet dvoch kladných čísel môže byť iba kladný.

- Okrem rána (pred prvým hodom) sa Dášine čísla **nemôžu rovnať**.

  Predstavme si, že v nejakom momente má Dášenka čísla P' a D'. Po ďalšom hode bude mať čísla $P' + D'$ a D' (resp. P' a $D' + P'$). Ak by sa ale tieto dve čísla rovnali, tak platí $P' + D' = D'$ (resp. $P' = D' + P'$), čo znamená, že $P' = 0$ (resp. $D' = 0$). Podľa predchádzajúceho pozorovania sa však niečo také nemôže stať.

Podúloha a)

Dášenka doniesla Bobovi čísla $P = 15$ a $D = 35$.

Ako zistíme začiatočné čísla a postupnosť hodov? Najlepšie by bolo odsimulovať Dášenkino hádzanie, nevieme však, kde začať. Mohli by sme teda vyskúsať všetky možné začiatočné hodnoty p. Tých je iba 15. Uvedomme si totiž, že platí $p \leq P = 15$ a $p \leq D = 35$. Nepoznáme však ani postupnosť hodov, museli by sme vyskúšať všetky možnosti toho, čo mohla Dáša hodiť.

Skúšať všetky možnosti hodov je naozaj náročné a počet možností stúpa veľmi rýchlo s rastúcim počtom hodov. V prvom kroku mohla hodiť buď H alebo Z. To sú dve možnosti. V druhom kroku síce platí to isté, ale už musíme rátať s tým, čo padlo v prvom kroku, takže máme štyri možnosti: ZZ, ZH, HZ, HH. Pri troch hodoch je tých možností opäť dvakrát viac, lebo pre každú z postupností pre dva hody máme dve možnosti čo mohlo padnúť ďalšie: $ZZ\boldsymbol{Z}, ZZ\boldsymbol{H}, ZH\boldsymbol{Z}, ZH\boldsymbol{H}, HZ\boldsymbol{Z}, HZ\boldsymbol{H}, HH\boldsymbol{Z}, HH\boldsymbol{H}$. Každým hodom sa počet možností, ktoré musíme vyskúšať, zdvojnásobí. Už pre 10 hodov musíme vyskúšať viac ako 1 000 možností, preto to nevyzerá ako správny postup.

Skúsme sa teda na to pozrieť zozadu. Pri tomto postupe aspoň poznáme výsledné hodnoty P a D. Čísla, ktoré mala Dášenka pred posledným hodom si označme P' a D', ich hodnotu zatiaľ nepoznáme. Následne padla minca, čísla sa zmenili a dostali sme čísla P a D. Posledný hod mohol byť Z aj H, skúsme preto vyskúšať obe možnosti.

Posledný hod: znak

Ak čísla pred posledným hodom boli P' a D' a hodili sme Z, tak nové čísla musia byť $P' + D'$ a D'. Zároveň vieme, že nové čísla sú $P = 15$ a $D = 35$ (keďže to bol posledný hod), takže musí platiť:

$$P' + D' = P = 15$$
$$D' = D = 35$$

Z týchto rovníc vyplýva, že $D' = 35$ a $P' + 35 = 15$, teda $P' = -20$.

To sa ale vylučuje s naším pozorovaním – nemôžeme mať záporné číslo. Posledný hod teda nemohol byť znak.

Posledný hod: hlava

Budeme postupovať úplne rovnako ako predtým.

Čísla pred posledným hodom boli P' a D' a hodili sme H, takže nové čísla musia byť P' a $D' + P'$. Zároveň vieme, že nové čísla sú $P = 15$ a $D = 35$ (keďže to bol posledný hod), takže musí platiť:

$$P' = P = 15$$
$$D' + P' = D = 35$$

Z toho jasne vyplýva, že $P' = 15$ a $D' + 15 = 35$, teda $D' = 20$.

Obe čísla su kladné, takže posledný hod mohol byť H. Keďže sme však hod znak vylúčili, **posledný hod musela byť hlava**.

Zvyšné hody

Teraz však máme úplne rovnakú úlohu, ale s číslami 15 a 20. Rovnakou úvahou ako predtým by sme zistili, že posledný hod nemohol byť znak (čísla pred hodom by museli byť -5 a 20), ale musel byť hlava. Takže dostaneme čísla 15 a 5. A takto postupujeme, až kým nemáme dve rovnaké čísla:

Hody		Z	Z	H	H
Prvé	5	10	15	15	15
Druhé	5	5	5	20	35

Začiatočné čísla p a d boli rovné číslu **5**, a skrátená postupnosť hodov je **2Z, 2H**.

Tento príklad bol pekný. Keď sme išli odzadu, tak v každom kroku nám vyšla iba jedna možnosť hodu, druhá nebola možná, lebo viedla k záporným číslam. Musí to tak však byť vždy? Vieme si z tohto príkladu odvodiť aj nejaké všeobecnejšie pozorovania?

- Ak máme výsledné čísla P a D, tak predchádzajúce čísla P' a D' vieme zistiť pomocou jedného odčítania (ak vám nasledujúce rovnice nie sú jasné, skúste sa vrátiť vyššie, kde sme odvádzali hodnoty P' a D'):
  - Hod Z:

$$P' = P - D$$
$$D' = D$$

  - Hod H:

$$P' = P$$
$$D' = D - P$$

- Ak sú čísla P a D rôzne, tak práve jedno z $P - D$ alebo $D - P$ je kladné.
- Aby sme nedostali záporné čísla, tak vždy odčítavame menšie číslo od väčšieho. To zároveň určuje, posledný hod mince. Ak je menšie prvé číslo, posledný hod bol H, ak je menšie druhé číslo, posledný hod bol Z.
- Keďže čísla sú vždy rôzne, tak nikdy nemáme na výber medzi hlavou a znakom. Rovnaké čísla môžeme mať iba na začiatku nášho zisťovania, keď sme prišli k hľadaným číslam z rána.
- Postupnosť hodov zisťujeme odzadu.

Podúloha b)

Dášenka doniesla Bobovi čísla $P = 111$ a $D = 9$.

Rovnako ako v podúlohe a) to vieme spraviť postupným odčítavaním odzadu, čím dostaneme takýto výsledok:

Akcia		H	H	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z
Prvé	3	3	3	12	21	30	39	48	57	66	75	84	93	102	111
Druhé	3	6	9	9	9	9	9	9	9	9	9	9	9	9	9

Určite vás ale nebavilo odčítavať 9 stále a stále dookola – až dvanásťkrát. Nedalo by sa to spraviť rýchlejšie?

Vieme že 9 potrebujeme odčítať od čísla 111 toľkokrát, aby toto prvé číslo bolo menšie ako 9. Takže ak je rozdiel medzi 111 a 9 rovný 102, tak potrebujeme odčítať toľko 9, aby ich súčet bol aspoň 102. Zároveň to ale nemôžeme prehnať a dostať sa k záporným číslam. Hľadáme teda najmenší počet 9, ktorých súčet je aspoň 102.

Naša otázka teda je: "Koľkokrát musíme vynásobiť číslo 9, aby sme dostali číslo 102?", na čo si vieme odpovedať pomocou delenia $102/9 \doteq 11,3$. Vidíme, že 11 deviatiek nebude stačiť a potrebujeme ich 12 (lebo $12 \cdot 9 = 108$ a to je naozaj väčšie ako 102). Mohli by sme teda povedať, že potrebný počet deviatiek je číslo $\frac{102}{9}$ zaokrúhlené nahor, teda 12.

