Milí řešitelé a řešitelky!

První série 24. ročníku je opravena. Pokud se Vám nepovedla, nezoufejte – z chyb je třeba se poučit a druhá série se Vám nepochybně podaří lépe. Než se však vrhnete do jejího řešení, doporučujeme pročíst vzorová řešení – jistě v nich najdete spoustu užitečných triků.

Vzorová řešení první série čtyřiadvacátého ročníku KSP

24-1-1 Podvádíme s XORem

Nejdříve je potřeba rozmyslet podmínky, za kterých lze součástky rozdělit na hromádky se stejným XORem.

Operace XOR je komutativní. Můžeme tedy nejdříve spočítat XOR přes všechny součástky první hromádky, poté XOR přes součástky té druhé. Označme tyto výsledky x a y. Z definice operace XOR snadno nahlédneme, že $x \oplus y$ je v případě $x = y$ rovno nule. Naopak, pro $x \neq y$ je $x \oplus y$ vždy různé od nuly.

Navíc kvůli komutativitě a asociativitě též platí, že pro dané součástky bude XOR mezi libovolnými dvěma hromádkami vždy stejný – hromádky oddělíme závorkami a na pořadí operandů v rámci nich nezáleží. Díky tomu dostáváme, že nulový XOR všech součástek je postačující a zároveň nutnou podmínkou pro existenci řešení.

V případě, kdy je XOR všech prvků nulový, musíme pro co největší rozdíl hodnot hromádek dát kolegovi buď nic, nebo nejlevnější prvek. V zadání nebylo řečeno, zda kolegova hromádka musí býti neprázdná, tudíž jsme i taková řešení považovali za správná. Pro nenulový XOR prvků pak švindl na kolegovi provést nelze.

Jan Bok

24-1-2 Rozházené řádky v BASICu

Tato úloha se ukázala jako lehká, většina z Vás došla ke správnému řešení. Častou chybou byla buď absence důkazu, nebo důkaz pouze pro konkrétní případ.

Pro jednoduchost si budeme zpřeházené řádky představovat pouze jako posloupnost čísel řádků, jak jdou na vstupu po sobě. Představují vlastně permutaci. Nyní si můžeme všimnout několika věcí:

Celou permutaci můžeme rozložit na několik samostatných cyklů. Jako cyklus označíme takovou vybranou podposloupnost prvků, ve které stačí prohodit prvky pouze v rámci této podposloupnosti, abychom dostali prvky na správné pozice (na ty, na které patří), a která se zároveň nedá rozložit na žádné menší cykly.

Speciálním případem cyklu je cyklus o jednom prvku, který představuje prvek již správně umístěný na svém místě. Dokažme si, že v jakémkoliv větším cyklu velikosti k nám stačí právě $k-1$ výměn prvků k tomu, abychom všechny prvky dostali na správné místo.

U cyklu se dvěma prvky platí předpoklad triviálně, zde nám stačí právě jedna výměna k tomu, abychom na správné místo dostali oba dva prvky. U větších cyklů můžeme lehce nahlédnout, že každou výměnou umístíme na správné místo právě jeden prvek. Nakonec se dostaneme do situace, kdy dojde k prohození posledních dvou prvků, při které umístíme správně oba prvky.

Protože jste ale v odevzdaných řešeních měli problém hlavně se správným důkazem, ukážeme si ještě formálně lepší důkaz pomocí indukce. Cykly s jedním a dvěma vrcholy jsme si rozebrali již výše, takže rovnou přistoupíme k indukčnímu kroku a budeme předpokládat, že pro k prvků potřebujeme právě $k-1$ výměn.

Nyní si vezmeme cyklus s $k+1$ prvky. Pokud prvek A vyměníme s prvkem, který se aktuálně nachází na správné pozici prvku A, rozdělíme náš cyklus na dva. Jeden jednoprvkový cyklus je samostatný prvek A, druhý cyklus s k prvky tvoří všechny zbylé prvky z původního cyklu.

