**PRASK**
**I. ročník, 2014/15**
**Katedra základov a vyučovania informatiky FMFI UK,**
**Mlynská Dolina, 842 48 Bratislava**

# Vzorové riešenia 2. kola letnej časti

vzorák napísal Žaba
(max. 6 b za riešenie)

## 1. Prefíkaný kocúr

### Agátová

Na prvej ulici žije iba jeden prefíkaný kocúr a našou úlohou je zistiť, v ktorom dome. Keď prídeme do nejakého domu, v ktorom nebýva kocúr, zakaždým sa dozvieme, či býva v dome s vyšším alebo nižším číslom.

Nebudeme teda otáľať a rovno sa programu spýtame, či prefíkaný kocúr nebýva náhodou v dome 47. Možno budeme mať šťastie a uhádneme to na prvý pokus. Neprekvapivo však na obrazovke zbadáme odpoveď `„Prefíkaný kocúr býva v dome s vyšším číslom."`.

Otázka je, čo vieme z tejto odpovede zistiť. Zjavne kocúr nebýva v dome 47. To však nie je všetko. Z odpovede, ktorú sme dostali vyplýva, že nemôže bývať ani v žiadnom dome s menším číslom. Teda musí bývať v niektorom dome s číslom od 48 po 1000. Pomocou jednej otázky sme teda neodstránili len jednu možnosť, ale rovno 47 možností. A ak by sme mali šťastie a zistili by sme, že kocúr býva v dome s číslom nižším ako 47, odstránili by sme až 954 možností, lebo by nemohol bývať v žiadnom dome s väčším číslom.

Ako teda vidíme, ak ako prvý navštívime dom $x$, v ktorom kocúr nebýva, podľa odpovedi nám ubudne $x$ alebo $1001 - x$ možností. Ktoré $x$ sa teda chceme spýtať ako prvé? Ako naznačuje zadanie, na šťastie sa nemôžeme spoliehať, lebo kocúr sa schováva tam, kde by sme ho najmenej čakali. Čo v praxi znamená, že náš program vám odpovedá takým spôsobom, aby ste vždy vylúčili menej možností. Mohli by ste sa teda otázku 47 pýtať ako prvú koľkokrát chcete, odpoveď by bola stále tá istá, lebo je to menej výhodné.

To znamená, že ak sa spýtam na dom $x$, odstránim $\min(x, 1\,001 - x)$ možností, pretože takto odpovedá náš program. Boli by sme radi, keby toto číslo bolo čo najväčšie možné. A to nastane pre $x = 500$ (alebo tiež $x = 501$), teda v polovici intervalu 1 až 1 000.

A čo ďalej? Stačí opakovať túto myšlienku. Po prvom opýtaní nám zostali už len domy s číslami 501 až 1 000. Keď sa opýtam na nejaký dom $x_2$ v tomto intervale, z odpovede dokážem vylúčiť buď časť od 501 po $x_2$, alebo tú od $x_2$ po 1 000. A opäť bude najlepšie sa spýtať na stred tohto intervalu, teda $x_2 = 750$, lebo vtedy nám určite zostane najmenší počet zvyšných možností.

Môžeme si teda náš postup zovšeobecniť. Budeme si udržiavať dve čísla $zac$ a $kon$, ktoré budú udávať začiatok a koniec intervalu domov, v ktorých sa môže nachádzať kocúr. Na začiatku $zac = 1$ a $kon = 1\,000$. Následne sa v každom kole spýtam otázku $x = \frac{zac+kon}{2}$, čo je vlastne číslo v polovici intervalu $\langle zac, kon\rangle$. Ak je to číslo náhodou desatinné, zaokrúhlim ho nadol. Ak dostanem odpoveď, že kocúr býva v dome s vyšším číslom, upravím $zac = x + 1$, v opačnom prípade zmením $kon = x - 1$. Celý postup skončí, keď sa $zac = kon$. V dome s číslom $zac$ sa náš kocúr musí už určite nachádzať.

A tento postup bude fungovať pre ľubovoľne veľa domov (stačí správne nastaviť hodnotu $kon$). Princíp, ktorý sme použili sa volá **binárne vyhľadávanie** a v informatike je vcelku užitočný a používaný[1]. Dokonca sa často používa aj v reálnom živote, napríklad keď hľadáte konkrétnu stranu v knihe.

Otázkou však je, koľko otázok sa budem musieť spýtať. Uvedomme si, že v každom kroku zmenším počet možností na polovicu, pretože po opýtaní sa na čísla v strede intervalu nám jedna polovica odpadne. Teda počet otázok, ktoré musíme položiť vieme odhadnúť ako počet opakovaných delení čísla 1 000 dvojkou, kým nám nezostane 1. Čo je v tomto prípade 10. A pre 1 000 000 bude počet potrebných otázok 20. Ako vidíme, tento počet rastie pomerne pomaly. Samozrejme, budeme potrebovať ešte tak 1 alebo 2 otázky naviac, lebo interval sa nie vždy rozdelí na presnú polovicu kvôli zaokrúhľovaniu, ale na to máme dostatočnú rezervu.

### Biela

Na Bielej ulici je to trochu inak. Teraz sa v každom dome nachádza kocúr, každý je však inak prefíkaný. Naviac, najskôr prefíkanosť stúpa, až kým nedosiahne 100 a potom klesá. Otázka je, v ktorom dome býva kocúr s prefíkanosťou 100.

[1] Aj na miestach, kde by ste ho naozaj nečakali.

Aby sme sa už nemuseli rozprávať o kocúroch a prefíkanostiach, predstavme si ulicu ako postupnosť 1 000 čísel, ktoré na začiatku rastú, až kým nedosiahnu hodnotu 100 a potom klesajú až do konca. Ak viete čo je to graf funkcie, mohli by ste si túto postupnosť predstaviť ako graf funkcie, ktorý vyzerá ako kopec s jedným vrcholom. No a hľadáme práve pozíciu tohoto vrchola – pozíciu maxima.

Poučený z predchádzajúcej podúlohy sa spýtame otázku na 500-té číslo. Dostaneme odpoveď napríklad 80. Čo sme sa však dozvedeli o pozícii najväčšieho čísla (100)? Bohužiaľ nič. Hľadaná stovka sa môže stále nachádzať napravo aj naľavo od pozície 500. Z jedného čísla totiž nevieme zistiť, či naša postupnosť ešte stúpa a 100 sa nachádza až niekde ďalej, alebo už naša postupnosť klesá a 100 sa nachádza niekde predtým.

Potrebovali by sme teda zistiť, či naša postupnosť v bode 500 rastie alebo klesá. Z jedného čísla to nevieme, ale čo tak z dvoch po sebe idúcich? Spýtame sa preto na 501-vú pozíciu. Ak dostaneme číslo väčšie ako 80, vieme že postupnosť ešte len stúpa a maximum sa nachádza v druhej polovici postupnosti. Ak dostaneme číslo menšie, tak postupnosť klesá a maximum je v prvej polovici. V každom prípade, pomocou dvoch otázok sme schopný určiť, v ktorej polovici postupnoti sa nachádza číslo 100. A tento postup môžeme opakovať.

Tak ako predtým, budeme mať dve premenné $zac$ a $kon$, ktoré budú určovať interval $\langle zac, kon\rangle$, v ktorom sa môže nachádzať číslo 100. Na začiatku $zac = 1$ a $kon = 1\,000$. V každom kole sa spýtame dve otázky $x = \frac{zac+kon}{2}$ a $y = \frac{zac+kon+1}{2}$ (opäť zaokrúhľujem nadol). Ak platí, že odpoveď na otázku $y$ je väčšia ako odpoveď na otázku $x$, tak to znamená, že postupnosť na pozícii $x$ rastie, maximum sa nachádza v druhej časti, preto nastavím $zac = y + 1$. V opačnom prípade je opoveď na $y$ menšia ako odpoveď na $x$, tak na pozícii $x$ postupnosť klesá, preto nám na vyskúšanie zostáva len prvá časť a teda nastavíme $kon = x - 1$.

Tak ako v predchádzajúcom prípade aj teraz sa v každom "kole" zbavíme polovice možností. V tomto prípade nás to však bude stáť dve otázky. Preto očakávame približne 20 otázok potrebných na nájdenie odpovede. V prípade ulice s 1 000 000 domami by to teda bolo zhruba 40 otázok.

### Cesnaková

Táto ulica bola na riešenie asi najťažšia. Hlavne preto, že sa ťažko kontrolovalo, či ju riešite optimálnym spôsobom čo vyústilo v to, že sa vám mohlo podariť ju vyriešiť na 3 body aj pomocou neoptimálnej stratégie. V popisoch som si však na to dal pozor a nenechal som sa oklamať neúplnými riešeniami.

Ako teda vyzerá naše zadanie. Na začiatku sme mali postupnosť čísel, ktorá bola celú dobu rastúca. Potom sme však zobrali prvých $x$ najmenších prvkov a v tom istom poradí sme ich presunuli na jej koniec. Označme si hodnotu prvého prvku výslednej postupnosti $a$ a hodnotu posledného prvku ako $b$. Naša postupnosť teraz vyzerá tak, že niekoľko prvkov rastie začínajúc hodnotou $a$ (presnejšie po prvok na $1001 - x$-tom mieste, ale my $x$ nepoznáme, takže nám tento fakt nijak nepomôže), potom nastane prudký pád, kde nasledovný prvok je menší ako predchádzajúci a potom opäť prvky stúpajú až po hodnotu $b$. Navyše vieme, že platí $b < a$ (rozmyslite si prečo) a preto všetky prvky v prvej časti sú väčšie ako $b$ a všetky prvky v druhej časti sú menšie ako $a$.

Nebudeme otáľať a spýtame sa na 500-tý prvok. Dostaneme odpoveď napríklad 80. Čo sme sa dozvedeli o pozícii čísla 100? Opäť nič. Môže sa nachádzať aj napravo od pozície 500, ak zlom ešte nenastal alebo sme zlom minuli a nachádza sa niekde naľavo. A nepomôže nám, ani keď sa spýtame na susedné prvky. Totiž okrem jednej dvojice platí, že nasledujúci prvok je väčší. A môžete sa staviť, že tú zaujímavú dvojicu vám náš program neukáže.