Iný pohľad na to je, že potrebujeme od 111 odčítavať 9 kým nebudeme dostávať záporné čísla. Čo je vlastne inak povedané, "Koľkokrát sa deviatka zmestí do čísla 111?". To sa dá vyjadriť podielom $111/9 \doteq 12,3$, ktorý zaokrúhlime nadol, dostaneme teda číslo 12.

Vidíme, že oba postupy nám dávajú rovnaký výsledok – 12 odčítaní, ktorý sedí s naším predošlým riešením. Namiesto dvanástich odčítaní teda spravíme jedno delenie, do skrátenej postupnosti hodov si zapíšeme $12Z$ a nové hodnoty P a D budú $111 - 12 \cdot 9$, teda 3 a 9.

Tento postup vieme samozrejme zovšeobecniť. Majme čísla P a D, o ktorých predpokladajme, že $P > D$ (ak to bude naopak, tak postup bude takmer rovnaký). Chceme preto zistiť, koľko znakov v rade padlo na minci.

Pýtame sa teda, koľkokrát sa hodnota D zmestí do P, čo vieme zistiť pomocou celočíselného delenia P/D (celočíselné delenie zaokrúhľuje nadol), túto hodnotu si označme k. Následne si do postupnosti hodov zapíšeme kZ, hodnota D ostane nezmenená a novú hodnotu P vypočítame ako $P - k \cdot D$.

Môžete si všimnúť ešte jednu zaujímavú vlastnosť. Od čísla P odpočítavame hodnotu D najviac krát ako vieme. To čo nám ostane je preto zvyšok po delení čísla P číslom D. Tomu naozaj zodpovedá aj hodnota $P - k \cdot D$. A keďže väčšina programovacích jazykov vie zvyšok po delení počítať priamo, najrýchlejší spôsob ako zistiť novú hodnotu P je použiť príkaz `P = P%D` (`%` označuje zvyšok po delení).

Čo ak je nová hodnota nula? Túto možnosť, že $P - k \cdot D = 0$ sme doteraz ignorovali. Všimnime si však, že v takomto prípade je číslo P násobkom čísla D. Ak teda od P odčítam D iba $k - 1$ krát, dostanem dve rovnaké čísla, obe rovné D. Takáto situácia preto nastane iba raz, a to na konci nášho výpočtu. Do skrátenej postupnosti vtedy zapíšeme $(k-1)Z$ a za hodnoty čísel p a d dosadíme aktuálne D.

Podúloha c)

Vedeli by ste riešiť takýto problém pre ľubovoľnú dvojicu čísel P a D?

V predošlých častiach sme si ukázali všetky potrebné postupy, ostáva spojiť ich dokopy.

- Ak je $P > D$ tak zisťujeme dĺžku sekvencie *znakov*
  $k = \frac{P}{D}$ zaokrúhlené nadol
  hodnota D sa nemení
  - Ak P nie je deliteľné D
    zmeníme hodnotu P na $P - k \cdot D$
    zapíšeme kZ do skrátenej postupnosti
  - Ak je P deliteľné D
    zmeníme hodnotu P na $P - (k-1) \cdot D$ (čo je vlastne D)
    zapíšeme $(k-1)Z$ do skrátenej postupnosti
- Ak je $D > P$ tak zisťujeme dĺžku sekvencie *hláv*
  $k = \frac{D}{P}$ zaokrúhlené nadol
  hodnota P sa nemení
  - Ak D nie je deliteľné P
    zmeníme hodnotu D na $D - k \cdot P$
    zapíšeme kH do skrátenej postupnosti
  - Ak P delí D
    zmeníme hodnotu D na $D - (k-1) \cdot P$ (čo je vlastne P)
    zapíšeme $(k-1)H$ do skrátenej postupnosti

Tento postup spraví jeden krok nášho postupu. Budeme ho preto opakovať až kým sa čísla P a D nebudú rovnať. Všimnite si naviac, že v každom kroku sa zmení to, ktoré číslo je väčšie, takže naša skrátená postupnosť bude striedať hlavy a znaky.

Tento postup vieme samozrejme zapísať aj v ľubovoľnom programovacom jazyku. Skúste si pozrieť nižšie uvedenú implementáciu v Pythone. Zistíte, že sa veľmi podobá tomu, ako sme si to zapísali pred chvíľou.

Listing programu (Python)

```
print("Napis prve cislo:")
P = int(input())                                    # napr. číslo 111
print("Napis druhe cislo:")
D = int(input())                                    # napr. číslo 9

zoznam = []                                         # tu zapisujeme skrátenú postupnosť
while P != D:                                       # kým sa čísla P a D nerovnajú opakuj
    print(P,D)
    if P > D:
        k = P // D # celočíselné delenie
        if P % D != 0:  # číslo P nedáva po delení D zvyšok 0, nie je ním teda deliteľné
            P = P - k*D
            zoznam.append((k, 'Z'))
        elif P % D != 0:  # zvyšok je 0, P je deliteľné D
            P = P - (k-1)*D
            zoznam.append((k-1, 'Z'))
```

```
    elif D > P:
        k = D // P # celočíselné delenie
        if D % P != 0: # číslo D nie je deliteľné P
            D = D - k*P
            zoznam.append((k, 'H'))
        elif D % P == 0:  # číslo D je deliteľné P
            D = D - (k-1)*P
            zoznam.append((k-1, 'H'))

# keďže zisťujeme ťahy odzadu, tak musíme zoznam obrátiť
zoznam.reverse()
print("Skrateny zoznam hodov:", zoznam)             # [(2, 'H'), (12, 'Z')]

# vypiseme aj zaciatocne cislo
print("Rano dostala Dasa cisla rovne hodnote:", P)  # 3
```

Je to dostatočne rýchle?

Pokúsme sa zamyslieť nad tým, koľkokrát musíme zopakovať vyššie uvedený postup. V jednom kroku sa rozhodneme, ktoré z čísel P a D je väčšie a toto väčie číslo upravíme, pričom bude menšie ako druhé číslo. Takže to, ktoré číslo je väčšie, sa bude v každom kroku striedať. Ak je najskôr $P > D$, tak hneď potom bude $P < D$, následne opäť $P > D$, a tak ďalej až kým $P = D$.

Pri každom tomto kroku vložíme do skrátenej postupnosti jeden *záznam*, teda jedno číslo a hod Z alebo H. Trvanie nášho programu bude teda priamo úmerné dĺžke skrátenej postupnosti. Stále však nevieme, aká dlhá bude táto postupnosť a či to náš algoritmus zvládne aj pre čísla veľkosti milión.

Pozrime sa bližšie na to, ako sa mení číslo P. V každom druhom kroku sa veľkosť tohto čísla zmenší, keďže hodnoty P a D sa zmenšujú nastriedačku. Ako rýchlo sa toto číslo zmenšuje? Ak by sa v každých dvoch krokoch zmenšilo iba o 1, tak je to zlé. Pretože naša skrátená postupnosť by potom mala dĺžku $2 \cdot P$, čo je veľa. Môže sa však číslo P zmenšiť o tak málo? Skúste nájsť takú dvojicu P a D, že po úprave čísla P sa toto číslo zmenší iba o 1.

V skutočnosti totiž platí, že hodnota čísla P sa musí pri jednej úprave **zmenšiť aspoň na polovicu**. Ukážme si, prečo je to tak. Všimnite si, že číslo P sa mení v závislosti od čísla D. Dokonca sme si ukázali, že výsledná hodnota P musí byť po úprave menšia ako hodnota D. Ak je teda číslo D malé, bude malá aj nová hodnota P[1]. Nová hodnota P preto bude aspoň polovičná ak platí, že $D \leq \frac{P}{2}$ – číslo D je menšie ako polovica P a preto aj P bude po úprave menšie ako jeho polovica.