O jednoprvkový cyklus se již zajímat nemusíme a z indukčního předpokladu víme, že na druhý cyklus o k prvcích potřebujeme právě $k-1$ výměn. Tedy na původní cyklus s $k+1$ prvky jsme potřebovali $(k-1)+1$ výměn. Tím jsme dokázali naše tvrzení.

Jak tedy spočítat, kolik výměn potřebujeme k navrácení všech prvků na správná místa? Jednou z možností je projít všechny cykly v posloupnosti na vstupu a v každém spočítat počet nutných prohození (tedy počet prvků v cyklu zmenšený o jedna).

Druhou možností je uvědomit si, že za každý cyklus nám stačí započítat pouze onu „-1“, neboli stačí nám spočítat počet cyklů a odečíst ho od celkového počtu prvků (tedy pro posloupnost délky N rozdělenou do C cyklů bude správná odpověď $N - C$).

Implementačně i časově jsou oba postupy stejně náročné. Přesněji pro variantu počítající počet cyklů je paměťová složitost $\mathcal{O}(N)$, protože si na vstupu musíme načíst informace o každém prvku a pamatovat si, které prvky jsme již v cyklu prošli.

A časová složitost je také $\mathcal{O}(N)$, jelikož na každý prvek sáhneme právě dvakrát – jednou při lineárním procházení, jednou při procházení každého cyklu.

Program (C):
`http://ksp.mff.cuni.cz/viz/24-1-2.c`

Jiří Setnička

24-1-3 Turnaj jazyků

Zadání této úlohy by se na první přečtení lekl asi každý; snad proto mi přišla jen hrstka řešení od pár odvážlivců. Pojďme se tedy podívat, jak by zadání vypadalo napsané stručněji a s menší porcí pohádky.

Mějme soutěž o K kolech s N soutěžícími ($N, K \leq 1000$). V každém kole může být vyřazen libovolný (i nulový) počet soutěžících. Po posledním kole ve hře musí zůstat právě jeden, BestLang, jehož bodový zisk za celou soutěž máme maximalizovat.

Body v jednom kole se počítají celočíselně podle vzorce $Vyhra(v, h) = v \cdot 100\,000/h$, kde h je počet hráčů na začátku kola, ze kterých je v vyřazeno. Zisk soutěžícího za celou hru je součet získaných bodů ze všech kol.

Výstup programu má být posloupnost, která v k-tém prvku obsahuje počet vyřazených v k-tém kole. Při tom uvažujeme průběh hry, během které BestLang dosáhne maximálního počtu bodů.

V zadání stále máme některé trochu chlupaté části. Na první pohled působí divně, že by mohlo mít smysl nikoho nevyřazovat. Aby situace byla jasnější, uvedeme si několik jednoduchých pozorování:

- Zisk bodů nezávisí na čísle kola. Jde jen o počet jazyků na začátku kola a počet vyřazených.
- V soutěži proběhne nejvýše $N-1$ vyřazení. Může se stát, že v některém z kol nikdo vyřazen být nemůže – například pro $K > N-1$.
- Nezáleží na umístění kol bez vyřazení, protože tato nijak nemění stav hry (body, počet zbývajících soutěžících). Bez újmy na obecnosti je můžeme umístit třeba na konec.

Příprava je za námi, o problému už trochu něco víme, pusťme se do něj tedy pořádně. První je po ruce procházení všech možných průběhů her, se svojí složitostí $\mathcal{O}(N^K)$ je však beznadějně pomalé. Kdo už někdy potkal podobnou úlohu, bude se zamýšlet nad použitím dynamického programování. I mně se hodilo při psaní vzorového řešení, hodlám se k němu však dostat malou oklikou.