Potrebujeme sa teda spýtať takú otázku, ktorá by nám prezradila, či sa maximum nachádza naľavo alebo napravo od pozície 500. Alebo, inak formulované, chceli by sme zistiť, na ktorej strane sa nachádza náš zlom. Uvedomme si, že ak by sme mali číslo $a$ (teda hodnotu prvého prvku postupnosti), vieme to zistiť naozaj jednoducho. Ak je naše zistené číslo (v našom prípade 80) menšie ako $a$, znamená to, že sa nachádza za zlomom. Ak je väčšie, tak zlom ešte nenastal. No a hodnotu $a$ zistíme otázkou na pozíciu 1.

Náš algoritmus bude teda fungovať nasledovne. Na začiatku položíme jednu otázku na pozíciu 1 a odpoveď si zapamätáme do premennej $a$. Tak ako pred tým, aj teraz budeme mať dve premenné $zac$ a $kon$, ktoré budú udávať začiatok a koniec možného intervalu. V každom kole sa spýtame otázku $x = \frac{zac+kon}{2}$ a dostaneme odpoveď $y$. Ďalší krok bude mierne zložitejší.

- Ak $a < 100$, $a < y$ a $y < 100$: nachádzame sa ešte pred zlomom, hľadaná hodnota je niekde ďalej, preto $zac = x + 1$.
- Ak $a < 100$, $a < y$ a $y > 100$: sme v prvej časti, ale hodnotu 100 sme už preskočili, preto $kon = x - 1$.
- Ak $a < 100$ a $a > y$: zlom sme už preskočili, a teda sme preskočili aj hľadanú hodnotu (keďže tá je väčšia ako $a$), preto $kon = x - 1$.
- Ak $a > 100$ a $a < y$: zlom je niekde ďalej a s ním aj od $a$ menšia hodnota 100, teda $zac = x + 1$.

- Ak $a > 100$, $a > y$ a $y < 100$: zlom sme už prešli, ale hodnota 100 sa nachádza niekde ďalej, nastavíme $zac = x + 1$.
- Ak $a > 100$, $a > y$ a $y > 100$: prešli sme zlom aj hodnotu 100, musíme hľadať niekde predtým, preto $kon = x - 1$.

Vyzerá to trochu komplikovane, ale keď sa nad tým trochu zamyslíte a nakreslíte si obrázok, zistíte, že je to veľmi priamočiare.

Ostáva nám už len odhadnúť potrebný počet otázok. Na začiatku položíme jednu otázku, aby sme zistili hodnotu $a$, ale potom to už funguje ako klasické binárne vyhľadávanie, pýtame sa do stredu intervalu a zakaždým sa rozhodneme, ktorú polovicu už nepotrebujeme. Budeme teda potrebovať zhruba 11 otázok, pre postupnosť dlhú 1 000 000 zhruba 21.

### Na záver

V tejto úlohe sme si predstavili jeden veľmi zaujímavý algoritmus – binárne vyhľadávanie. Je to veľmi užitočný nástroj, ktorý častokrát aplikujeme aj v skutočnom živote, nehovoriac už o informatike. Jeho hlavnou výhodou je malé množstvo otázok, ktoré sme sa potrebovali spýtať. Ak máme 1 000 000 prvkov, v ktorých sa snažíme nájsť jeden konkrétny, nemusíme položiť milión otázok, stačí nám ich zhruba 20. Má to však komplikáciu, ak niečo takéto chceme použiť, prvky musia byť zoradené od najmenšieho po najväčší.

V našich úlohách síce tomu nebolo vždy tak, ale ak si všimnete, vo všetkých boli prvky zoradené pomerne pekným, jednoduchým a takmer usporiadaným spôsobom. Je dobré si zapamätať túto dôležitú podmienku a pred použitím binárneho vyhľadávania si rozmyslieť, či je splnená.

## 2. Párty v časopriestore

vzorák napísal Mário
(max. 15 b za riešenie)

### Prvé riešenie, ktoré vás napadne

Pre obyvateľov najspodnejšieho a najpravejšieho domčeka je najlepšia cesta zjavná, lebo majú na výber iba jednu možnosť – ísť stále dole alebo stále doprava. Cesty vieme zapísať ako DDD…D, PPP…P, keď použijeme formát ako v podúlohe c), kde D je dole, P je vpravo.

Najlepšia cesta do druhého domčeka odspodu a do druhého sprava sa hľadá trošku ťažšie, no tiež tu nie je veľa rôznych možností.[2] Do každého z týchto dvoch domčekov sa dá dostať 9 rôznymi cestami. Napríklad do toho druhého sprava pôjdeme stále doprava a jedenkrát sa počas cesty rozhodneme zísť dole.

Pre ďalšie domčeky by sme vedeli nájsť najlepšiu cestu podobným spôsobom. Vyskúšali by sme **všetky možné cesty** a z nich by sme vybrali ku každému domčeku tú, kde získame najviac cukríkov. Ale ako to vieme spraviť? Ako vieme zabezpečiť, že žiadnu cestu nevynecháme a žiadnu nebudeme počítať dvakrát? Tu nám pomôže práve zápis cesty ako v podúlohe c).

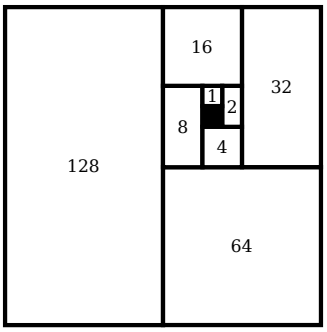
### Ako spočítať všetky cesty

- Začíname na ľavom hornom políčku, máme 42 cukríkov a prešli sme cestu dĺžky 0.
- Môžeme sa rozhodnúť ísť doprava, to budeme mať $42 + 5 = 47$ cukríkov, prejdená cesta je P.
  Môžeme sa rozhodnúť ísť dole, to budeme mať $42 + 7 = 49$ cukríkov, prejdená cesta je D.
  Podarilo sa nám spočítať všetky cesty dĺžky 1.
- Ako vieme spočítať všetky cesty dĺžky 2? Z konca každej cesty dĺžky 1 sa rozhodneme ísť vpravo alebo dole a dostaneme cesty:
  PP $47 + 6 = 53$, PD $47 + 7 = 54$, DP $49 + 7 = 56$, DD $49 + 6 = 55$
- Opakovaním by sme dostali všetky cesty dĺžky $3, 4, \ldots$

Vďaka tomu, že sú domčeky na diagonále, sú všetky rovnako ďaleko od ľavého horného rohu. Teda na to, aby sme sa dostali do ľubovoľného domčeka potrebujeme spraviť presne 9 krokov[3]. Pre vyriešenie úlohy by sme zopakovali postup 9-krát a na záver vieme zistiť, ktorá cesta vedie ku ktorému domčeku, napríklad z počtu P, D v jej zápise [4]. Samozrejme, praktickejšie by bolo si ku každej ceste písať políčko, na ktorom skončila.

[2] V porovnaní s cestami ku zvyšným domčekom.
[3] V podúlohe b) 10 krokov.
[4] Každá cesta, ktorá vedie napríklad do tretieho domčeka zhora, musí obsahovať presne 3×D.

Po každom kroku sa nám počet doterajších ciest zdvojnásobí, takže vieme, že po 9 krokoch budeme mať spočítaných $2 \cdot 2 \cdot \cdots \cdot 2 = 2^9 = 512$ ciest. Okrem toho sme doteraz museli vypočítať 1 cestu dĺžky 0, 2 cesty dĺžky 1, 4 cesty dĺžky 2 atď., teda $1 + 2 + 4 + 8 + 16 + 32 + 64 + 128 + 256 = 512 - 1$[5].




Vyzerá to ako veľa práce. Za pár hodín sa to ale dá spraviť a vyriešite hneď aj podúlohu c). Pokiaľ by ste dostali tabuľku veľkosti $80 \times 80$, na konci by ste museli spočítať $2^{79}$ ciest a ak k tomu prirátame aj všetky doterajšie kratšie cesty, $2^{80} - 1$ a trvalo by vám to $2^{70}$-krát dlhšie.

Takýto postup, hoci môže trvať dlho, je v princípe dobrý, lebo ním viete získať zaručene správne riešenie.

Ak máte funkčný postup a dokážete prinútiť počítač, aby ho spravil za vás, viete 3 hodiny počítania zmeniť na 20 minút programovania a pár sekúnd behu programu. Bohužiaľ, pri vstupe $80 \times 80$ by vám veľmi nepomohli ani dnešné počítače. Dokážu síce spraviť niekoľko miliárd operácií za sekundu, ale $2^{80} \cong 10^{24}$, teda program by stále bežal rádovo milión miliárd sekúnd.

Ak vám už chvíľku trhá kútikmi úst, že prečo to robíme tak zdĺhavo, tak je dobre. Toto jednoduché riešenie skutočne skrýva návod k optimálnemu, rýchlemu riešeniu.

### Nechceme počítať nič, čo nepotrebujeme

Možno ste si všimli, že cesty PD a DP skončia obe na rovnakom políčku, no po ich prejdení získame rôzny počet cukríkov – 54 a 56. Keby sme hľadali cesty dĺžky 3 v predošlom postupe, pokračovali by sme z oboch ciest (DP, PD) dole aj vpravo. Cestami DPD, PDD (aj DPP, PDP) sa potom dostaneme na to isté políčko a bez toho, aby sme sa pozerali na plánik vieme, že cesta DPD bude výhodnejšia ako PDD. Cesty začínajúce PD teda nikdy nebudú najvýhodnejšie, lebo rovnaké cesty začínajúce DP budú výhodnejšie. Keď vieme, že nám stačí pamätať si len **najvýhodnejšiu cestu, ktorá končí na danom políčku**, predošlý postup by sme upravili takto:

- Cesta dĺžky 0 je len jedna, získame 42 cukríkov.
- Cesty dĺžky 1 sú 2 – P,D – a končia na 2 rôznych políčkach – políčko s 5, políčko so 7
- Cesty dĺžky 2 sú 4 – PP, PD, DP, DD – ale končia len na 3 políčkach, ktoré tvoria diagonálu. Pre každé z týchto políčok vyberieme najlepšiu cestu a zostanú nám cesty: PP, DP, DD.
- Cesty dĺžky 3 budeme počítať už len $2 \times 3 = 6$. Dostávame 6 ciest, ktoré končia na 4 políčkach a teda opäť vyberieme len tie najlepšie pre každé políčko na diagonále.