Čo však v prípade, keď $D > \frac{P}{2}$? Pozrime sa na to, čo spraví náš algoritmus. Najskôr zistí, koľkokrát sa do čísla P zmestí číslo D. Odpoveďou je raz, pretože D je väčšie ako polovica P, takže dvakrát sa tam nezmestí. V ďalšom kroku sa od P jedenkrát odpočíta číslo D. No a výsledok musí byť menší ako polovica P. Ak totiž máte číslo P a zoberiete z neho viac ako jeho polovicu (číslo D), ostane vám menej ako polovica (nová hodnota P).

Tým pádom sa po každých dvoch krokoch hodnota čísla P zmenšila aspoň na polovicu. Čo sa stane, ak bude číslo P milión? V najhoršom prípade sa toto číslo zakaždým zmenší presne na polovicu až kým z neho neostane číslo 1, čo je najmenšie číslo, s ktorým Dášenka mohla začínať.

$$1000000/2 = 500000/2 = 250000/2 = 125000/2 = 62500/2 = 31250/2 = 15625/2 \doteq 7812/2 = 3906$$

$$3906/2 = 1953/2 \doteq 976/2 = 488/2 = 244/2 = 122/2 = 61/2 \doteq 30/2 = 15/2 \doteq 7/2 \doteq 3/2 \doteq 1$$

Predošlý zápis znázorňuje, ako by sa hodnota P zmenšovala[2]. Vidíme, že na to, aby sme z nej dostali 1 potrebujem iba 19 krokov. Samozrejme, nezabudnime, že hodnota P sa zmení iba v každom druhom kroku, niekedy sa totiž musí zmenšovať aj D. Stále z toho však vychádza, že nám stačí 38 krokov. A aj to iba v najhoršom možnom prípade.

Pre zaujímavosť, číslo 19 – koľkrát môžme vydeliť číslo 1 000 000 číslom 2 – je aj približná hodnota dvojkového logaritmu z čísla 1 000 000. Logaritmovanie je v podstate opak umocňovania.

Keď chceme skrátene zapísať súčin veľa dvojok, tak to môžme skrátene napísať pomocou umocnenia:

$$2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 2^8$$

Poprípade súčin k dvojek je 2^k.

Logaritmom sa označuje odpoveď na otázku: Na koľkú musíme umocniť číslo 2, aby sme dostali výsledok y? Ak by sme teda chceli zistiť, na koľkú treba umocniť 2, aby sme dosali číslo 256 (teda hľadáme x rovnice $2^x = 256$), tak odpoveďou by bol práve $\log_2(256)$, čo je 8.

[1] To sme videli napríklad v príklade, kde $P = 111$ a $D = 9$. V jednom kroku sa P zmenšilo na hodnotu 3, muselo byť totiž menšie ako D.

[2] Takýto zápis samozrejme nie je matematicky správny. Určite totiž neplatí, že $488/2 = 244/2$. Pekne však znázorňuje, čo sme chceli ukázať, a to je trochu úspornejšie ako písať $488/2 = 244; 244/2 = 122 \ldots$.

O našom postupe by sme preto mohli povedať, že potrebuje najviac $2 \cdot \log_2 P$ krokov.

Podúloha d)

Vedeli by ste dokázať, že počiatočné číslo p delí obe výsledné čísla P a D?

Začať môžeme tým, že sa vrátime ku konkrétnym príkladom z úloh a) a b). Všimnime si, že v oboch prípadoch platí, že číslo p nedelí len výsledné čísla P a D, ale všetky čísla, ktoré si Dášenka zapísala. V podúlohe a) vidíme iba násobky čísla 5, v podúlohe b) iba násobky čísla 3. To je podozrivé.

Pozrime sa teda, čo sa deje keď Dášenka začne hádzať mincou. Začína s dvoma číslami p a d, ktoré sú rovnaké, a teda zjavne sú obe deliteľné číslom p. Po prvom hode mincou môže mať Dášenka zapísané čísla $2p$ a p, ktoré sú opäť deliteľné číslom p. Po druhom hode mincou má buď dvojicu $3p$ a p, alebo dvojicu $2p$ a $3p$, čo sú všetko násobky p.

No a táto vlastnosť sa nezmení. Čo totiž robí Dášenka? Jedno číslo nahradí súčtom a druhé nechá nezmenené. Ale ak sčítame dve čísla deliteľné číslom p, tak opäť dostaneme číslo deliteľné p. Skúsme si to ukázať o niečo formálnejšie. Číslo deliteľné p si vieme zapísať ako $a \cdot p$ pre nejaké kladné a. Zoberme si teda ľubovoľné dve čísla deliteľné p, ktoré vieme zapísať ako $a \cdot p$ a $b \cdot p$. Ich súčet je $a \cdot p + b \cdot p = (a + b) \cdot p$. Tento súčet je ale zjavne opäť deliteľný číslom p, lebo sa dá zapísať ako $c \cdot p$, pre $c = a + b$.

Keďže Dášenka začína s dvoma číslami deliteľnými číslom p (dvojica p, d) a robí iba operáciu sčítanie, všetky čísla, ktoré zapisuje sú naďalej deliteľné číslom p. Tým pádom to platí aj pre výsledné čísla P a D.

Podúloha e)

Vedeli by ste dokázať, že ak nejaké číslo x delí obe výsledné čísla P a D, tak delí aj pôvodné číslo p?

Skúsme postupovať podobne ako v podúlohe d). Zoberme si číslo x, ktoré delí aj P aj D. Tieto dve čísla sa teda dajú zapísať ako $P = a \cdot x$ a $D = b \cdot x$, pre vhodné a a b. Čo sa s týmito číslami dialo krok pred tým? V podúlohe a) sme si ukázali, že hodnoty predchádzajúcich čísel získame tak, že menšie číslom nezmeníme a od väčšieho to menšie odčítame. Tvárme sa teda, že $P > D$, pre opačný prípad by sme robili to isté, akurát naopak.

Hodnota D ostane rovná $b \cdot x$, ale hodnota P sa zmení na rozdiel týchto dvoch čísel. Bude preto platiť, že $P = a \cdot x - b \cdot x$. To však vieme zapísať ako $P = (a - b) \cdot x$. Nová hodnota čísla P bude teda opäť deliteľná číslom x.

Naviac, táto vlastnosť platí všeobecne a vyššie máme uvedený aj jej dôkaz. Ak si zoberieme dve čísla deliteľné x, tak ich rozdiel bude opäť deliteľný číslom x. Pri našom postupe teda budeme celý čas pracovať s číslami, ktoré sú deliteľné číslom x. To bude platiť aj keď sa na konci nášho postupu dostaneme k číslam p a d. Ak teda číslo x delí výsledné čísla, delí aj začiatočné p.

Čo vyplýva z vlastností d) a e) o čísle p?

V podúlohe d) sme ukázali, že číslo p delí obe čísla P a D. Takúto hodnotu označujeme ako *spoločný deliteľ*. Podúloha e) zase hovorí, že každý spoločný deliteľ x čísel P a D delí aj číslo p. Z toho ale vyplýva, že $x \leq p$. Uvedomme si, že ak nejaké číslo a delí číslo b, tak b musí byť väčšie alebo rovné a.

Ak je teda každý spoločný deliteľ menší alebo rovný ako p tak to značí, že p je **najväčší spoločný deliteľ** čísel P a D.

Na záver

V tejto úlohe sme teda celý čas počítali najväčšie spoločné delitele dvoch čísel. A postup, ktorý ste počas toho vymysleli sa volá Euklidov algoritmus a je to najrýchlejší spôsob ako túto hodnotu vypočítať.