Nebudeme totiž ze začátku vůbec počítat vyřazené soutěžící, ale bodový zisk. Sestrojíme rekurzivní funkci `Zisk`, která pro zadaný počet kol a hráčů spočítá, jaký nejvyšší počet bodů může BestLang získat.

```
int Zisk(int k, int h){
  if (h == 1)
    // poslední soutěžící už nemá koho vyřadit
    return 0;
  if (k == 1)
    // v posledním kole končí i zbylí soupeři
    return Vyhra(h - 1, h);
  int max = 0;
  // v: počet vyřazených (aspoň jeden)
  for (int v = 1; v <= h - 1; v++) {
    int vyhra
          = Vyhra(v, h) + Zisk(k - 1, h - v);
    if (max < vyhra)
      max = vyhra;
  }
  return max;
}
```

Na této funkci je snadno vidět, že skončí a vrátí správný výsledek. Také se objeví jedna důležitá pravidelnost uvnitř úlohy: maximální zisk z posledních k kol je možné spočítat s pomocí maximálního zisku z posledních $k-1$ kol. Nejvýrazněji ovšem stále bije do očí exponenciální časová složitost.

Co naplat, pro zrychlení budeme muset obětovat kousek paměti. Všimneme si, že se rekurzivně ptáme častokrát na stejnou věc – například pokud BestLang nejprve vyřadí jednoho soupeře a potom dva, další rekurzivní volání jsou stejná, jako kdyby nejprve vyřadil dva a potom jednoho.

Dvěma parametry funkce budeme indexovat dvourozměrné pole s tabulkou již spočítaných hodnot zisku. Funkce `Zisk` se při každém volání nejprve podívá, jestli si výsledek nepamatuje. Pokud ano, místo nového počítání vrátí známou hodnotu z tabulky, jinak ji spočítá a před vrácením zapíše.

Nakonec dáme dohromady všechen vtip a postřehy, které jsem dosud utrousil, opustíme rekurzi a půjdeme na řešení dynamicky. Dosavadní pomocná tabulka se stává tím hlavním, o co nám jde. Od `Zisk(K,N)` k `Zisk[K,N]` tak daleko není, význam sloupců a řádků je tedy zřejmý.

Ke spočítání tabulky vlastně jen použijeme to, co už jsme uměli při rekurzi. Jediný myšlenkový rozdíl je, že musíme postupovat pozpátku, od konce hry, po jednotlivých kolech (řádcích).

Poslední kolo (první řádek) má ve všech svých buňkách výhru pro vyřazení všech soupeřů. Při výpočtu každé buňky předchozího řádku se stejně jako v rekurentní verzi hledá maximum ze součtu budoucího zisku a aktuální výhry. Když výpočet dojde až k `Zisk[K,N]`, máme hledaný výsledek.

Opravdu? Ne tak docela, původní úloha se ptala po posloupnosti počtů vyřazených, o maximálním bodovém zisku vůbec nemluvila. Ale tato posloupnost je jenom popisem, jak tolik bodů získat. Jde snadno zrekonstruovat, pokud si každá buňka tabulky pamatuje počet vyřazených soupeřů, při kterém bylo dosaženo maxima bodů. K tomu bude potřeba druhá, stejně velká tabulka, což paměťovou složitost nezhorší.

Paměti celkem potřebujeme $\mathcal{O}(N \cdot K)$, času $\mathcal{O}(N^2 \cdot K)$, protože na každé buňce tabulky trávíme čas $\mathcal{O}(N)$ výpočtem maxima.

Prostor pro zlepšení je dle mého názoru na úrovni konstant, ne typu složitosti. Dokázat to bohužel neumím. Problém nevypadá na první pohled tak složitě, ale celočíselné dělení se každému chytřejšímu přístupu staví do cesty.

Na to narazili i někteří řešitelé. Překvapil mě program, který vypadal, že by mohl fungovat, běží v čase $\mathcal{O}(N \cdot K)$ a prostoru $\mathcal{O}(K)$. Také dynamické programování, ale tentokrát přes počet soutěžících, ne přes počet kol, jak popisuji výše. Většinou dával správný výsledek, ale pro $N, K \leq 100$ se zhruba 500krát seknul. Rozhodnutí, ve kterém kole vyřadit dalšího soutěžícího, bylo uděláno docela správně, ale to bohužel nestačí, protože někdy je potřeba některému kolu počet vyřazených zmenšit.

