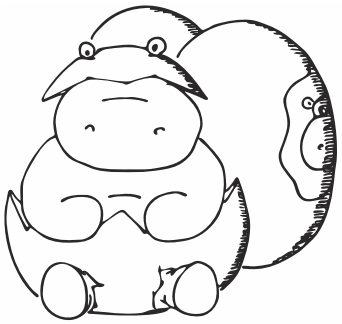
Milí řešitelé a řešitelky!



Držíte v ruce čtvrtý leták 24. ročníku KSP. Každá série letos obsahuje 8 úloh a z nich se 5 nejlépe vyřešených započítává do celkového bodového hodnocení.

Nově je možno být přijat na MFF UK za úspěšné řešení KSP. Úspěšným řešitelem se stává ten, kdo získá za celý ročník alespoň 50 % bodů. Za letošní rok půjde získat maximálně 300 bodů, takže hranice pro úspěšné řešitele je 150.

Upozorňujeme letošní maturanty, že termín odevzdání páté série bude příliš pozdě na to, aby pátou sérií dohánělí chybějící body. Diplom úspěšného řešitele ale můžeme v případě potřeby zaslat i dříve, budete-li mít dost bodů.

Termín odevzdání čtvrté série je stanoven na **pondělí 9. dubna** v 8:00 SEČ, což znamená, že papírové řešení byste měli podat na poštu do středy 4. dubna, aby nám stihlo přijít. CodExová úloha má termín o den posunutý, protože nám ji opravuje automat – 10. dubna v 8:00.

Řešení přijímáme elektronicky na stránce *https://ksp.mff.cuni.cz/submit/*. Chcete-li s námi komunikovat bezpečně, můžete si ověřit náš HTTPS certifikát – zde je jeho SHA1 hash: `7F:53:E7:00:60:F2:24:93:8F:52:51:EC:1E:A8:34:54:86:69:32:7D`.

Také nám řešení můžete poslat klasickou poštou na adresu

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

Čtvrtá série čtyřiadvacátého ročníku KSP

Den první: „Já se na to tedy podívám“ povzdechl si John a vstal od papírování. Přitom hodil okem po kalendáři. Pátý leden 2137, už jenom tři dny do slavnostního otevření sekce výstupních zařízení muzea počítačů. A pořád nemáme funkční exponáty, zaklel v duchu a poklusem se dal k oddělení elektronkových počítačů.

U přesné kopie prvního elektronkového počítače ENIAC z roku 1946, jak hlásal holografický panel, jej přivítal jeden z techniků. „Problém je s tímhle panelem,“ řekl a ukázal na desku se žárovičkami. Ta, jak si John pamatoval, neměla u původního ENIACu žádný klíčový význam, ale měla sloužit pro vizualizaci výpočtu. Běžní lidé chtěli vidět, že ta ohromná konstrukce něco dělá a blikající světla byla nejlepší. Od té doby také několik desítek let přežívala představa počítače, jako stroje s chaoticky blikajícími kontrolkami.

24-4-1 Iniciály předků — 10 bodů

Tým techniků chce nechat na panelu se žárovičkami postupně zobrazovat nějaký řetězec znaků, aby panel pěkně blikal a upoutalo to procházející lidi. Panel je tvořený spoustou sloupců žárovek a každý sloupec umí zobrazit jeden znak.

Technici si jako správní hračičkové sepsali iniciály všech svých předků až do 20. století a ty chtějí nechat zobrazovat na panelu.

Nicméně, čím je řetězec delší, tím déle se musí vkládat do paměti počítače. Technici si chtějí ušetřit práci, a tak by chtěli znát takovou jeho nejkratší část, jejímž opakováním se dá vypsat celý řetězec.

Vaším úkolem je tedy napsat program, který si na vstupu přečte řetězec znaků (složený z velkých písmen anglické abecedy), nalezne v něm nejkratší úsek, jehož opakováním vznikne celý řetězec, a vypíše jeho délku.

vstup	odpověď
`ABABAB`	`2`
`ABABA`	`5`
`AAA`	`1`

Tato úloha je praktická a řeší se ve vyhodnocovacím systému CodEx.[1] Přesný formát vstupu a výstupu, povolené jazyky a další technické informace jsou uvedeny v CodExu přímo u úlohy.

John se po vyřešení problému ještě chvíli procházel oddělením nejstarších počítačů a zkoumal další výstupní zařízení. U jedné staré tiskárny se dal do řeči s průvodcem, který si zrovna cvičil svůj výklad.