Viac o Euklidovom algoritme si môžete prečítať na Wikipédii (https://sk.wikipedia.org/wiki/Euklidov_algoritmus). Jediný rozdiel s naším riešením je ten, že klasický Euklidov algoritmus skončí, keď je jedno z čísel rovné nule a nie keď sú rovnaké. Vďaka tomu netreba špeciálne ošetrovať podmienku deliteľnosti čísla P číslom D.

3. Pohodlnejšie Kreslo

vzorák napísal(a) Dávid
(max. 15 b za riešenie)

Vzorové programy nájdete aj na stránke s editorom ksp.sk/~prask/specialne/4/2/3, kde si môžete odsimulovať ich beh. Pri čítaní vzorového riešenia odporúčame si to aj vyskúšať. Hoci vám slovne načrtneme základné myšlienky, predsa len je to prehľadnejšie, keď sa to aj hýbe.

Podúloha a.

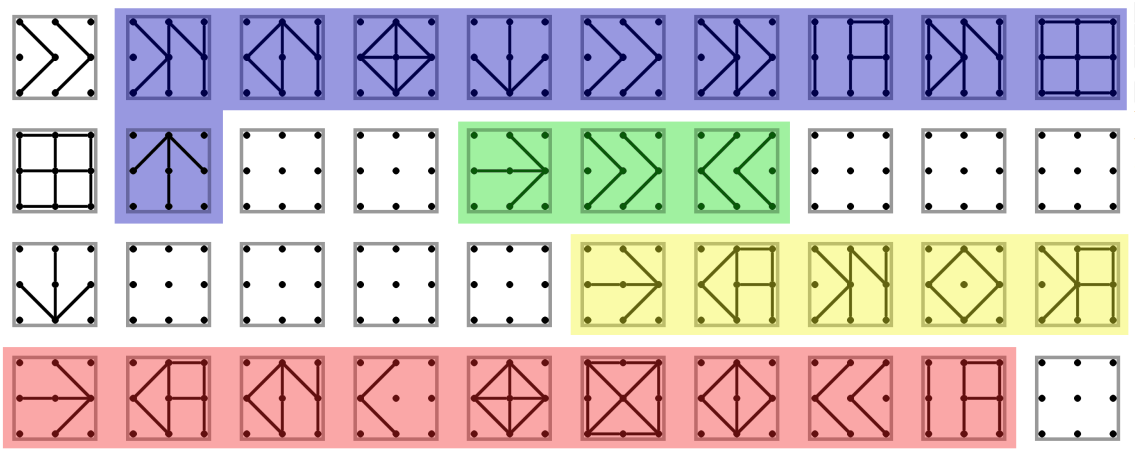
Na vstupe je číslo n a potom dva krát n čísel. Na výstup vypíšte najskôr druhých n čísel a po nich prvých n čísel.

Riešenie tejto úlohy sa skladá z troch častí. Načítame prvú polovicu, načítame a vypíšeme druhú polovicu a na koniec vypíšeme prvú polovicu.

Prvá časť (modrá) načítava n čísel a ukladá ich do pamäte. Posledný príkaz v riadku je tam aby preskočil načítanie čísla n, ktoré sme použili na začiatku. Šípka o riadok nižšie kompenzuje otočenie, ktoré spôsobí príkaz nad ňou, keď nemá čo uložiť do N (keď preskočíme načítanie pred ním).

Druhá časť (zelená) načítava a rovno vypisuje druhú polovicu čísel. Nasleduje krátky žltý úsek, ktorý pripraví hodnoty v N a A pred treťou časťou.

Tretia časť (červená) vypíše n čísel z pamäte a ukončí program.



Podúloha b.

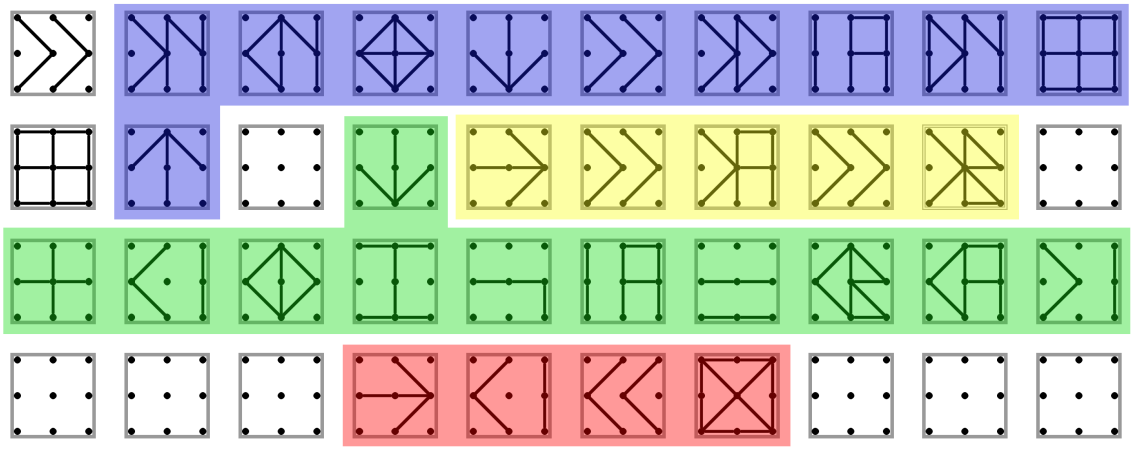
Na vstupe je číslo n a potom n čísel. Za nimi sú dve čísla a a b. Na výstup vypíšte súčet čísel medzi a-tym a b-tym číslom (vrátane).

Zadanie začína podobne ako predošlá úloha. Vďaka tomu môže aj náš program začínať rovnako. Prvá časť (modrá) je preto úplne rovnaká.

V druhej časti (žltá) nám tento krát stačí načítať len dve čísla, preto žiadny cyklus.

V tretej časti budeme zväčšovať číslo v A, ktoré zároveň určuje miesto v pamäti, až kým sa dostaneme na číslo v B. V každom kole ešte pripočítame obsah D do I, v ktorom bude na konci súčet celého úseku.

Na koniec (červená) už len vypíšeme obsah I.



Podúloha c.

Na vstupe je n čísel. Vypíšte najmenšie a druhé najmenšie číslo.

Jednou z možností ako riešiť túto úlohu je vyriešiť podúlohu e. a mierne ju upraviť. Dá sa to však aj oveľa jednoduchšie a dokonca nepotrebujeme ani pamäť. Stačia nám dva registre. Budeme používať A, kde bude najmenšie číslo a B, kde bude druhé najmenšie.

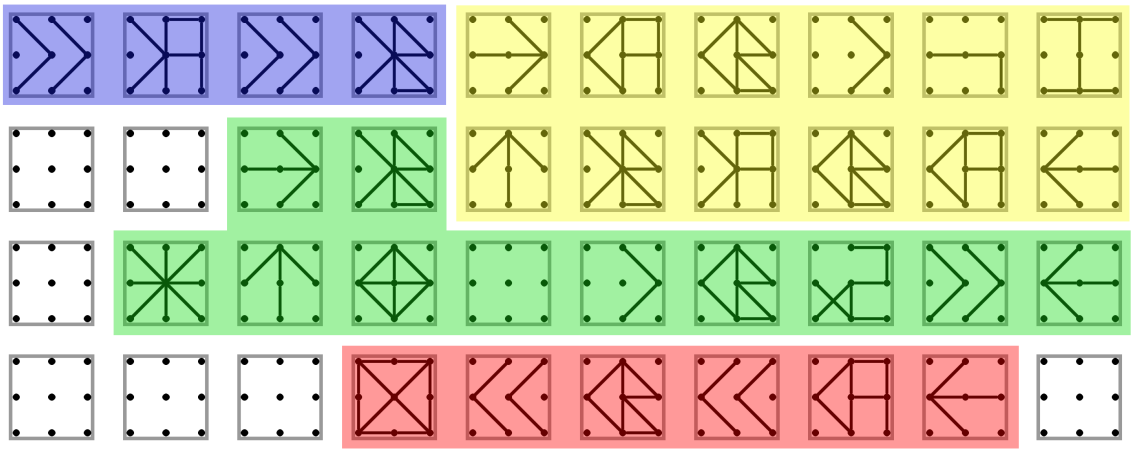
Ako prvé (modrá) musíme do týchto registrov niečo uložiť, aby sme nepracovali s nulou, ktorá tam je na začiatku. Použijeme prvé dve čísla na vstupe.