Týmto spôsobom vieme postupne spočítať najlepšiu cestu pre každé políčko tabuľky. Ako „vedľajší produkt" sa dozvieme na záver aj najlepšie cesty do každého domčeka.

Po 9 krokoch vieme 10 najlepších ciest – jednu do každého domčeka. Keď pridáme 1 riadok/diagonálu mapky, ako v podúlohe b), máme z každého z 10 políčok na výber z 2 možností, kam pokračovať, teda existuje 20 ciest do 11 políčok. Vyberieme si len tú najlepšiu pre každý domček.

Na predošlý postup sa vieme pozrieť aj z iného uhla. Predpokladajme, že máme spočítané cesty dĺžky 9. Teda najlepšie cesty do koncových políčok v úlohe a). Ak chceme vyriešiť úlohu b), stačí sa pozrieť z každého nového políčka hore a doľava a vybrať si lepšiu cestu, po ktorej do neho vieme prísť. Najväčší počet cukríkov, ktorý vieme získať na políčku v riadku $r$ a v stĺpci $s$ si označíme ako $best[r][s]$ a vieme to vypočítať ako:

$$best[r][s] = \max(best[r-1][s], best[r][s-1]) + mapa[r][s]$$

[5] To, že $1 + 2 + 4 + \cdots + 2^n = 2^{n+1} - 1$ si môžete dokázať napríklad matematickou indukciou, alebo sa môžete nechať presvedčiť úžasným obrázkom.

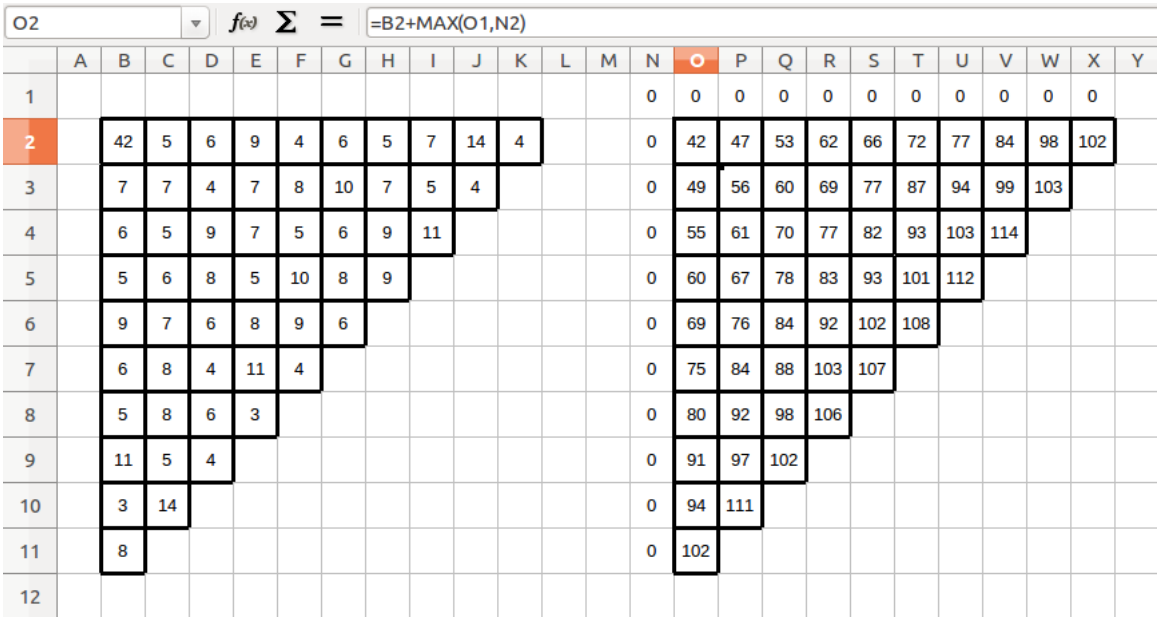
*best* z políčok mimo mapky si môžeme definovať ako 0 a potom bude sedieť, že $best[1][1] = \max(0,0) + 42$, $best[1][2] = max(0,42) + 5$, atď.

Ak viedli viaceré cesty na jedno políčko a vybrali sme si tú najlepšiu, vyberali sme si vždy z dvoch ciest – zhora a zľava. Počas výpočtu sme teda **pre každé políčko tabuľky zvažovali najviac 2 cesty.** Políčok bolo $\frac{10\times10-10}{2} + 10 = 55$[6] v a). Celkovo sme teda zvážili približne $10^2$ ciest. Vieme tak, že ak by sme dostali tabuľku $80 \times 80$, tiež sa pri počítaní najlepšej cesty pre každé políčko rozhodujeme len medzi 2 možnosťami a zvažovali by sme približne $80^2$ ciest. Výpočet takejto úlohy by nám trval približne 64-krát dlhšie.

### Máme postup, ale aj tak sme leniví a nechce sa nám sčítavať veľa čísel

Keď už máme postup upravený tak, že sa stačí pozerať na 2 políčka, hore a vľavo, môžeme úlohu vyriešiť napríklad v exceli:

- Ručne opíšeme alebo si skopírujeme z pdf zadaní tabuľku s číslami. Povedzme, že ľavý horný roh umiestnime do bunky B2.
- Tabuľku s výsledkami umiestnime napravo, s ľavým horným rohom v bunke O2. Necháme ju zatiaľ prázdnu.
- Políčka okolo druhej tabuľky vyplníme nulami: hodnoty *best* mimo tabuľky.
- Do políčka O2 vložíme rovnicu `=MAX(O1,N2)+B2`, čo zodpovedá rovnici pre best.
- Potiahnutím za pravý dolný roh O2 roztiahneme rovnice na ďalšie políčka




| | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | | | | | | | | | | | | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 2 | | 42 | 5 | 6 | 9 | 4 | 6 | 5 | 7 | 14 | 4 | | | 0 | 42 | 47 | 53 | 62 | 66 | 72 | 77 | 84 | 98 | 102 | |
| 3 | | 7 | 7 | 4 | 7 | 8 | 10 | 7 | 5 | 4 | | | | 0 | 49 | 56 | 60 | 69 | 77 | 87 | 94 | 99 | 103 | | |
| 4 | | 6 | 5 | 9 | 7 | 5 | 6 | 9 | 11 | | | | | 0 | 55 | 61 | 70 | 77 | 82 | 93 | 103 | 114 | | | |
| 5 | | 5 | 6 | 8 | 5 | 10 | 8 | 9 | | | | | | 0 | 60 | 67 | 78 | 83 | 93 | 101 | 112 | | | | |
| 6 | | 9 | 7 | 6 | 8 | 9 | 6 | | | | | | | 0 | 69 | 76 | 84 | 92 | 102 | 108 | | | | | |
| 7 | | 6 | 8 | 4 | 11 | 4 | | | | | | | | 0 | 75 | 84 | 88 | 103 | 107 | | | | | | |
| 8 | | 5 | 8 | 6 | 3 | | | | | | | | | 0 | 80 | 92 | 98 | 106 | | | | | | | |
| 9 | | 11 | 5 | 4 | | | | | | | | | | 0 | 91 | 97 | 102 | | | | | | | | |
| 10 | | 3 | 14 | | | | | | | | | | | 0 | 94 | 111 | | | | | | | | | |
| 11 | | 8 | | | | | | | | | | | | 0 | 102 | | | | | | | | | | |
| 12 | | | | | | | | | | | | | | | | | | | | | | | | | |

Voilá. Excel[7] to spočítal za nás. Najskôr počíta políčka, ktoré vie spočítať: ľavý horný roh. Potom počíta políčka, ktoré sú na nich závislé – čísla v druhom stĺpci, prvom riadku a v prvom riadku, druhom stĺpci. Potom čísla vo vzdialenosti 3. A zvyšok tabuľky podobne.

Z tejto tabuľky je aj vidno, kadiaľ sme na ktoré políčko prišli, a teda vieme zistiť cesty ku každému domčeku „odzadu".

## 3. Prekliaty hrad

vzorák napísal Maja
(max. 15 b za riešenie)

Ahojte, ako ste asi zistili, celá táto úloha sa točí okolo googlenia. A dopočula som sa, že Vás riadne potrápila :). A pritom na jej riešenie vám stačilo poznať pár príkazov pre Google vyhľadávač a zopár ďalších užitočných vecí.

[6] Zoberieme tabuľku $10 \times 10$, odpočítame diagonálu v strede, zoberieme polovicu zvyšku tabuľky a pripočítame diagonálu v strede.

[7] Calc, alebo ľubovoľný tabuľkový editor.

Počas čítania tohoto vzoráku narazíte na veľa šikovných spôsobov ako spresniť vyhľadávanie Googlu. A možno sa budete pýtať, odkiaľ ste ich mali poznať. Nuž, presne odtiaľ, dkiaľ aj my, mali ste si ich vygoogliť. Takže na začiatok úlohy si bolo treba vygoogliť ako sa správne googli. Veľa užitočných tipov a trikov nájdete napríklad tu.

Ešte drobná poznámka k tejto úlohe – ak sa vám podarilo niektorú z podúloh vyriešiť inak ako je vo vzorovom riešení, vôbec to nevadí, rovnako ak sa vám podarilo nájsť iný výsledok, ak je ten váš korektný.

a) Vašou úlohou bolo zistiť, v texte akej pesničky sa vyskytuje fráza *Hojdanie sa so*. Ak ste do Googlu zadali iba text `hojdanie sa so`, prípadne ste k tomu ešte pridali aj slová ako `text piesne` a podobne, Google vám nenašiel nič správne. A tak to aj malo byť.