Při samotné implementaci je vhodné zamyslet se nad datovými typy. Aby byla překročena v prvcích pole velikost 32bitového integeru, muselo by každé z maximálně tisíce kol přispět víc než milionem bodů; ze vzorce však jasně vyplývá, že největší možná výhra za jedno kolo se pouze blíží statisíci.

Program (C):
`http://ksp.mff.cuni.cz/viz/24-1-3.c`

Tomáš „Palec“ Maleček

24-1-4 Složitá složitost

Na úvod poznamenám, že ste si niektorí správne všimli, že ak premennú `odm` inicializujeme na hodnotu $\lfloor\sqrt{n}\rfloor$, tak potom pre niektoré hodnoty n pole `zač` nebude veľkosťou stačiť.

Algoritmus najprv rozdelí pole dĺžky n na $\sqrt{n}$ vzostupne zoradených úsekov $\sqrt{n}$ dlhých. Kým zoradíme jeden úsek (algoritmus využíva Bubblesort), strávime v najhoršom prípade $\mathcal{O}(n)$ času, a to práve vtedy, keď je úsek zoradený zostupne. Zoradenie každého úseku nám preto bude trvať $n\sqrt{n}$ v najhoršom prípade.

Potom algoritmus označí minimá v jednotlivých úsekoch, ktorých je rovnako ako úsekov, tedy $\sqrt{n}$. Ďalej nasleduje n prechodov, kde v každom prechode bude vybraté jedno minimum (ktorých je $\sqrt{n}$) do výsledného poľa.

Za nové minimum označí algoritmus prvok, ktorý je v rámci úseku bezprostredne za aktuálne vybratým minimom. Vytváranie výsledného poľa má teda časovú zložitosť $\mathcal{O}(n\sqrt{n})$.

Z predchádzajících odstavcov plynie, že výsledná časová zložitosť je $\mathcal{O}(n\sqrt{n} + n\sqrt{n}) = \mathcal{O}(n\sqrt{n})$.

Konečne pár slov k pamäťovej zložitosti. Potrebujeme si pamätať n prvkov postupnosti a pri vytváraní výsledného poľa $\sqrt{n}$ pozíc miním, teda pamäťová zložitosť je $\mathcal{O}(n)$.

Peter Zeman

24-1-5 Razítková grafika

Dříve, než začneme hledat největší možné razítko, jakým lze obrázek vyrazítkovat, podíváme se, jak zjistit, zda obrázek lze vyrazítkovat razítkem velikosti S.

Všimneme si, že bod, který je umístěn nejvíce nahoře a nejvíce vlevo, můžeme vyrazítkovat jen tak, že v něm bude mít razítko levý horní roh. Pokud tedy existuje čtverec velký $S \times S$, který má levý horní roh právě v tomto políčku, tak razítko můžeme použít. V opačném případě víme, že obrázek vyrazítkovat nejde.

Na razítkování razítkem velkým S tedy použijeme následující algoritmus. Obrázek budeme procházet po řádcích a vždy, když najdeme černé políčko, tak se podíváme, jestli existuje čtverec velký $S \times S$ mající levý horní roh v tomto políčku. Pokud ano, tak tento čtverec smažeme, a pokud ne, tak obrázek nelze obarvit.

Pokud tímto způsobem projdeme celý obrázek, tak jsme jej právě vyrazítkovali. Každé políčko maximálně jednou přebarvujeme a maximálně jednou přes něj projdeme. Tento algoritmus tedy běží v čase $\mathcal{O}(W \cdot H)$, kde W je šířka a H výška obrázku.

Nyní, když umíme razítkovat, nám stačí najít největší velikost razítka, se kterým obrázek dokážeme vyrazítkovat. Takový přímočarý postup začneme s razítkem o velikosti $\mathcal{O}(\min(W, H))$ a budeme jej postupně zmenšovat, dokud se nám obrázek nepovede vyrazítkovat.