Následne prejdeme do žltej časti, ktorá skontroluje, či v A nieje väčšie číslo ako v B, a prípadne ich vymení. Keď ich vymení, podmienka bude už splnená, čiže sa to zopakuje najviac dva krát.

Vždy keď skončí žltá časť, prejdeme do zelenej, ktorá postupne načítava zvyšok vstupu. Každé číslo po načítaní zduplikujeme a porovnáme s B. Ak je menšie, znamená to, že je jedným z dvoch doteraz najmenších tak ním prepíšeme B. Po takejto zmene opäť prejdeme do žltej časti.

Ak načítané číslo nie je menšie, vymažeme jeho druhú kópiu a pokračujeme na ďalšie.

Nakoniec, keď sa minú čísla na vstupe, (červená) už len vypíšeme A a B.



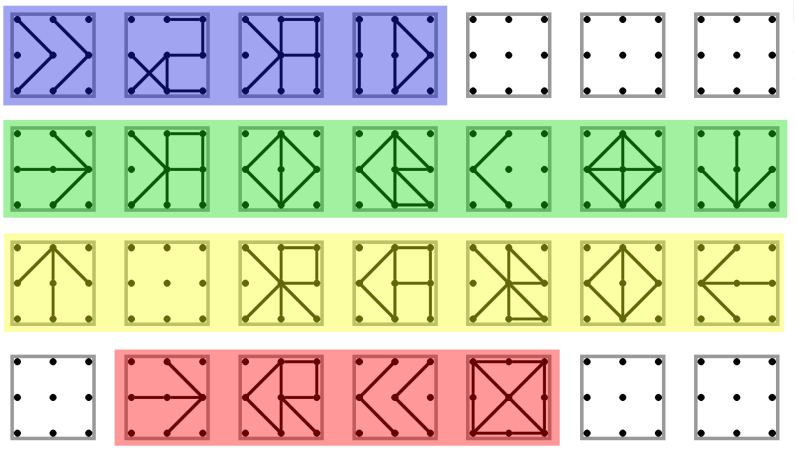
Podúloha d.

Na vstupe je n čísel. Vypíšte číslo, ktoré je na vstupe najviac krát.

Jedno z možných riešení, je opäť použiť riešenie podúlohy e, a v utriedenej postupnosti nájsť najdlhší úsek rovnakých čísel. Existuje však aj oveľa elegantnejšie riešenie, ktoré využíva úplne iný prístup ako predošlé úlohy.

Tento prístup sa líši v tom, že namiesto toho aby sme si čísla zo vstupu ukladali do pamäte, budeme ich používať ako adresu do pamäte a v pamäti bude hodnota koľko krát sme dané číslo už videli (modrá časť). Aby sme sa k týmto hodnotám dostali aj po tom, ako čísla načítame, budeme si všetky čísla zo vstupu kopírovať a ukladať na zásobník.

Keď máme takto načítaný vstup, dostávame sa k druhej časti (zelená), kde prechádzame cez všetky čísla na zásobníku a hľadáme to, ktoré sa na vstupe objavilo najviac krát. Keď nájdeme číslo, ktoré bolo na vstupe viac krát ako to, ktoré si pamätáme, pokračujeme na žltú časť, kde si uložíme do B koľko krát to bolo a do R aké číslo to bolo. Keď sa nám minú čísla na zásobníku, určite sme našli to, ktoré sa vyskytlo najviac krát. Preto prejdeme na červenú časť a vypíšeme hodnotu z R.



Podúloha e.

Na vstupe je n čísel. Vypíšte všetky tieto čísla zoradené od najmenšieho po najväčšie.

Zoradiť čísla je úloha ktorá sa v informatike vyskytuje veľmi často a preto ľudia vymysleli veľa spôsobov ako to spraviť[3]. Niektoré sú pomalšie a niektoré rýchlejšie. Nám viac záleží na jednoduchosti ako na rýchlosti, preto sme si vybrali Insertsort.

Jeho názov je odvodený od slova insert (vložiť), pretože postupne pridáva ďalšie a ďalšie číslo do už zoradenej postupnosti.

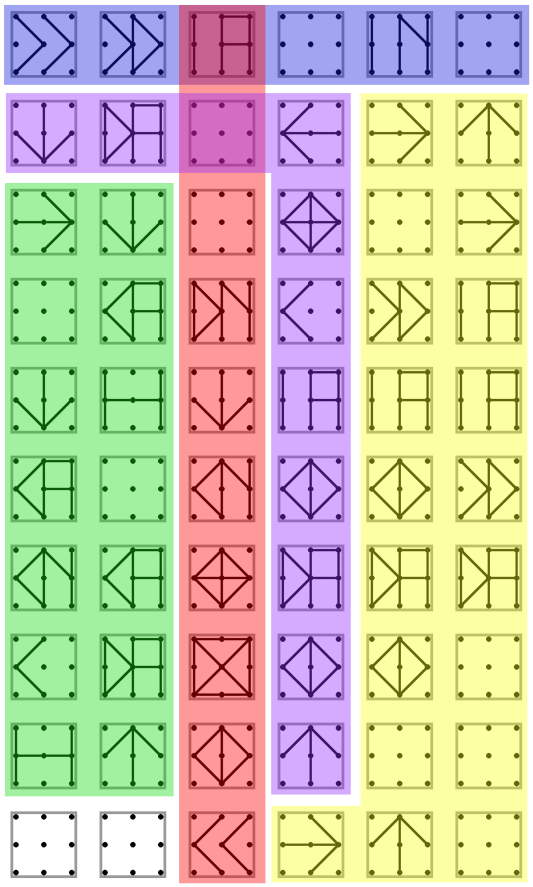
Náš program bude zoraďovať čísla v našej postupnosti od konca. Znamená to, že sa najprv budeme pozerať iba na posledné číslo. To je samo o sebe v správnom poradí (nemá na výber). Potom k nemu pridáme predposledné, to pred ním, až po to prvé. Predstavme si, že už máme nejakú postupnosť zoradenú a chceme do nej pridať ďalšie číslo. Nové číslo máme na začiatku tejto postupnosti. Porovnáme ho s číslom za ním. Ak je väčšie, znamená to, že má byť až za ním – vymeníme ich. Opäť sa pozrieme na naše nové číslo a to za ním a prípadne ich vymeníme a opakujeme. Keď narazíme na to, že ďalšie číslo je väčšie, znamená to, že aj čísla za ním budú väčšie, a čísla pred našim číslom musia byť menšie, keďže sme ich preskočili. Poradie pôvodných čísel sme nezmenili a nové číslo je na správnom mieste – máme zoradenú postupnosť.

Program sme rozdelili na 5 častí. Prvá a najjenoduchšia z nich (modrá) opäť slúži len na načítanie vstupu a posledná (červená) vypíše utriedené čísla z pamäte.