   Problémom totiž je, že keď zadáte do Googlu zadáte viacero slov, hľadá všetky stránky, kde sa vyskytuje každé z týchto slov. Ale nijak neberie do úvahy to, že tieto slová sa majú nachádzať za sebou. Kľúčom k riešeniu tejto úlohy bolo teda vedieť, že frázy sa do Googlu zadávaju pomocou úvodzoviek. Google potom hľadá iba stránky, kde sa daný text vyskytuje celý pekne za sebou. Ak ste teda do Googlu zadali: `„hojdanie sa so" text piesne` ako tu, Google vypíše iba jeden relevantný výsledok a to je text piesne *Lany del Rey Video Games*, čo je obľúbená pesnička autorky tejto úlohy. Okrem toho Google vypíše už iba link na zadanie tejto úlohy :).

b) Táto úloha bola možno ešte trochu jednoduchšia ako predchádzajúca. Google v časti hľadania obrázkov ponúka aj funkciu hľadanie podľa obrázka (momentálne je dostupná tak, že na hlavnej google stránke kliknete na obrázky a potom namiesto písania do vyhľadávacieho riadku kliknete na ikonu fotoaparátu hneď vedľa). Alebo ste mohli link na obrázok napísať do vyhľadávacieho riadku a Google vás už naviedol ďalej. Každopádne na obrázku je maják, ktorý môžete nájsť v USA, v štáte Oregon.

c) Kľučovým údajom v tejto úlohe bolo, že máte prehľadávať iba stránku `ksp.sk`. A to presne viete Googlu povedať pomocou príkazu `site:`. Takže riešenie tejto úlohy malo vyzerať takto: `nemaja site:ksp.sk`. Ak sa Vám to podarilo tak ako tu, našli ste pdf súbor s programátorskými zadaniami. Veta, v ktorej sa vyskytovalo slovo Nemaja bola: *Ak sa dá postupnosť usporiadať pomocou najviac jednej výmeny, vypíšte Maja (bez úvodzoviek), inak vypíšte neMaja.*

d) Táto úloha už bola trošku ťažšia ako predchádzajúce. Poznali ste frázu, ktorú chcete vyhľadávať, ale niečo Vám v strede chýbalo a nevedeli ste, čo. A opäť, Googlu viete pomocou znaku `*` povedať, že na dané miesto môže pridať ľubovoľne veľa znakov. Nemusí tam pridať nič, ale môže. Riešenie teda malo vyzerať takto: `„pán * bol osvietený"`. A tu sa dozviete, že v knihe *Jánošík* od *Pavla Beblavého* bol osvietený pán Moravy. Hviezdička je užitočná funkcia napríklad aj keď hľadáte text nejakej pesničky, ale poznáte len určité slová, ktoré nejdú priamo za sebou.

e) Na niektoré veci Google (možno) nevie odpovedať. Ale existujú aj iné veľmi múdre stránky. Veľmi dobrý vyhľadávač odpovedí na rôzne otázky je Wolfram Alpha. Oceníte ho napríklad pri riešení matematických úloh. Dokáže kresliť funkcie a mnoho ďalšieho. Ale vráťme sa k našej úlohe. Ak sme Wolframu zadali ako otázku `weather topoľčany 3.4.1962`, naozaj nám povedal, že v Topoľčanoch bolo priemerne 8 stupňov Celzia a mnoho ďalšieho. Počula som, že viacerých z Vás zaujímalo, prečo práve Topoľčany. Dôvod je veľmi jednoduchý – nešlo to ľahko vygoogliť a narodila som sa tam :).

f) Táto úloha nebola až tak komplikovaná na vyhľadávanie, skôr išlo o pochopenie vyhľadaných informácií a pravdepodobne aj porozumenie angličtine (síce je to smutné, ale bez toho sa už veľmi na internete fungovať nedá). Ak ste zadali do googlu `#900050 color` ako tu, pravdepodobne ste sa dozvedeli, že táto farba je zadaná v 16-kovej sústave pomocou RGB modelu. RGB model funguje tak, že sa povie, ako veľa daná farba obsahuje červenej(Red), zelenej(Green) a modrej(Blue). Prvé dve číslice zodpovedajú červenej zložke farby atď. Nami zadaná farba neobsahuje žiadnu zelenú(zelená zložka je 00), takže ak ju budeme chcieť stmaviť, budeme musieť odobrať farbu z červenej a modrej, pričom treba zachovať ich pomer. Teda napríklad môžeme obidve predeliť tým istým číslom. Treba si však dať pozor, že číslo 90 a 50 sú v 16-kovej sústave. Preto ich polovica nie je 45 a 25 ako by ste si mysleli, ale 48 a 28. Správne stmavenie by preto bolo #480028. Bolo však jednoduché nájsť stránku, ktorá spraví všetky výpočty za vás, napríklad táto.

g) Toto je trošku technickejšia úloha, ktorá sa Vám ale môže v živote veľmi hodiť. Určite sa už aj vám stalo, že ste si napríklad potrebovali nejaký súbor otvoriť v mobile, ale ten nepodporoval daný formát. V tejto konkrétnej úlohe bol odporúčaný postup vygoogliť si, aké formáty kníh sa dajú čítať na Kindle Paperwhite (napr. ste mohli do Googlu zadať `Kindle Paperwhite formats`). Potom ste si chvíľku museli pogoogliť niečo o daných formátoch a napríklad ste mohli zistiť, že MOBI je vhodný formát na čítanie pre Kindle.

Tento formát zachováva aj veľkosti písma, farby a pod. Ako sa prevádza do tohto formátu si tiež viete vygoogliť, dokonca sa dajú nájsť aj nejaké vhodné online prevodníky, napríklad tento.

h) Každý počítač pripojený k internetu má svoju IP adresu. Niekedy sa o ňu delíte s inými zákazníkmi vášho poskytovateľa, vždy z nej však možno vytiahnuť užitočné informácie. Napríklad, z akej krajiny pochádzate. Toto je potom možné použiť na filtrovanie, kto má na stránku prístup. Častokrát to využívajú stránky televízií, ktoré môžu vysielať iba pre svoju krajinu. IP adresa je niečo, čo pred serverom neskryjete a teda vás podľa nej možno identifikovať. Existuje však veľmi jednoduchý spôsob, ako tento systém prekabátiť. Stačí aby si požadovanú stránku otvoril nejaký iný počítač, okopíroval ju a poslal vám ju. Takéto počítače existujú a nazývajú sa proxy servre. Webové služby potom vidia IP adresy proxy servrov a nie tie vaše. Existujú proxy servre, ktoré sú platené a zabezpečia vám anonymitu na internete ale aj také, čo sú zadarmo a prihodia vám k stránke nejakú reklamu. Keď teda do Googlu zadáte `proxy server`, nájde vám množstvo použiteľných stránok. Napríklad `https://hide.me/en/proxy`. Tam zadáte adresu, a keďže server sa nachádza mimo Slovenska, všetko funguje. Na záver ešte jeden Google trik. Google Translate si tiež stiahne stránku, preloží ju a pošle vám kópiu. Takže si stačilo našu stránku nechať preložiť.

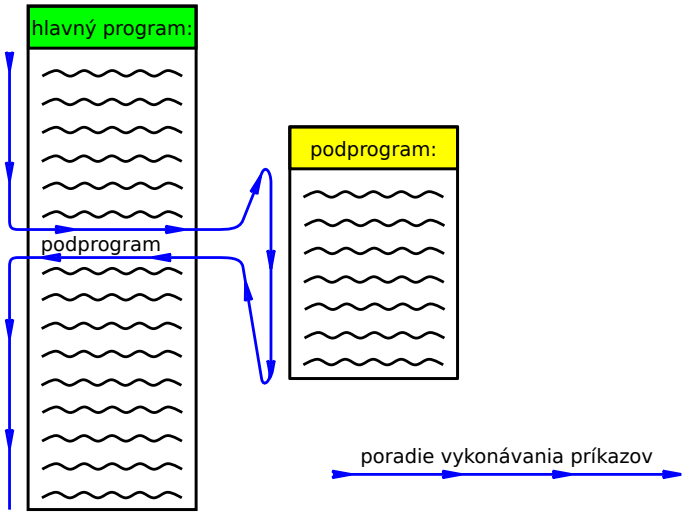
## Študijný text: O podprogramoch a rekurzii

V úlohách 4 a 5, kde ste riešili problémy robota Zerga, bolo treba používať aj komplikovanejšie koncepty ako je napríklad rekurzia. Tento študijný text slúži na bližšie vysvetlenie tohoto pojmu a toho, čo všetko sa s ňou dá robiť. Odporúčame si ho prečítať pre hlbšie pochopenie zvyšných dvoch vzorákov a takisto pojmu rekurzie.

### Čo je to podprogram?

Koncept podprogramu je jednoduchý: je to skupina príkazov, ktorej dáme nejaké meno a neskôr ju podľa tohto mena voláme.

Celé to teda vyzerá tak, že niekde definujeme podprogram a v inej časti programu ho potom zavoláme. Keď sa program dostane na toto miesto, vykoná všetky príkazy podprogramu a potom pokračuje z miesta, kde bol podprogram zavolaný. Počítač si teda (aspoň pri väčšine programovacích jazykov) vždy uchováva informáciu o tom, odkiaľ bol podprogram zavolaný, aby sa tam po jeho dokončení mohol vrátiť (pozri Obr. 1).

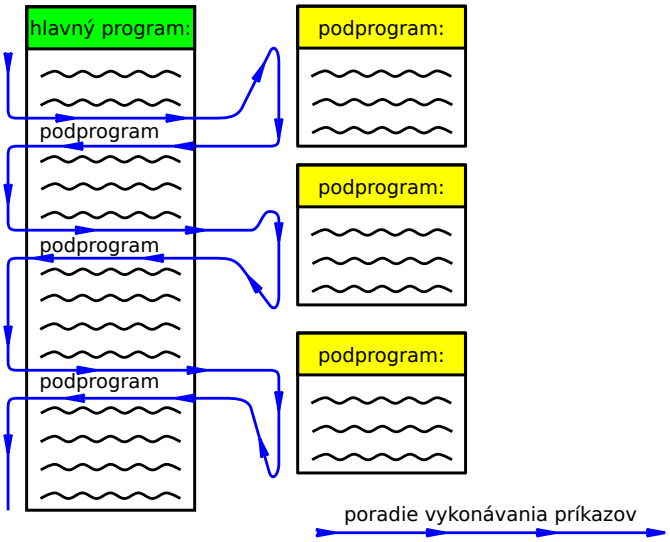



Obr. 1: Volanie podprogramu

Toto vyzerá ako úplná zbytočnosť – radšej sme mohli príkazy z podprogramu napísať na miesto, odkiaľ sme ho chceli volať. Výhodou podprogramu ale je, že ho môžeme zavolať aj viackrát, z rôznych miest programu. Ak vieme, že nejaký zložitejší úkon budeme musieť robiť na rôznych miestach programu, môžeme si na to napísať podprogram, čím si ušetríme kopec písania a znížime tým aj šancu, že sa pri písaní pomýlime (pozri Obr. 2).