„Než přišly na scénu různé obrazovky, velmi oblíbenou metodou výstupu byl tisk na různý tiskárnách či elektronických psacích strojích,“ začal průvodce. „Neuměly samozřejmě tisknout nějakou grafiku, natožpak trojrozměrně, jako ty dnešní. Ale zvládaly celkem rychle tisknout znaky. Tiskárny zvládající tisk nějakých jiných obrazců než znaků přišly až v 60. letech 20. století.“

Přešel k prvnímu exponátu „Nejdříve se používaly jehličkové tiskárny, které se stylem fungování podobaly psacím strojům. Pomocí jehliček přitiskávaly barevnou pásku na papír... pozor, pane, ta páska pořád barví. Až teprve počátkem 70. let se začal prosazovat laserový tisk. Ten funguje v základě tak, že se pomocí laseru na správných místech vybije elektrostaticky nabitý rotující selenový válec. Toner se přichytí pouze na vybitá místa, pohybem válce následně přilne na papír. A nakonec se toner do papíru zapeče.“

„A co inkoustové tiskárny, myslel jsem, že přišly dříve než laserové?“

„To je častý omyl. Inkoustové tiskárny se začaly prosazovat až počátkem 80. let, ale protože byly levnější na výrobu, prosadily se hlavně v domácím prostředí. Fungují tak, že se inkoust v tryskách tiskové hlavy zahřeje pomocí maličkého elektrického tělíska asi na 300°C a pak vlivem tlaku ve velké rychlosti vystřikne z trysky na papír.“

Výklad o tiskárnách byl ale přerušen jedním ze strážných muzea. „Problém šéfe, spadnul nám systém zjišťování polohy exponátů. Víme, kde exponát je, ale systém už nevyhodnotí, jestli je pořád uvnitř budovy, nebo ne.“ To tu ještě scházelo, pomyslí si John, ale nedávaje na sobě znát únavu posledních dní se vydává za strážným do velínu bezpečnosti, kde si nechává vysvětlit fungování celého systému, tedy spíše jeho nefungování. Bude nutné ho celý přepsat.

[1] `http://ksp.mff.cuni.cz/zaciname/codex.html`

24-4-2 Sledování exponátů — 8 bodů

Hranice muzea počítačů má tvar nekonvexního mnohoúhelníku. Na sledovaném exponátu je připevněno čidlo, které vysílá jeho aktuální polohu (určenou například pomocí GPS).

Měli byste strážným v muzeu pomoci tím, že vymyslíte postup, jak zjistit, jestli je exponát ještě na území muzea, nebo ne. Jediné, co máte k dispozici, je poloha exponátu a posloupnost vrcholů hranice muzea.

Byla už skoro půlnoc, když John konečně vítězoslavně klepl do potvrzovací klávesy a zvedl se od počítače. Tak, další problém vyřešený, poklepal se v duchu po rameni. Teď ale musím stihnout ten banket v aule muzea.

Rychle proběhl skrz kancelář, vzal na sebe společenský oblek a pospíchal, ať stihne alespoň půlnoční přípitek.

24-4-3 Cinkání skleničkami — 8 bodů

Přípitky ve 22. století mají několik základních pravidel. Stojí se v kruhu, všichni si musí cinknout se všemi a dále se nesmí cinkat „křížem“ (když si dva páry lidí cinkají, nesmí se jim zkřížit ruce).

A aby to nebylo tak jednoduché, cinká se v taktech. Vždy na úder gongu si člověk buď cinkne s někým, nebo zůstane stát. Pak na další úder gongu s dalším člověkem a tak dále, dokud si necinkne se všemi.

Vás zajímá, kolik nejméně takovýchto taktů bude potřeba, aby si navzájem cinklo N lidí a také správný postup, jakým si budou cinkat. Nezapomeňte dokázat, že to na méně taktů nejde.

Druhý den ráno přišel John do práce s hrozným bolením hlavy – neměl to včera s těmi přípitky přehánět. V kanceláři si jenom vzal něco na bolest, počkal, až prášek zabere, a pak šel zkontrolovat konečné přípravy otevření expozice. Jen co vešel do hlavní haly, všiml si podivného ruchu u dveří skladu.