Fialová časť skontroluje, či je číslo v pamäti väčšie ako číslo pred ním. Ak je, znamená to, že ich treba vymeniť a prejdeme na žltú časť. Inak, ak je menšie alebo rovnaké, posunieme sa o jedno dozadu a pokračuje na zelenú časť.

Žltá časť vymení číslo v pamäti s tým ktoré je pred ním, posunie sa o jedno ďalej a pokračuje na zelenú.

Zelená časť dáva pozor na to, či sme sa neposunuli príliš ďaleko, ale aj na to, či sme sa ešte nedostali na začiatok. Ak sme sa dostali za koniec poľa, znamená to, že sme vymieňali posledné dve čísla a teraz sú už v správnom poradí. Preto nám stačí adresu zmenšiť o 1 a vrátiť sa tak ku poslednému číslu. O ostatné sa už postará fialová časť. Ak sme sa dostali na začiatok poľa, fialová časť musela skontrolovať všetky čísla, čiže máme usporiadané pole a môžeme čísla vypísať. Vtedy prejdeme na červenú časť.



[3] https://sk.wikipedia.org/wiki/Triediaci_algoritmus

4. Prefixove rezne

vzorák napísal(a) Jaro
(max. 15 b za riešenie)

Brute-force

Ak vám nenapadlo žiadne efektívne riešenie, jedna z prvých vecí, ktoré treba skúsiť je riešenie hrubou silou. Každé správne riešenie, hoci aj pomalšie, zväčša získa nejaké body. Preto sa ho nabudúce nebojte odoslať – radšej odovzdať riešenie, ktoré vám napadlo ako prvé, než mať za úlohu 0 bodov.

Najjednoduchšia vec, ktorú môžeme spraviť je vyskúšať všetky možné úseky rezňov a pre každý z nich zistiť, či sa v ňom nachádzajú všetky exisexistujúce druhy. Každý súvislý úsek rezňov vieme určiť jeho začiatkom a koncom, ktoré si označíme premennými z a k. V dvoch cykloch preto vieme prejsť cez všetky možné dvojice hodnôt z a k.

Pre daný úsek však potrebujeme vedieť zistiť, či sa v ňom nachádzajú všetky druhy rezňov. Na to si vytvoríme pole `videli[]`, do ktorého si budeme značiť, koľko rezňov daného typu sa nachádza v úseku od z po k. Presnejšie, v premennej `videli[i]` bude počet rezňov typu i. Pomocou cyklu potom prejdeme zadaný úsek, do poľa `videli[]` si spočítame počty rezňov daného typu a posledným cyklom zistíme, či je každá hodnota `videli[i]` aspoň 1 – v úseku je aspoň jeden rezeň typu i. Ak náš úsek spĺňa zadanú podmienku a je najkratší z dobrých úsekov, ktoré sme zatiaľ videli, tak si ho zapamätáme.

Časová zložitosť tohto riešenia je $O(n^3)$. Rôznych úsekov, teda dvojíc z a k, je zhruba n^2 a pre každý z nich musíme prejsť vybraný úsek a zistiť, či je v ňom každý rezeň aspoň raz, čo bude trvať $O(n+m)$ operácií. Pamätať si potrebujeme iba rezne zo vstupu, ktorých je n, a pole `videli[]`. Pamäťová zložitosť je preto $O(n+m)$

Listing programu (C++)

```
#include <bits/stdc++.h>
using namespace std;

const int INF = 1000000000;

int main () {
    int n, m;
    cin >> n >> m;
    vector<int> rezne(n);
    for (int i = 0; i < n; i++) {
        cin >> rezne[i];
        rezne[i]--;
    }

    int najkratsi = INF;
    for (int z = 0; z < n; z++) {
        for (int k = z; k < n; k++) {
            vector<int> videli(m, 0);
            for (int i = z; i <= k; i++) {
                videli[rezne[i]]++;
            }
            for (int i = 0; i < m; i++) {
                if (videli[i] < 1) break;
                if (i+1 == m) {
                    najkratsi = min(najkratsi, k-z+1);
                }
            }
        }
    }

    if (najkratsi == INF) cout << -1 << endl;
    else cout << najkratsi << endl;
}
```

Zrýchlenie predchádzajúceho riešenia

Ďalším dôvodom, pre ktorý je dobré začať s pomalým riešením je, že pomalé riešenie slúži často ako základ efektívnejšieho riešenia. Skúsme sa teda pozrieť na to, čo robilo naše predchádzajúce riešenie naviac. Keď sa pozrieme, v akom poradí skúšame úseky, zistíme, že po úseku od z po k nasleduje úsek od z po $k+1$. Ten sa však od predchádzajúceho úseku líši iba tým, že má o jeden rezeň viac – rezeň na pozícii $k+1$. Teda ani pole `videli[]` sa veľmi nezmení, stačí pridať tento jeden rezeň. Namiesto toho, aby sme teda opäť prechádzali celý úsek, pridáme iba jediný rezeň.

Pre každý z n^2 úsekov tak spravíme iba jednu operáciu na zmenu poľa `videli[]` a následne sa pozrieme, či je v tomto poli každý rezeň aspoň raz. Časová zložitosť je teda $O(n^2 m)$, čo je o kúsok lepšie. Ešte však nie sme hotoví.

Prechodom na ďalší úsek nám pribudol jediný rezeň. Ak úsek od z po k neobsahoval každý rezeň aspoň raz, tak úsek od z po $k+1$ ho obsahuje iba ak na pozícii $k+1$ bol posledný chýbajúci druh rezňa. Ak sme teda pridali rezeň typu i, tak hodnota `videli[i]` sa zmenila z 0 na 1 (rezeň i v úseku nebol a zrazu tam jeden je). Naše riešenie si teda bude všímať, či sa hodnota poľa `videli[]` zmenila z 0 na 1 a bude počítať, koľko takýchto

udalostí nastalo. Počet týchto udalostí si bude pamätať v premennej d. V okamihu, keď uvidí m takýchto zmien tak vie, že v danom úseku je každý druh rezňa aspoň raz – každý druh totiž vytvorí jednu takúto udalosť prvýkrát keď ho do nášho úseku pridáme.

Výsledná časová zložitosť je $O(n^2)$. Pre každý úsek totiž iba zmeníme jednu hodnotu v poli `videli[]` a pozrieme sa, či táto zmena pridala do úseku nový druh rezňov.

Listing programu (C++)

```
#include <bits/stdc++.h>
using namespace std;

const int INF = 1e9;

int main () {
    int n, m;
    cin >> n >> m;

    vector<int> rezne(n);
    for (int i = 0; i < n; i++) {
        cin >> rezne[i];
        rezne[i]--;
    }

    int najkratsi = INF;
    for (int z = 0; z < n; z++) {
        vector<int> videli(m, 0);
        int d = 0;
        for (int k = z; k < n; k++) {
            videli[rezne[k]]++;
            if (videli[rezne[k]] == 1) d++;
            if (d == m) {
                najkratsi = min(najkratsi, k-z+1);
            }
        }
    }
    if (najkratsi == INF) cout << -1 << endl;
    else cout << najkratsi << endl;
}
```

Vzorové riešenie

Nerobíme však stále niečo zbytočne? Pozrime sa ešte raz na to, čo robíme. Zvolíme si začiatok z a postupne zväčšujeme pozíciu konca k. Do poľa `videli[]` si značíme počty rezňov jednotlivých druhov, ktoré sú v tomto úseku. A vždy, keď nájdeme nový druh rezňa, hodnota `videli[i]` sa zmení z 0 na 1, tak zväčšíme hodnotu d. Ak pre úsek platí, že $d = m$ tak vieme, že daný úsek obsahuje každý druh rezňa aspoň raz.