### Jazykové okienko

Existujú rôzne slová označujúce podprogramy, každé má trochu iný význam:




Obr. 2: Viacnásobné volanie podprogramu

1. ***procedúra*** je najjednoduchší druh podprogramu: Je to jednoducho zhluk príkazov, ktoré sa vykonajú (dali by sa nakopírovať rovno do programu namiesto volania podprogramu a nič by sa nezmenilo).

2. ***funkcia*** je termín prebraný z matematiky. Ide o podprogram, ktorý má na základe nejakých vstupných údajov vypočítať nejaký výstup. Tento vypočítaný výsledok potom odovzdá programu, ktorý ho zavolal. V niektorých programovacích jazykoch (napr. Haskell) sa trvá na tom, aby funkcia nerobila nič iné, aby nemala žiadne vedľajšie účinky (teda sa naozaj správala ako funkcia v matematike). Iné programovacie jazyky (napr. C++) sú v tomto ohľade menej striktné a povoľujú, aby funkcia robila aj veci ovplyvňujúce zvyšok programu. V takýchto jazykoch potom nie je potrebné mať špeciálny zápis na procedúry (procedúry sú funkcie, ktoré nič nevracajú). Stačí ak funkcia vráti ľubovoľnú hodnotu (napr. 0), ktorá sa potom zvyškom programu ignoruje.

   Preto sa slovom funkcia občas myslí aj podprogram všeobecne.

3. Niekedy sa možno stretnete aj s termínom ***metóda***. Toto slovo sa používa v objektovo orientovanom programovaní a neznamená nič iné ako funkciu alebo procedúru, ktorá existuje v kontexte nejakého objektu (ak ste tomuto vôbec nerozumeli, nevadí, zatiaľ to ani nepotrebujete).

V tomto texte budem ďalej namiesto slova ***podprogram*** používať slovo ***funkcia***. A postupnosti príkazov, ktoré tvoria funkciu budeme hovoriť ***telo*** funkcie.

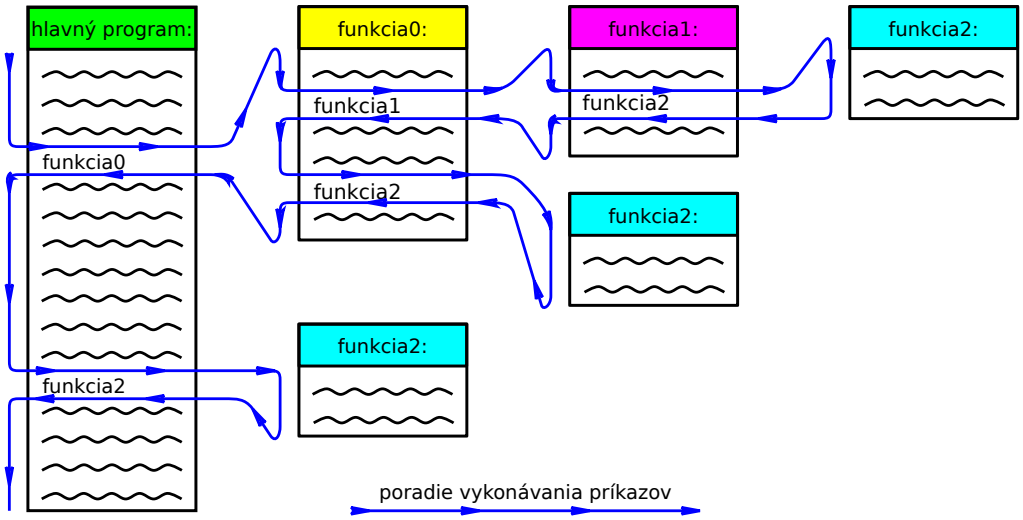
### Volanie funkcie vo funkcii

Volanie funkcie nie je obmedzené iba na hlavný program. Vnútri tela funkcie môžeme volať iné funkcie. Vnútri iných funkcií potom môžeme volať ďalšie atď. Vyzerá to asi ako na obrázku 3.

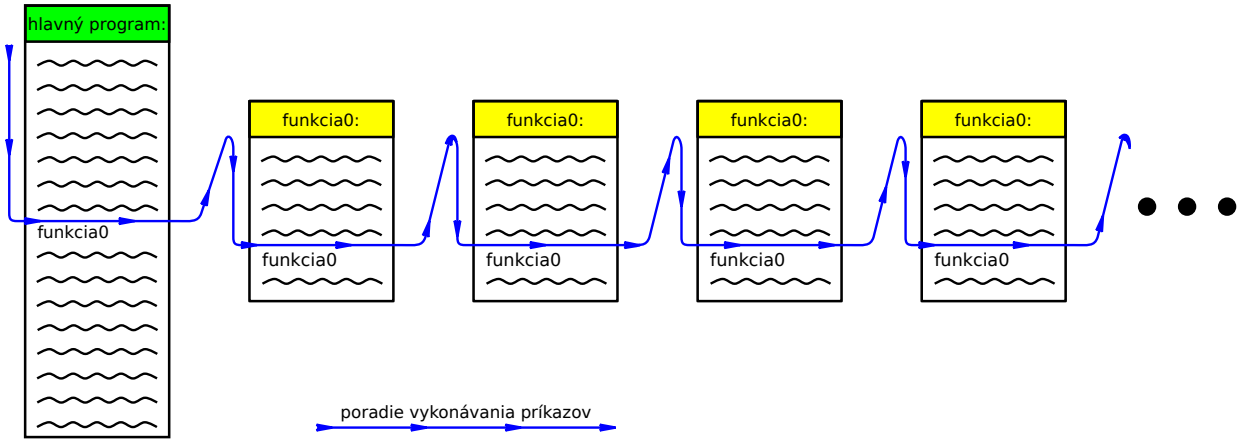
Pri každom volaní funkcie si počítač zapamätá, odkiaľ bola zavolaná. Túto informáciu si uchováva, až kým sa nevykoná celé telo danej funkcie. Vtedy program túto informáciu využije (aby vedel, kde má pokračovať) a následne ju už môže zabudnúť. To znamená, že keď vnútri jednej funkcie zavoláme druhú funkciu, počítač si musí zároveň pamätať, odkiaľ bola zavolaná prvá funkcia, aj odkiaľ bola zavolaná druhá funkcia. Pre príklad na obrázku to znamená, že keď sa `funkcia2` vykonávala prvýkrát, počítač si musel pamätať miesta, z ktorých boli zavolané `funkcia2`, `funkcia1`, aj `funkcia0`. Keď bola `funkcia2` volaná druhý raz, už si nemusel pamätať, odkiaľ bola volaná `funkcia1`, keďže táto funkcia medzičasom skončila. Keď bola `funkcia2` zavolaná tretí raz, už si nemusel pamätať ani to, odkiaľ bola volaná `funkcia0`.

### Nekonečná rekurzia

Čo by sa stalo, keby sme vnútri tela funkcie zavolali tú istú funkciu? Po prvom zavolaní funkcie v hlavnom programe by sa začali vykonávať jej príkazy, až pokým by funkcia nezavolala sama seba. V tom okamihu by sa




Obr. 3: Volanie funkcie vo funkcii




Obr. 4: Obr. 4: Nekonečná rekurzia

znovu od začiatku začali vykonávať príkazy danej funkcie, až kým by znovu nezavolala sama seba. A takto by to išlo ďalej a ďalej (pozri Obr. 4).

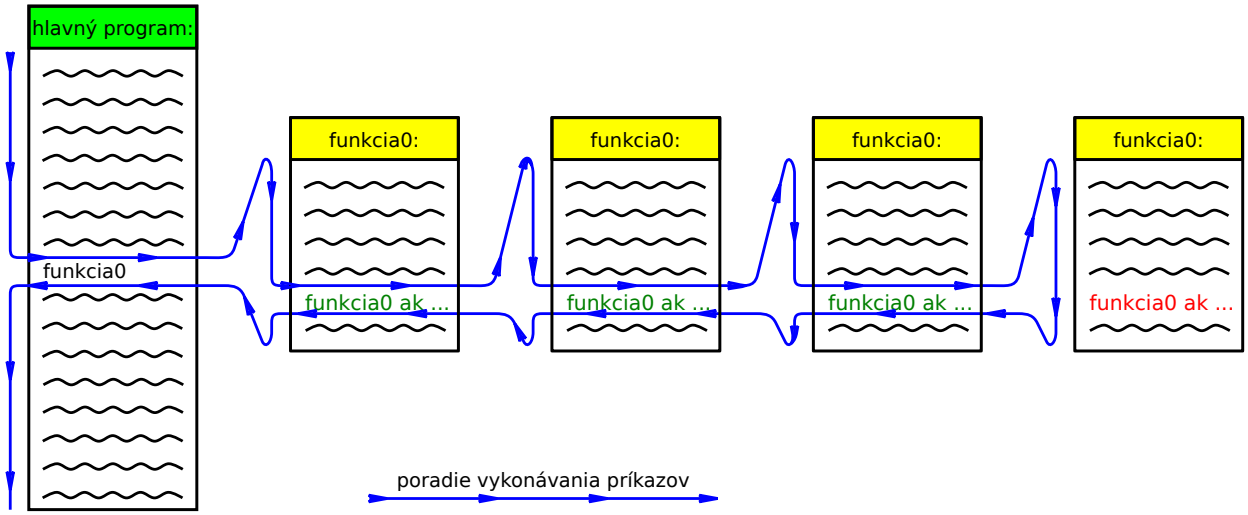
Technika, kde funkcia volá sama seba, sa nazýva ***rekurzia***. V praktickom programovaní vznikajú pri nekonečnej rekurzii isté technické problémy. Konkrétne, pri každom zavolaní funkcie si počítač musí zapamätať, odkiaľ bola zavolaná, pričom túto informáciu môže zabudnúť, až keď sa funkcia úplne vykoná (čo nenastane nikdy). To znamená, že po nejakom počte zavolaní sa mu zaplní celá časť pamäte, ktorú mal na tento účel vyhradenú. Preto je v mnohých jazykoch nepoužiteľná.