Tento postup má časovou složitost $\mathcal{O}(\min(W, H) \cdot W \cdot H)$, protože například pro černý obrázek s bílým pravým dolním rohem s každým razítkem projdeme skoro celý obrázek.

Další věc, které si můžeme všimnout, je, že pokud obrázek lze vyrazítkovat razítkem velkým S, tak S musí dělit délky všech vodorovných i svislých souvislých úseků (myšleno v rámci jednoho řádku či sloupce).

Tedy velikost razítka musí dělit největšího společného dělitele délek těchto úseků. Stačí nám tedy zkoušet jen velikosti razítek, které dělí největšího společného dělitele. Zdrojový kód tohoto algoritmu je přiložen k vzorovému řešení.

Určitě nevyzkoušíme více než $2 \cdot \sqrt{\min(W, H)}$ razítek, protože žádné číslo k nemá více než $2 \cdot \sqrt{k}$ dělitelů. Snadno můžeme nahlédnout, že pokud $k = a \cdot b$, tak potom $a \leq \sqrt{k}$ nebo $b \leq \sqrt{k}$.

Na počítání největšího společného dělitele použijeme Euklidův algoritmus, který pracuje v logaritmickém čase. Součet čísel, pro které jej zavoláme, je maximálně $W \cdot H \Rightarrow$ celkem Euklidovým algoritmem ztratíme nejvýše čas $\mathcal{O}(W \cdot H)$.

Časovou složitost nám nejvíce ovlivňuje samotné razítkování, celkem tedy dostáváme $\mathcal{O}(W \cdot H \cdot \sqrt{\min(W, H)})$.

Program (C++):
`http://ksp.mff.cuni.cz/viz/24-1-5.cpp`

Karel Tesař

24-1-6 V bludišti s krumpáčem

Jak jste téměř všichni uhádli, mřížka, ve které se pohybujeme, je jen speciálním případem grafu. Je tedy nasnadě pokusit se aplikovat některé grafové algoritmy, které známe z KSP kuchařek či odjinud.

Na náš problém s bludištěm by se hodil jeden ze dvou algoritmů – prohledávání do šířky nebo Dijkstrův algoritmus. Prohledávání do šířky (BFS) má tu výhodu, že nalezne nejkratší cestu ze začátku do cíle v lineárním čase (u nás $\mathcal{O}(N \cdot M)$).

Ve své základní podobě však neumí pracovat se skutečností, že některé cesty, ač stejně dlouhé na počet políček, jsou různě dlouhé co do vzdáleností. Jinak řečeno, nepracuje s ohodnocenými hranami (či v našem případě vrcholy).

Druhý algoritmus, Dijkstrův, vyhledá nejkratší cestu v grafu ohodnoceném nezápornými reálnými čísly. Používá k tomu datovou strukturu *halda*, proto jej také máme popsaný v kuchařce o haldách. Bohužel, jeho časová složitost je vyšší, zde by byla alespoň $\mathcal{O}(N \cdot M \cdot \log(N \cdot M))$.

Snadné řešení tedy bylo napsat „Dijkstra“ a dostat pár bodů. Na plný počet nezbývá, než se zamyslet nad tím, jestli nejde prohledávání do šířky upravit, aby pomohlo i nám, případně jestli nejde Dijkstra zrychlit.

Jednodušší bude upravit prohledávání do šířky. To je v naší kuchařce implementováno pomocí fronty (pokud nevíte, jak fronta funguje, utíkejte si to přečíst).

Když procházíme jedno políčko, obvykle chceme všechny jeho sousedy přidat dozadu do fronty. To v našem bludišti neplatí, protože chceme souseda přidat buď dozadu, nebo „o K míst dál,“ tedy nejen za všechny ve frontě, ale navíc ještě za všechny sousedy, kteří jsou blíž.