Hodnota d sa však iba zväčšuje. To znamená, že keď raz pre nejaký úsek platí $d = m$, tak aj všetky dlhšie úseky začínajúce na pozícii z obsahujú všetky druhy rezňov. Tieto úseky sú však dlhšie, preto sa nám ich neoplatí ani skúšať. Ako teda zväčšujeme hodnotu k, tak v okamihu, keď $d = m$, sme našli najkratší úsek začínajúci na z, ktorý obsahuje každý druh rezňa. Ďalšie, väčšie hodnoty k nemusíme ani skúšať a môžeme začať skúšať ďalší začiatok $z + 1$.

Toto ešte nestačí, uvedomme si však nasledujúcu vec. Najkratší úsek obsahujúci všetky rezne, ktorý začína na pozícii $z + 1$ nemôže končiť skôr ako aktuálna hodnota k. Pozícia k bola prvá taká, že úsek z až k obsahoval všetky druhy rezňov. Predstavme si, že úsek od $z + 1$ až k' obsahuje všetky rezne, pričom $k' < k$. Tak potom zjavne aj úsek od z po k' obsahuje všetky druhy rezňov. Je to predsa úsek, ktorý obsahuje o jeden rezeň viac ako úsek od $z + 1$ po k'. Hodnota k však bola najmenšia taká, takže žiadne menšie k' existovať nemôže.

Keď teda začneme so začiatkom $z+1$ nemusíme skúšať všetky úseky, ktoré končia skôr ako k. Tie totiž určite všetky rezne obsahovať nebudú. Vieme však vhodne upraviť hodnoty v poli `videli[]` a d? Úseky od z po k a od $z + 1$ po k sa líšia iba v rezni na pozícii z, ktorý sme odobrali. Ak bol druhu i tak to znamená, že musíme zmenšiť hodnotu `videli[i]` o 1. A počet druhov v tomto úseku (hodnota d) sa zmení iba ak sme odstránili jediný rezeň svojho druhu, teda ak sa hodnota `videli[i]` zmenila z 1 na 0.

Ľahko teda upravíme naše polia a hodnotu d. Následne môžeme opäť zvyšovať koniec k, až kým sa d nebude znova rovnať hodnote m. Vtedy posunieme začiatok z a pokračujeme. Pre každý začiatok tak nájdeme najkratší úsek obsahujúci všetky druhy rezňov. Z týchto úsekov zoberieme ako výsledok najkratší z nich.

Časová zložitosť takéhoto riešenia je $O(n)$. Uvedomme si, že v každom kroku nášho algorimu o 1 zväčšíme buď hodnotu z alebo k. Obe tieto hodnoty môžeme zväčšiť najviac n krát, preto spravíme najviac $2n$ operácií. Pri každom posune pritom upravíme iba jedinú pozíciu poľa `videli[]` a premennú d. Pamäťová zložitosť je stále $O(n + m)$.

Všimnite si, ako sme sa z najľahšieho riešenia postupne prepracovali až k vzoráku. A stačilo si iba všímať, čo robíme navyše.

Listing programu (C++)

```cpp
#include <bits/stdc++.h>
using namespace std;

const int INF = 1e9;

int main () {
    int n, m;
    cin >> n >> m;

    vector<int> rezne(n);
    for (int i = 0; i < n; i++) {
        cin >> rezne[i];
        rezne[i]--;
    }

    int najkratsi = INF;
    vector<int> videli(m, 0);
    int d = 0;
    int k = 0;
    for (int z = 0; z < n; z++) {
        while(k < n && d != m) {
            videli[rezne[k]]++;
            if (videli[rezne[k]] == 1) {
                d++;
            }
            k++;
        }
        if (d == m) {
            najkratsi = min(najkratsi, k-z);
        }
        videli[rezne[z]]--;
        if(videli[rezne[z]] == 0) {
            d--;
        }
    }
    if (najkratsi == INF) cout << -1 << endl;
    else cout << najkratsi << endl;
}
```

vzorák napísal(a) Roman
(max. 15 b za riešenie)

5. Písmenková šifra

Dĺžka k-teho slova

Začneme tým, že si spočítame dĺžku slova na k-tej pozícii. Napríklad pre $k = 28$ je hľadaná dĺžka 2, pretože 28-me slovo je `AB`, ktoré má dve písmená.

Koľko slov má dĺžku 1? Presne 26, sú to jednopísmenové slová `A` až `Z`. A koľko slov má dĺžku 2? Slovo dĺžky dva dostaneme tak, že vyberieme jedno písmeno na prvú pozíciu a jedno písmeno na druhú pozíciu. Na jednu pozíciu pritom vieme vybrať 26 rôznych písmen a tieto dve pozície sa navzájom neovplyvňujú. Dokopy máme teda $26 \cdot 26$, teda 26^2 možností. Vo všeobecnosti, slov dĺžky d je 26^d. Na každú z d pozícií totiž môžeme vybrať jedno z 26 písmen.

Teraz si uvedomme, že ak má k-te slovo dĺžku d, tak v čísle k sú započítané aj všetky kratšie slová, tie totiž predchádzajú tým s dĺžkou d. V takomto prípade preto vieme, že $k > 26 + 26^2 + 26^3 \cdots 26^{d-1}$. Súčet na ľavej strane totiž predstavuje počet všetkých slov kratších ako d – slov dĺžky 1, 2, 3... $d-1$.

Ak teda chceme zistiť dĺžku k-teho slova, hľadáme najmenšie d také, že $k \leq 26 + 26^2 + \cdots + 26^d$. To vieme ľahko výpočítať pomocou jedného `while` cyklu, v ktorom pripočítavame stále väčšie mocniny 26, až kým nedostaneme číslo väčšie alebo rovné k.

Tento postup si môžeme ilustrovať na nasledovnom príklade:

```
k = 743

                    26 <  743   // Slovo je dlhsie ako 1
      26 + 26*26 = 702 <  743   // Slovo je dlhsie ako 2
702 + 26*26*26 = 18278 >= 743   // Slovo ma dlzku 3
```

Prvé písmeno slova dĺžky k

Úspešne sme zistili dĺžku k-teho slova, ktorú sme označili d. Poďme sa teda zamerať iba na slová tejto dĺžky. Už vieme, že ich je 26^d. Koľké z nich však hľadáme? Nemôže to byť k-te, pretože v hodnote k sú zahrnuté aj všetky kratšie slová. Ak teda chceme zistiť, ktoré d písmenové slovo hľadáme, musíme od k odpočítať kratšie slová, ktorých je $26 + 26^2 + \cdots + 26^{d-1}$. Nová hodnota k bude preto rovná tomuto rozdielu.

Pokúsme sa teraz zistiť jednotlivé písmená tohto slova. Zameriame sa na poslednú vlastnosť, ktorú sme ešte nepoužili – abecedné usporiadanie. Vieme, že našich 26^d slov dĺžky d je usporiadaných abecedne, teda najskôr všetky slová, ktoré začínajú na písmeno `A`, potom všetky, čo začínajú na `B` atď. až po `Z`.

Naviac si uvedomme, že je rovnako veľa slov, ktoré začínajú na `A` ako tých, ktoré začínajú na `B`. Ak totiž zoberieme slovo začínajúce na `B` a toto prvé písmeno zmeníme na `A`, dostaneme slovo začínajúce na `A`. Zároveň, dve rôzne slová, ktoré začínajú na `B` vytvoria dve rôzne slová začínajúce na `A`. Dokonca, táto vlastnosť platí pre ľubovoľnú dvojicu písmen. Tým pádom je týchto 26^d slov rozdelených na 26 rovnako veľkých skupín, každá s 26^{d-1} slovami. (Tú istú vlastnosť si môžeme uvedomiť aj tak, že ak zafixujeme prvé písmeno, na každú zo zvyšných $d-1$ pozícií môžeme dať jedno z 26 písmen.)