### Chvostová rekurzia

Rekurzia však nemusí byť nutne nekonečná. Ak funkcia vo svojom tele volá sama seba **len ak sú splnené určité podmienky**, pričom tieto podmienky po nejakom počte zavolaní prestanú platiť, funkcia sa zavolá iba konečný počet krát. Po poslednom zavolaní funkcie sa funkcia vykoná (bez toho, aby sa zavolala znovu), následne sa dokončí vykonávanie predošlého volania (vykonajú sa príkazy nachádzajúce sa za volaním funkcie), potom sa dokončí predošlé volanie atď., až kým sa nedostaneme k miestu, kde bola funkcia zavolaná na začiatku (pozri Obr. 5).

### Všeobecná rekurzia

Nie sme obmedzení na to, aby sa funkcia vo svojom tele volala iba raz, pokojne sa môže zavolať aj viackrát.




Obr. 5: Obr. 5: Chvostová rekurzia

Alebo môže volať nejakú inú funkciu, ktorá potom naspäť zavolá ju. Kreativite sa medze nekladú.

## 4. Zerg Bot 3

vzorák napísal Baklažán
(max. 0 b za riešenie)

Tento vzorák bude používať mnohé pojmy a techniky vysvetlené vo vzoráku k predošlej sérii, preto pred jeho čítaním odporúčam, aby ste si prečítali vzorové riešenia k úlohám 4 a 5 z predošlej série Prasku, resp. k úlohe Z1 z predošlej série KSP (ak ste ich náhodou ešte nečítali).

### 1 - Tam a späť

Na prejdenie levelu stačí, aby robot chodil stále rovno a vždy, keď príde na rozsvietený prepínač, prepínač vypol a otočil sa. Napíšeme si teda funkciu, ktorá urobí jeden krok a v prípade potreby vypne prepínač a otočí robota. Túto funkciu potom robota necháme vykonávať dookola v nekonečnej rekurzii.

Narazíme však na drobný problém. Robot počas svojho pohybu chodí aj po už vypnutých prepínačoch, preto ak po vstupe na rozsvietený prepínač najprv prepínač vypneme, robot nemá ako zistiť, že sa má otáčať. To môžeme vyriešiť napríklad tak, že si napíšeme pomocnú funkciu, ktorá prepne prepínač a otočí robota. Túto funkciu potom zavoláme, keď vstúpime na rozsvietený prepínač. Tiež by fungovalo, keby sme robota po vstupe na rozsvietený prepínač najskôr otočili a až potom prepínač vypli.

Kód:

```
funkcia0

aha funkcia0:
krok
funkcia1 ak svieti
funkcia0

aha funkcia1:
prepni
dolava
dolava
```

### 2 - Tam, kde sa skrýval bazilisk

V tomto leveli sa najprv bolo treba trochu pohrať s prepínačmi a zistiť, ako fungujú. Keďže väčšina chodbičiek má dĺžku 3, určite sa nám zíde funkcia, ktorá urobí 3 kroky:

```
aha funkcia0:
```

```
krok
krok
krok
```

Po troche experimentovania nájdeme spôsob, ako level prejsť. Jednou z možností je obísť celú komnatu v smere hodinových ručičiek začnúc v juhovýchodnom rohu a cestou rozsvietiť všetky prepínače okrem prepínača v severovýchodnom rohu. Môžeme si všimnúť, že tri steny komnaty vyzerajú úplne rovnako, preto sa nám oplatí napísať si funkciu, ktorá začínajúc v rohu prepne prepínač v danom rohu, prejde popri jednej stene (cestou prepne aj prepínač "vo výklenku") a skončí v ďalšom rohu. Aby sme túto funkciu mohli volať bezprostredne viackrát po sebe, v koncovom rohu sa ešte otočíme doprava:

```
aha funkcia1:
prepni
funkcia0
dolava
krok
prepni
dolava
dolava
krok
dolava
funkcia0
doprava
```

S pomocou tejto funkcie potom ľahko napíšeme celý program:

Hlavný kód:

```
funkcia0
doprava
funkcia0
doprava
funkcia1
funkcia1
funkcia1
funkcia0
dolava
funkcia0
krok
```

### 3 - Komprimácia

Najprv si spočítame, že rovné úseky majú postupne dĺžky 9, 4, 7, 13, 4, 6, 15, 6 a 12 krokov. Keďže medzi týmito dĺžkami je dosť veľa násobkov trojky, zíde sa nám funkcia, ktorá urobí 3 kroky. Tiež sa nám tam dvakrát opakuje štvorka, preto si definujeme aj funkciu na urobenie 4 krokov (samozrejme s využitím funkcie pre 3 kroky). S pomocou týchto dvoch funkcií budeme vedieť urobiť 7 krokov na dva príkazy. Keďže máme dva úseky dlhé 6, oplatí sa nám napísať si aj funkciu na 6 krokov. Nakoniec sa nám zíde aj funkcia na 9 krokov – nielenže máme úsek dlhý 9, ale skombinovaním deväť- a štvorkrokovej funkcie vieme na dva príkazy urobiť 13 krokov a skombinovaním deväť- a šesťkrokovej vieme urobiť 15 krokov. Pri definícii každej z týchto funkcií môžeme, samozrejme, využívať tie predchádzajúce. Naše funkcie teda vyzerajú takto:

```
aha funkcia3:
krok
krok
krok

aha funkcia4:
funkcia3
krok

aha funkcia6:
```

```
funkcia3
funkcia3

aha funkcia9:
funkcia6
funkcia3
```

S pomocou takýchto funkcií vieme ľahko napísať program, ktorý prejde level a spolu s definíciami funkcií má presne 30 príkazov, teda sa akurát zmestí do limitu. Rovné úseky z našich funkcií poskladáme ako 9, 4, 4+3, 9+4, 4, 6, $9+6$, 6 a $9+3$. Toto riešenie sa však dá ešte zlepšiť. Môžeme si totiž všimnúť, že po všetkých úsekoch, kde sme použili `funkciu4`, nasledovala zákruta doprava. Preto môžeme príkaz `doprava` pridať na koniec `funkcie4` a túto funkciu vždy volať až na konci úseku. Podobne po každom úseku obsahujúcom `funkciu6` nasleduje zákruta doľava, preto môžeme príkaz `dolava` pridať na koniec `funkcie6`. Tým si ale pokazíme `funkciu9` (keďže tá vo svojej definícii používa `funkciu6`), preto ju musíme prepísať, napríklad takto:

```
aha funkcia9:
funkcia3
funkcia3
funkcia3
```

S pomocou takýchto funkcií potom vieme napísať program, ktorý má spolu s definíciami iba 26 príkazov:

Hlavný kód:

```
funkcia9
dolava

funkcia4

funkcia3
funkcia4

funkcia9
funkcia4

funkcia4

funkcia6

funkcia9
funkcia6

funkcia6

funkcia9
funkcia3
```

### 4 - Svetielka

Potrebujeme rozsvietiť 5 prepínačov na východnom konci chodby a zhasnúť 5 prepínačov na západnom konci chodby. Celé to môžeme urobiť napríklad takto:

1. Najprv prepneme prvých 5 prepínačov.
2. Potom pôjdeme rovno, až kým nenarazíme na stenu, pričom cestou budeme vypínať všetky svietiace prepínače.
3. Pri stene sa otočíme.
4. Pôjdeme až do konca levelu iba rovno.

Prvý bod urobíme veľmi jednoducho: napíšeme si funkciu, ktorá prepne prepínač a urobí krok, túto funkciu potom zavoláme päťkrát.

```
aha funkcia0:
prepni
krok
```

Na druhú časť si napíšeme takúto funkciu:

```
aha funkcia1:
krok
prepni ak svieti
funkcia1 ak nie je stena vpredu
```

Kým pred robotom nebude stena, táto funkcia sa bude rekurzívne volať dookola. Po príchode k stene sa volať prestane a všetky jej zavolania postupne skončia.

Tretia časť je jednoduchá a na štvrtú časť nám stačí funkcia, ktorá bude v nekonečnej rekurzii robiť kroky:

```
aha funkcia2:
krok
funkcia2
```

Hlavný program teda bude vyzerať nasledovne:

Hlavný kód:

```
funkcia0
funkcia0
funkcia0
funkcia0
funkcia0
funkcia1
dolava
dolava
funkcia2
```

### 5 - Tajná kombinácia

Keďže o tajnej kombinácii nič nevieme, nezostáva nám nič iné, ako začať postupne skúšať všetky možné kombinácie. Nemusíme to však skúšať ručne. Namiesto toho si môžeme napísať program, ktorý na hracom pláne postupne vytvorí všetky možné kombinácie, tento program spustiť a potom sa už len pozerať, pri ktorej kombinácii sa jet torch vypne.

Napíšme si `funkciu0`, ktorá za predpokladu, že je robot otočený smerom na východ, vytvorí na prepínačoch, ktoré sú medzi robotom a východným koncom chodby (vrátane prepínača, na ktorom robotom práve stojí) postupne všetky možné kombinácie, pričom na konci nechá vypínače v rovnakom stave ako boli na začiatku a robot bude stáť na rovnakom políčku ako na začiatku, ale otočený na západ. Tento popis sa vám môže zdať trochu krkolomný, ale neskôr sa nám zíde. Pri programovaní tejto funkcie využijeme princíp vysvetľovania významu slova pomocou toho istého slova.

Ako by takáto funkcia mala vyzerať?

- V prípade, že je robot na začiatku volania funkcie na východnom konci chodby, je to jednoduché. Máme vytvoriť všetky možné kombinácie na prepínači, na ktorom robot práve stojí. Takéto kombinácie sú iba dve – buď je prepínač vypnutý alebo je zapnutý. V jednom z týchto stavov je na začiatku a do druhého ho jednoducho prepneme. Chceme, aby robot skončil otočený smerom na západ a aby prepínače boli v pôvodnom stave, preto robota ešte otočíme a prepínač prepneme naspäť.

  ```
  prepni
  dolava
  dolava
  prepni
  ```

- Prípad, keď robot na začiatku volania funkcie nie je na východnom konci chodby, je trochu zložitejší, keďže sa musíme postarať o väčší úsek prepínačov a teda musíme vytvoriť viac kombinácií. Počet prepínačov navyše dopredu nevieme. Všetky kombinácie však môžeme rozdeliť na dve skupiny: tie, pri ktorých je

prepínač, na ktorom robot práve stojí, zhasnutý a tie, pri ktorých je rozsvietený. Najprv vytvoríme všetky kombinácie z prvej skupiny: posunieme robota o jeden krok na východ a zavoláme `funkciu0`. Tým sa na všetkých prepínačoch východne od začiatočného vytvoria všetky možné kombinácie, teda sme vytvorili všetky kombinácie, pri ktorých je začiatočný prepínač vypnutý. Následne sa vrátime na políčko, kde sme začínali, rozsvietime ho a podobným spôsobom vytvoríme všetky kombinácie, pri ktorých je toto políčko rozsvietené. Nakoniec sa vrátime na začiatočné políčko a opäť ho zhasneme, aby boli na konci volania funkcie políčka v pôvodnom stave.

```
krok
funkcia0
krok
prepni
dolava
dolava
krok
funkcia0
krok
prepni
```

Ešte nejako potrebujeme docieliť, aby sa vždy vykonala tá správna z týchto dvoch možností. To vieme urobiť napríklad tak, že každú z týchto možností dáme do inej pomocnej funkcie a vo `funkcii0` sa iba rozhodneme, ktorú z týchto dvoch funkcií zavoláme. Pritom si, samozrejme, musíme dať pozor, aby sa nám nestalo, že po zavolaní prvej (a s tým súvisiacej zmene situácie) sa zavolá aj druhá.