Využijeme toho, že máme jen dva typy políček a můžeme si udělat dvě fronty. Jednu pro *rychlá* políčka, tedy ta, kterými procházíme za jeden krok, a jednu pro *pomalá* políčka, tedy pro zdi. Políčka budeme dávat do front podle toho, kterého typu jsou.

Musíme ještě zajistit, abychom nezapomněli vybírat z fronty pro pomalá políčka, když už je čas (tedy poté, co jsou všechny kratší cesty vybrány). Stačí si ke každému políčku připsat, v kterém čase jej máme z fronty vyzvednout. Například pro $K = 5$ a pomalé políčko, které je sousedem políčka s hodnotou 15, připíšeme při uložení do fronty 20. Pro rychlá políčka vždy jen zvýšíme hodnotu o jedna.

Když pak vybíráme z front, jen porovnáváme, jestli má nižší hodnotu rychlé políčko, nebo pomalé políčko, a podle toho volíme nové políčko na prohledání.

V obou frontách budou políčka setříděná podle poznamenané hodnoty (stejně tak, jako by byla setříděna v BFS s jednou frontou). Náš algoritmus se tedy nesplete.

Asymptotická časová složitost je stejná jako pro BFS, neboť přidáním nové fronty nám vzniklo jen konstantní zpomalení – při vybírání dalšího políčka pro průchod jen porovnáváme, ze které fronty ho máme vzít, a jinak se chováme stejně, jako kuchařkové BFS.

Martin Böhm

24-1-7 Distribuované výpočty

Základní myšlenka rešení je jednoduchá – odpojit všechny počítače, které jsou napojeny na aktuální, dřív než sebe.

Budeme procházet graf do hloubky, dokud nenarazíme na vrchol s hranami vedoucími jen k počítačům již odpojeným či navštíveným během rekurze. Ten následně odpojíme (vše, co je k němu připojeno, bude po odpojení stále v síti) a podobně postupujeme dál.

Časová složitost je lineární vzhledem k hranám i vrcholům, protože každý počítač navštívíme právě jednou a na každou hranu se podíváme maximálně dvakrát (z každého konce). Paměťová taktéž.

Program (Pascal):
`http://ksp.mff.cuni.cz/viz/24-1-7.pas`

Pavel Čížek

24-1-8 Pojďte pane, budeme si hrát

Z úkolu v matematické části určitě nebude nikdo smutný, protože byl vcelku lehký, což se projevilo i na veselém bodovém zisku – až na detaily byla řešení správně.

Pro hru s odebíráním žetonů v případě maximálně tří hromádek bylo potřeba ověřit, že smutné stavy jsou právě ty, v nichž je XOR všech velikostí hromádek nulový. Smutný stav si lze představit jako stav předem prohraný – pokud jste ve smutném stavu, soupeř vás může porazit. Je-li pozice mimo smutný stav, hráč na tahu má výherní strategii.

V zadání se tvrdí, že strategie funguje pro libovolný konečný počet hromádek. Abyste nám věřili, vrhněme se na obecnější důkaz!

Bude se nám hodit asociativita a komutativita XORu, tedy že můžeme velikosti hromádek XORovat v libovolném pořadí. Ověření těchto vlastností je mechanickou záležitostí spočívající v rozboru případů. Také je dobré uvědomit si, že i-tý bit v XORu velikostí hromádek může být roven jedné právě tehdy, když má lichý počet hromádek i-tý bit jedničkový.

Začněme nejlehčím požadavkem: prohraný stav se všemi hromádkami prázdnými je smutný. Velikosti hromádek jsou shodně nuly, jejich XOR je nula, tedy stav je smutný.

Proč všechny tahy ze smutných pozic vedou do pozic, které smutné nejsou? To už tak zřejmé není. Mějme tah ze smutné pozice, který odebere k žetonů z hromádky H. XOR všech hromádek byl dosud nula, tedy XOR hromádek mimo H je přesně velikost H.