Ak sa pozeráme iba na d písmenové slová tak vidíme, že prvých 26^{d-1} začína na písmeno `A`. Ak je teda $k \leq 26^{d-1}$, tak prvé písmeno hľadaného slova je `A`. A ak je k väčšie, tak môžeme pokračovať. Ďalších 26^{d-1} slov totiž začína na `B`, takže ak $k \leq 2 \cdot 26^{d-1}$, tak hľadané slovo začína na písmeno `B`. A keď ani to, môžeme sa pozrieť na písmená `C`, `D`...

Týmto spôsobom sa určite dopracujeme k prvému písmenu zadaného slova. Ak by sme chceli tento postup naprogramovať, mohli by sme použiť jedoduchý `while` cyklus, ktorý by skúšal, kam patrí k. Keďže sú však jednotlivé skupiny rovnako dlhé, vieme použiť malý trik. Uvedomme si totiž, že ak sa snažíme zistiť, v ktorej skupine sa nachádza číslo k, je to akoby sme sa pýtali: Koľkokrát sa do čísla k zmestí 26^{d-1}? A na túto otázku vieme odpovedať pomocou delenia. Je tu len jediný problém. Do čísel 1 až $26^{d-1}-1$ sa 26^{d-1} zmestí nulakrát, ale do čísla 26^{d-1} sa zmestí raz. A rovnako aj v ostatných skupinách. Aby sme sa tejto nepríjemnosti vyhli, tak nebudeme deliť číslo k, ale číslo $k-1$. Teraz naozaj platí, že číslo $\frac{k-1}{26^{d-1}}$ (zaokrúhlené nadol) určuje, do ktorej skupiny patrí hľadané slovo. A tiež poradie prvého písmena tohto slova v abecede.

Ďalšie písmená a kratšie slová

Poznáme dĺžku slova d a takisto jeho prvé písmeno, nech je to napríklad `T`. Zvyšok slova už nemusíme preto hľadať medzi všetkými slovami dĺžky d. Stačí sa zamerať na slová dĺžky d, ktoré začínajú na písmeno `T`. Ktoré z týchto slov hľadáme? Opäť to nebude k-te z nich. V hodnote k sú totiž zarátané aj všetky slová dĺžky d, ktoré začínajú na písmeno, ktoré je v abecede skôr ako `T`. Koľko je takýchto slov?

Písmeno `T` je dvadsiate v abecede. Pred ním je teda 19 iných písmen (`A` až `S`). A na každé z týchto písmen začína 26^{d-1} slov dĺžky d. Všetky tieto slová preto musíme od k odčítať. Nová hodnota k bude teda $k-19 \cdot 26^{d-1}$. V tomto momente máme v k naozaj správnu hodnotu – pozíciu hľadaného slova medzi všetkými slovami dĺžky d začínajúcich na `T`.

Chýba nám posledná vec. Všetky slová, ktoré nás zaujímajú začínajú písmenom `T`. Ako sú teda zoradené? Zoradené sú podľa zvyšných $d-1$ písmen, ktoré tvoria všetky slová dĺžky $d-1$ usporiadané podľa abecedy. Naša úloha sa preto nezmení, ak si povieme, že hľadáme k-te slovo dĺžky $d-1$. A to je predsa problém, ktorý sme už vyriešili v predchádzajúcej časti.

Vieme, že slová dĺžky $d-1$ sú rozdelené podľa prvého písmena na 26 skupín, každá po 26^{d-2} slov. A vieme aj to, ako zistiť prvé písmeno k-teho takéhoto slova. Keď to teda zistíme, toto písmeno bude druhým v poradí v našom pôvodnom probléme. A takisto sa nám problém opäť zmenší na hľadanie k-teho (pre vhodne upravené k) slova dĺžky $d-2$. Tento postup budeme môcť opakovať, zakaždým sa dozvieme o jedno písmeno viac a dĺžka hľadaných slov sa bude znižovať až na 0, kedy už nič nemusíme robiť, pretože také slová neexistujú.

Počet krokov algoritmu

Pozrime sa ešte na časovú zložitosť tohto algoritmu. Ak je dĺžka slova d, tak hľadanie tejto hodnoty bude trvať zhruba d operácií. A takisto nájdenie hľadaného slova bude trvať zhruba d operácií, pretože v každom kroku sa o jedna skráti dĺžka slov, na ktoré sa pozeráme. Zostáva zistiť, aká veľká je hodnota d.

Úplne na začiatku sme si ukázali, že platí:

$$k \leq 26 + 26^2 + 26^3 + \cdots + 26^d$$

Súčet napravo je počet všetkých slov dĺžky najviac d. Tento súčet je naviac súčet členov geometrickej postupnosti, na ktorý vieme použiť známy vzorec[4]:

$$k \leq \frac{26^{d+1} - 26}{25}$$

To znamená, že k je zhruba tak veľké ako hodnota 26^d. (Zanedbali sme nejaké konštanty, ktoré sa ale stratia v O-notácii.) Hodnotu d preto môžeme slovne popísať ako: číslo, na ktoré keď umocníme číslo 26, dostaneme hodnotu k.

No a na vyjadrenie takýchto hodnôt používajú matematici pojem logaritmus. Zapisuje sa to ako $\log_a b = c$, číta sa to ako logaritmus z b pri základe a sa rovná c a vyjadruje to, že ak umocníme hodnotu a na číslo c

[4] Ak neviete, čo je geometrická postupnosť, skúste si o nej niečo prečítať.

dostaneme číslo b, teda $(a^c = b)$. Časovú zložitosť preto môžeme vyjadriť ako $O(\log_{26} k) = O(\log k)$. Keďže tento výpočet urobíme pre každý riadok vstupu, tak celková zložitosť je $O(n \cdot \log k)$.

Opäť odporúčam pogoogliť si niečo o logaritmoch, je to v informatike veľmi používaný pojem a oplatí sa poznať vzorce na ich úpravu a vedieť, čo zhruba znamenajú.

Listing programu (C++)

```
#include <iostream>

using namespace std;

void solve() {
    long long k;
    cin >> k;
    long long s = 26, m = 26;

    // Zistenie dlzky slova
    // Premenna s postupne obsahuje pocet slov dlzky 1, 2, 3, atd.,
    // teda ziskava hodnoty 26, 26 + 26^2, atd.
    while (k > s) {
        m *= 26;
        s = s + m;
    }

    // Od k odpocitame vsetky kratsie slova. Nase slovo ma dlzku d,
    // takze kratsich slov bude 26 + 26^2 + ... + 26^(d-1).
    // Staci si uvedomit, ze premenna s obsahuje sumu
    // 26 + 26^2 + ... + 26^(d-1) + 26^d, kde d je dlzka nasho slova.
    // Potrebujeme z nej teda este odpocitat posledny clen 26^d, ale
    // to je prave hodnota v m.
    k = k - (s - m);

    // Nove m obsahuje pocet slov dlzky 26^(d - 1)
    m = m / 26;

    while (m > 0) {
        long long poz = (k - 1) / m;
        cout << (char) (poz + 'A');

        k = k - poz * m;
        m = m / 26;
    }

    cout << endl;
}

int main() {
    int n;
    cin >> n;
    for (int i = 0; i < n; ++i)
        solve();
    return 0;
}
```