Kód:

```
funkcia0

aha funkcia0:
funkcia1 ak je stena vpredu
funkcia2 ak je stena vlavo

aha funkcia1:
prepni
dolava
dolava
prepni

aha funkcia2:
krok
funkcia0
krok
prepni
dolava
dolava
krok
funkcia0
krok
prepni
```

Po tom, čo nájdeme správnu kombináciu, je už riešenie jednoduché:

Kód:

```
krok
prepni
krok
prepni
krok
krok
prepni
```

```
krok
prepni
krok
doprava
krok
krok
```

vzorák napísal Baklažán
(max. 0 b za riešenie)

## 5. Zerg Bot 5

Tento vzorák bude používať mnohé pojmy a techniky vysvetlené vo vzoráku k predošlej sérii, preto pred jeho čítaním odporúčam prečítať si vzorové riešenia k úlohám 4 a 5 z predošlej série Prasku, resp. k úlohe Z1 z predošlej série KSP (ak ste ich náhodou ešte nečítali).

### 1 - Svietiaca cesta

Napíšeme si funkciu, ktorá urobí krok správnym smerom a potom zavolá sama seba. Ako urobiť krok správnym smerom? Jednoducho vyskúšame všetky tri možné smery. Najprv urobíme krok dopredu a skontrolujeme, či sme na svietiacom prepínači. Ak áno, potom to bol krok správnym smerom a môžeme znovu zavolať našu funkciu. V opačnom prípade sa vrátime naspäť a skúsime nejaký iný smer, napríklad doľava (vzhľadom na smer, ktorým sme boli otočení na začiatku volania funkcie). Ak ani tým smerom nebude svietiaci prepínač, vrátime sa a pôjdeme posledným možným smerom, teda doprava. Keďže používame nekonečnú rekurziu, funkcia sa vždy vykoná iba po miesto, kde prvý raz zavolá sama seba, teda sa nám nemôže stať, že by po prvom zavolaní urobila ešte nejaké nežiadúce príkazy navyše. Keďže sa na viacerých miestach programu budeme chcieť vrátiť na políčko, odkiaľ sme prišli, môžeme si na to napísať pomocnú funkciu.

Kód:

```
funkcia0

aha funkcia0:
krok
funkcia0 ak svieti
funkcia1
doprava
krok
funkcia0 ak svieti
funkcia1
krok
funkcia0

aha funkcia1:
dolava
dolava
krok
```

### 2 - Rozsvecovanie

Najjednoduchší spôsob ako zaručene nájsť všetky stĺpy, je prehľadať celú mapu. Skúsme teda najprv napísať program, ktorý by po riadkoch prešiel po celom plániku (podobne ako v leveli 4-Šachovnica z úlohy Prask 1.4 z predošlej série), ak by tam neboli stĺpy. Robotovu cestu môžeme rozdeliť na 4 druhy úsekov: úseky, keď ide na východ, úseky, keď ide na západ, otáčanie pri východnom okraji miestnosti a otáčanie pri západnom okraji miestnosti. Pre každý z týchto druhov si napíšeme jednu funkciu a tieto štyri funkcie vhodným spôsobom necháme volať sa navzájom v nekonečnej rekurzii:

```
funkcia0

aha funkcia0:
krok
funkcia0 ak nie je stena vpredu
funkcia2
```

```
aha funkcia1:
krok
funkcia1 ak nie je stena vpredu
funkcia3

aha funkcia2:
doprava
krok
doprava
funkcia1

aha funkcia3:
dolava
krok
dolava
funkcia0
```

`funkcia0` slúži na cestu na východ: táto funkcia bude hýbať robotom dopredu a volať sama seba dookola, až kým robot nepríde k stene. Potom zavolá `funkciu2`, ktorá robota otočí (a posunie do ďalšieho riadku) a následne zavolá `funkciu1`, ktorá slúži na cestu na západ. `funkcia1` podobným spôsobom dovedie robota k západnému okraju miestnosti a zavolá `funkciu3`, ktorá robota otočí (a posunie o jedno políčko na juh) a zavolá opäť `funkciu0`. Takto sa budú tieto štyri funkcie volať dookola, až kým robot nepríde na spodok mapy.

To je síce všetko pekné, ale na mape sú aj stĺpy, okolo ktorých potrebujeme rozsvietiť prepínače. Najprv, samozrejme, budeme potrebovať vedieť rozoznať stĺpy od okrajov miestnosti. Na to môžeme využiť, že stĺpy sa, na rozdiel od okrajov miestnosti, dajú obísť. Napíšme si teda funkciu, ktorá obíde jeden stĺp:

```
aha funkcia4:
doprava
krok
dolava
krok
krok
dolava
krok
doprava
```

Všimnime si, že ak by sme túto funkciu zavolali, keď je tesne pred robotom okraj miestnosti, robot by sa počas jej vykonávania chvíľu pokúšal ísť do steny a nakoniec by skončil tam, kde začal, otočený rovnakým smerom, ako na začiatku (okrem niektorých prípadov, keď robot začínal v rohu miestnosti).

`funkciu0` teraz môžeme upraviť nasledovne:

1. Kým pred robotom nie je stena, budeme ním hýbať dopredu.
2. Keď robot príde k stene, pokúsime sa ju obísť (zavoláme `funkciu4`). Ak to bol stĺp, potom sa nám ho podarilo obísť a teraz už pred robotom nie je stena (tu využívame, že žiaden stĺp nemá dve políčka na východ od seba ďalší stĺp, ani okraj mapy. Ak to bol okraj miestnosti, potom sa nám ho nepodarilo obísť, teda je pred robotom stále stena.
   - Ak je teda pred robotom stena, znamená to, že sme pri východnom okraji mapy, teda môžeme zavolať `funkciu2`.
   - Ak pred robotom nie je stena, znamená to, že sme práve obišli stĺp. Rozsvietime teda štyri políčka okolo tohto stĺpu a pokračujeme ďalej smerom na východ (znovu zavoláme `funkciu0`).

Obdobným spôsobom upravíme aj `funkciu1`. Na rozsvecovanie štyroch políčok okolo stĺpu, pri ktorom stojíme, si môžeme pre lepšiu prehľadnosť kódu napísať pomocnú funkciu. Tú môžeme rozpísať príkaz po príkaze alebo môžeme použiť stručnejšiu verziu využívajúcu chvostovú rekurziu.

```
aha funkcia0:
krok
funkcia0 ak nie je stena vpredu
```

```
funkcia4
funkcia2 ak je stena vpredu
dolava
funkcia5
doprava
funkcia0

aha funkcia1:
krok
funkcia1 ak nie je stena vpredu
funkcia4
funkcia3 ak je stena vpredu
dolava
funkcia5
doprava
funkcia1

aha funkcia5:
prepni
krok
dolava
krok
funkcia5 ak nesvieti
```

V hlavnom programe nám potom stačí iba zavolať `funkciu0`.

### 3 - XOR

Hlavným problémom pri riešení tohto levelu je, že na určenie hodnoty jedného políčka si potrebujeme zapamätať hodnoty z až dvoch iných políčok. Robot si pritom dokáže pamätať iba to, aké funkcie sú práve zavolané (a v akom poradí) a pokiaľ už boli vykonané. A presne to využijeme – informáciu o políčkach budeme kódovať do toho, akú funkciu robot práve vykonáva.

Podobne ako pri predošlom leveli, aj teraz si napíšeme niekoľko funkcií, ktoré sa budú navzájom volať v nekonečnej rekurzii.

Na začiatok budeme chcieť funkciu, ktorá, keď je robot v spodnom riadku, vyvedie ho do horného riadku a zavolá vhodnú nasledujúcu funkciu v závislosti od toho, či políčko v hornom riadku svieti:

```
aha funkcia0:
dolava
krok
krok
doprava
doprava
funkcia1 ak svieti
funkcia2
```

Táto funkcia predpokladá, že robot je na začiatku otočený smerom na východ a na konci ho otočí smerom na juh. Potom zavolá buď `funkciu1`, alebo `funkciu2`. Nikdy sa nestane, že by sa z jedného volania `funkcie0` najprv zavolala `funkcia1` a potom aj `funkcia2`, pretože používame nekonečnú rekurziu a teda `funkcia1` (ak ju zavoláme) nikdy neskončí.

`Funkciu1` teda voláme v situácii, keď je robot v hornom riadku a stojí na svietiacom políčku. V takom prípade potrebujeme zísť do stredného riadku a zavolať vhodnú nasledujúcu funkciu:

```
aha funkcia1:
krok
funkcia3 ak svieti
funkcia4
```

`funkciu3` voláme v situácii, keď sme v strednom riadku a obe políčka svietia. V takom prípade políčko v spodnom riadku rozsvietiť netreba, teda sa môžeme presunúť do ďalšieho stĺpca a tam znovu zavolať `funkciu0`. `Funkcia4` sa má správať podobne, s tým rozdielom, že políčko v spodnom riadku rozsvietiť treba.

```
aha funkcia3:
krok
dolava
krok
funkcia0

aha funkcia4:
krok
prepni
dolava
krok
funkcia0
```

`Funkciu2` by sme mohli napísať rovnako, ako `funkciu1`, s tým rozdielom, že by volala nejakú `funkciu5`, ak políčko v strednom riadku svieti a `funkciu6`, ak nesvieti. `Funkcia5` by ale potom vyzerala úplne rovnako ako `funkcia4` a `funkcia6` by vyzerala rovnako ako `funkcia3`, preto môžeme rovno využiť tieto dve funkcie:

```
aha funkcia2:
krok
funkcia4 ak svieti
funkcia3
```

V hlavnom programe nám potom stačí zavolať `funkciu0`.