Po odebrání z H se XOR hromádek mimo H nezměnil, ale H ano. Tedy XORujeme dvě různá čísla, což nikdy nedá nulu, jelikož jejich binární zápis se musí lišit alespoň na jednom místě.

Zbývá poslední požadavek: není-li hráč ve smutném stavu, má tah vedoucí do smutného stavu (takže je vlastně ve „veselém“ stavu, protože má jistou výhru). Dalo by se říci, že z celé úlohy jde o nejzajímavější část, přičemž Vaše řešení se někdy lišila.

XOR velikostí hromádek je nenulový (označme ho X), my chceme po odebrání z jedné hromádky mít smutný stav. Označme i pozici nejlevějšího jedničkového bitu v binárním zápise X. Jedna z hromádek (označme ji H) musí mít velikost alespoň 2^{i-1} a i-tý bit roven jedné – máme lichý počet hromádek, jež mají v binárním zápise na pozici i jedničku.

Z hromádky H odeberu žetony v počtu menším nebo rovném 2^{i-1} tak, aby se vynuloval i-tý bit její velikosti a výsledný XOR všech hromádek byl nulový. Jak se přijde na to, kolik mám odečíst? Jednodušší je přemýšlet, jak velká má být hromádka H, aby výsledný XOR byl nula.

Řešení není těžké: vezmeme velikost hromádky H a překlopíme bit na místech, kde je v X jednička (tedy H vyXORujeme s X). Tím se v XORu všech hromádek změní parita počtu jedniček pouze na bitech, kde byl původně lichý počet jedniček, což dává nulový XOR všech hromádek. Číslo H se navíc muselo zmenšit, jelikož nejlevější změněný bit se překlopil z jedné na nulu.

Q. E. D. (Quite Easily Done nebo Quod Erat Demonstrandum, vyberte si.) Jak je vidět, nebylo potřeba nikde použít počet hromádek, i když jsme předpokládali, že jsou alespoň dvě.

Na závěr řešení tohoto úkolu dodejme, že popsaná hra se jmenuje Nim. Lze ji hrát i s modifikací, kde prohrává ten, kdo vezme poslední žeton z poslední hromádky. Definice smutné (předem prohrané) pozice se pak liší jen v určitých aspektech – můžete si jako cvičení rozmyslet v jakých.

Druhé části seriálové úlohy se zhostili jen nemnozí, ačkoliv šlo o kreativnější úkol. Bylo třeba v Pythonu napsat pro šestvorky ohodnocovací funkci a funkci generující tahy z dané pozice.

Pokud se zdá, že funkce generující tahy měla být jen otrocká práce spočívající ve vygenerování všech dvojic volných políček, není tomu tak. Předně je těch dvojic opravdu hodně (po prvním tahu $\binom{224}{2}$, tedy 24 976) a většina z tahů postrádá smysl, protože jsou třeba na kraji desky, kde se nikde v okolí nehraje.

Dobrou heuristikou mohlo být hledání linie svých značek, kterou je možno v jednom tahu vyhrát (tj. například 4 značky v řadě s oběma volnými konci). Pokud neexistuje, tak hledání soupeřovy linie, s níž by mohl vyhrát dalším tahem, a jinak generování všech dvojic z políček sousedících s nějakou značkou.

Ještě zajímavější je ohodnocování pozice. Určitě je dobré při něm zkoumat, jestli už není pozice vyhraná nebo prohraná. Jinak se hodí třeba hledat souvislé linie jednoho hráče, které mají alespoň na jednom konci volné políčko, a ty ohodnocovat podle délky (např. exponenciálně), přičemž je dobré zohlednit, jestli má linie oba konce volné, nebo jen jeden.

Ohodnocení je pak součet ohodnocení mých linií minus součet soupeřových linií. Vše pak záleží na dobrém nastavení konstant. Je to však jen jeden z mnoha možných přístupů a určitě půjde vymyslet lepší :-)

Pavel „Paulie“ Veselý