### 4 - Bludisko s bicyklom

V tomto labyrinte nám, na rozdiel od toho minulého (level 4-Bludisko z úlohy Prask 1.5 z predošlej série), pravidlo pravej ruky stačiť nebude. Namiesto toho použijeme iný algoritmus s názvom *prehľadávanie do hĺbky* (často sa označuje aj skratkou *DFS* odvodenou z anglického názvu *depth-first search*). Tento algoritmus robí presne to, čo potrebujeme: rozsvieti súvislú oblasť nerozsvietených políčok, na ktorej robot práve stojí, pričom na konci nám robota nechá na rovnakom políčku, ako bol na začiatku. Ako to celé funguje? Znovu použijeme princíp vysvetľovania významu slova pomocou toho istého slova:

Ako rozsvietiť súvislú oblasť nerozsvietených políčok, na ktorej práve stojíme (a skončiť pri tom na rovnakom políčku, na akom začíname):

1. Najprv rozsvietime políčko, na ktorom práve stojíme.
2. Ak je pred nami nerozsvietené políčko, vojdeme naň a rozsvietime súvislú oblasť nerozsvietených políčok obsahujúcu toto políčko. Potom sa vrátime naspäť (a otočíme sa rovnakým smerom, ako na začiatku).
3. Ak je teraz naľavo od nás nerozsvietené políčko, vojdeme aj na toto a tiež rozsvietime súvislú oblasť obsahujúcu toto políčko. Potom sa opäť vrátime.
4. Podobne, ak je teraz napravo od nás nerozsvietené políčko.
5. Nakoniec by sme sa ešte mali pozrieť dozadu. Ak však vieme, že odtiaľ sme prišli a teda tam určite nie je nerozsvietený prepínač, tak sa tam pozerať nemusíme.

Pri programovaní tohto algoritmu si treba dať pozor na dve veci:

1. Pri kontrolovaní susedných políčok budeme najprv kontrolovať, či je na danom políčku stena. V prípade, že nie je, budeme potrebovať vykonať sériu príkazov (ísť na dané políčko, skontrolovať, či svieti, atď.). Nemôžeme však každý z týchto príkazov podmieniť tým, či je pred robotom stena, keďže to sa môže už vykonaním prvého z nich zmeniť. Namiesto toho si na celú túto sériu napíšeme pomocnú funkciu, ktorú v prípade potreby zavoláme (a teda sa zaručene vykonajú všetky potrebné príkazy).

2. Musíme si dať pozor, aby sme sa nestratili v tom, ktorým smerom je robot práve otočený. Najjednoduchšie preto bude napísať obe funkcie tak, aby robota vždy vrátili otočeného rovnako, ako ho dostali.

Kód:

```
krok
funkcia0
dolava
```

```
dolava
krok
doprava
krok
krok

aha funkcia0:
prepni
funkcia1 ak nie je stena vpredu
dolava
funkcia1 ak nie je stena vpredu
dolava
dolava
funkcia1 ak nie je stena vpredu
dolava

aha funkcia1:
krok
funkcia0 ak nesvieti
dolava
dolava
krok
dolava
dolava
```

### Dobre, ale prečo to celé funguje?

**Nemôže sa stať, že sa funkcie budú volať donekonečna?** Nemôže. Všimnime si totiž, že `funkciu0` voláme vždy iba na nerozsvietených políčkach, pričom hneď na jej začiatku dané políčko rozsvietime. Políčka nikdy nezhasíname, teda `funkciu0` zavoláme najviac toľkokrát, koľko je na začiatku nerozsvietených políčok. `Funkciu1` voláme iba z `funkcie0`, najviac trikrát za jedno volanie `funkcie0`, teda ju tiež zavoláme iba konečne veľa krát.

**Nemôže sa stať, že by sme nejaké políčko nerozsvietili?** Nemôže. Predstavme si, že by sa niečo také stalo, teda že by sme nejaké políčko $X$ nerozsvietili. Vieme, že políčko, na ktorom sme prvý raz zavolali `funkciu0` (označme ho $S$), sme určite rozsvietili. Predstavme si, že by sme išli z $S$ do $X$. Začali sme na svietiacom políčku a skončili sme na nerozsvietenom, teda niekedy cestou sme určite prešli z rozsvieteného políčka na nerozsvietené. Toto svietiace políčko označme $A$ a susedné nerozsvietené označme $B$. Keďže $A$ svieti, určite sme niekedy zavolali `funkciu0` stojac na $A$ (lebo mimo `funkcie0` sme nič nerozsvecovali). Počas tohto zavolania sa však robot musel pozrieť aj na políčko $B$ (ak z neho priamo neprišiel), teda ho určite buď rozsvietil, alebo bolo rozsvietené už predtým. Dostali sme teda nezmysel (políčko $B$ zároveň svieti aj nesvieti), čiže sa nemôže stať, že by sme nejaké políčko zabudli rozsvietiť.

### 5 - Ďalšie jet torchové peklo

Vidíme, že budeme potrebovať vypnúť prepínač v severozápadnom rohu. Na to budeme potrebovať povypínať všetky prepínače v strednom rade. Tým si ale pokazíme jet torche v spodnom rade, preto po tom, čo vypneme prepínač v severozápadnom rohu, budeme musieť prepínače v strednom rade uviesť do pôvodného stavu. Potrebujeme si teda nejako zapamätať, v akom stave tieto prepínač boli. Táto informácia je príliš zložitá na to, aby sme ju mohli zakódovať iba do toho, ktorá funkcia bola zavolaná ako posledná. Môžeme však využiť, že robot si pamätá *všetky* funkcie, ktoré boli zavolané a ešte neskončili. Napíšeme si teda dve funkcie, ktoré sa budú navzájom volať – jednu budeme volať na svietiacich prepínačoch a druhú na zhasnutých. Nepoužijeme však nekonečnú rekurziu, ale chvostovú a pri vynáraní sa z nej budeme prepínače uvádzať do pôvodného stavu.

Ako to bude vyzerať v praxi? Opäť využijeme, že môžeme vysvetľovať slovo pomocou toho istého slova. Od našich dvoch funkcií budeme chcieť, aby sa správali takto:

`funkcia0`: Za predpokladu, že robot stojí na zhasnutom prepínači $X$ v strednom rade a pozerá sa na východ, táto funkcia najprv pozhasína všetky políčka od $X$ na východ, potom pôjde zhasnúť políčko v severozápadnom rohu a následne robota vráti na políčko $X$, pričom všetky vypínače od $X$ na východ vráti do pôvodného stavu. Na konci nechá robota otočeného na západ.

`funkcia1`: Bude robiť v podstate to isté, čo `funkcia0`, s tým rozdielom, že `funkciu1` budeme volať, keď je robot na svietiacom políčku.

A naprogramujeme ich takto:
`funkcia0`: Najprv urobíme krok dopredu. Potom môžu nastať tri prípady:

1. Ak sme už na východnom konci stredného radu (teda pred nami je stena), potom pôjdeme vypnúť vypínač severozápadnom rohu a vrátime sa naspäť (na toto si napíšeme pomocnú ‘funkciu2‘. Tú napíšeme tak, ab po jej skončení robot smeroval na západ).
2. Ak sme na zhasnutom prepínači, môžeme zavolať ‘funkciu0‘. Tá za nás urobí skoro všetkú prácu.
3. Ak sme na rozsvietenom prepínači, môžeme zavolať ‘funkciu1‘.

Nakoniec musíme vo všetkých troch prípadoch urobiť ešte jeden krok, aby sme skončili na rovnakom políčk na akom sme začínali.

Keďže už nepoužívame nekonečnú rekurziu, musíme si dať pozor, aby sme naozaj vždy vykonali iba jednu z týchto troch možností a nie viac po sebe. Preto musíme šikovne nastaviť podmienky:

```
aha funkcia0:
krok
funkcia2 ak je stena vpredu
funkcia0 ak nesvieti a nie je stena vzadu
funkcia1 ak svieti a nie je stena vzadu
krok
```

`funkcia1`: Najprv musíme vypnúť prepínač, na ktorom stojíme. Potom urobíme presne to, čo vo `funkcii0`. Dokonca ju môžeme priamo zavolať. Úplne nakoniec ešte zapneme prepínač, ktorý sme na začiatku zhasli (keďže máme prepínače vrátiť v pôvodnom stave).

```
aha funkcia1:
prepni
funkcia0
prepni
```

Chýba nám ešte `funkcia2`. Od tej by sme chceli, aby išla, až kým nepríde na svietiaci prepínač, potom tento prepínač zhasla, robota otočila a vrátila sa o rovnaký počet krokov naspäť. Niečo také sme už robili v leveli 5-Jet torchové peklo z úlohy Prask 1.5 z predošlej série. S tým rozdielom, že vtedy sme sa otáčali, keď bola pred nami stena. Úplne rovnakým spôsobom to urobíme aj teraz. Nesmieme však zabúdať na to, že na ceste máme zákrutu. Cestou tam teda robota pred každým krokom v prípade potreby otočíme doľava a cestou späť ho po každom kroku v prípade možnosti otočíme doprava (takáto formulácia nám zároveň zaručí, že robot bude nakoniec otočený na západ).

```
aha funkcia2:
dolava ak je stena vpredu
krok
funkcia2 ak nesvieti
funkcia3 ak svieti
krok
doprava ak nie je stena vpravo

aha funkcia3:
prepni
dolava
dolava
```

V hlavnom programe na začiatku zavoláme `funkciu0`, po ktorej skončení bude robot na rovnakom políčku ako na začiatku, ale otočený na západ a prepínač v severozápadnom rohu bude vypnutý. Ostatné prepínače budú v pôvodnom stave, teda nám stačí už len odnavigovať robota do cieľa. To je v porovnaní so zvyškom levelu už iba jednoduché cvičenie:

```
aha funkcia4:
doprava ak je stena vpredu
krok
funkcia4
```

Hlavný kód teda bude vyzerať takto:

Kód:

```
funkcia0
funkcia4
```