„Jako by nám někdo nepřál otevření,“ uvítal ho vrchní skladník. „Máme výpadek proudu ve skladu. A to zrovna potřebujeme navézt několik beden s těmi, jak se jim říkalo… monitory, myslím. Jsme schopni nabít každému vozíku baterie trochou energie, ale chtělo by to nějak optimalizovat jejich trasy, jinak to prostě z toho skladu nestihneme vyvézt.“

24-4-4 Vozíky ve skladu — 10 bodů

Moderní sklad 22. století je obsluhován pouze automatickými elektrickými vozíky. Ty normálně čerpají energii z rozvodné sítě vozíků, ale při výpadku této sítě jsou schopné fungovat i na baterie. Samotný sklad je spleť křižovatek a uliček. Uličky jsou obousměrné, mohou se křížit i víceúrovňově a setkávají se pouze na křižovatkách.

Navíc pod podlahou některých uliček jsou silnoproudé vodiče, které svým magnetickým polem ztěžují vozíkům průjezd. Silnoproudé vodiče jsou napojeny na oddělený okruh, výpadek je tedy neovlivní. Samozřejmě v nich „teče“ střídavý proud, který indukuje (střídavé) magnetické pole – to v jedné půlce svojí periody vozík zpomaluje, ve druhé zrychluje.

Skladníci zjistili, že by toto pole mohli využít – nastavili vozíky tak, aby jim průjezd libovolnou uličkou trval vždy stejně dlouho, a to právě půlku periody střídavého proudu. Délka cesty a magnetické pole ovlivňují jen spotřebu energie.

Vzhledem k výpadku napájení se hlavní skladník pokouší optimalizovat trasy jednotlivých vozíků a potřebuje od vás najít energeticky nejúspornější trasu (tj. trasu, při níž vozík spotřebuje nejméně energie z baterií) mezi dvěma křižovatkami, které si určí.

Na vstupu dostanete mapu skladu popsanou jednotlivými křižovatkami spolu s uličkami, které mezi nimi vedou. Každá ulička má danou spotřebu energie při průjezdu. Dále dostenete seznam uliček, pod kterými vedou silnoproudé vodiče – můžete si je představit tak, že každý lichý průjezd libovolnou křižovatkou zdvojnásobí jejich energetickou náročnost, každý sudý průjezd ji vrátí do počátečního stavu.

Samozřejmě víte i odkud kam má vozík jet – tedy startovní a cílovou křižovatku. Nejúspornější trasu vypište jako pořadí křižovatek. Nezapomeňte, že skladníci spěchají, vozík tedy nesmí nikdy stát.

Příklad: máme 5 křižovatek očíslovaných 0 až 4 a chceme vozík přepravit z 0 do 1. Všechny uličky obsahují silnoproudé vodiče a vedou mezi křižovatkami (v závorce je energie spotřebovaná při průjezdu): 0 a 1 (21), 0 a 2 (10), 2 a 3 (5), 3 a 4 (2), 2 a 4 (5), 4 a 1 (10).

Nejvýhodnější je použít cestu $0 \to 2 \to 3 \to 4 \to 1$, při níž se spotřebuje 39 jednotek energie. Kdyby se jelo uličkou z 0 rovnou do 1, stálo by to 42 jednotek; cesta $0 \to 2 \to 4 \to 1$ by stála 45 jednotek.

Když se ze skladu konečně dostaly i poslední palety s monitory, John si oddechl. Mezitím, co se hlavní skladník zabýval vozíky, si John dokonce něco stihl nastudovat i o prastarých monitorech.

Jak psali ve starém propagačním letáku, první monitory byly jednobarevné, napevno vestavěné do počítačů a nebylo možné k jakémukoliv počítači připojit jakýkoliv monitor. Za první univerzální grafickou kartu se standardizovaným adaptérem se dá považovat až Monochrome Display Adapter (MDA) z roku 1981 od Intelu, k němuž se dal přes konektor podobný pozdějšímu VGA (ale s méně piny) připojit jakýkoliv monitor s tímto konektorem. MDA umožňoval výstup 80 sloupců na 25 řádků znaků.

Později se objevily i karty podporující nejen znakový, ale i grafický režim a v roce 1987 přišel standard Video Graphics Array (VGA) se svým konektorem, který přežil přes 25 let. Stále se ale jednalo o analogový výstup. První digitální výstup do připojeného monitoru přišel až v roce 1999 společně se standardem a konektorem Digital Visual Interface (DVI).

John přestal pročítat brožuru, rychle nalistoval poslední kapitolu se základním rozebráním principu dvou nejrozšířenějších zobrazovačů přelomu tisíciletí a četl. Starším byla technologie katodové trubice, tedy CRT monitory. Fungovaly na stejném principu jako tehdejší televize. Elektronové dělo vysílalo proud nabitých částic, které byly usměrňovány velkými elektromagnety, na dopadovou plochu zvanou stínítko. Tam se pomocí látky zvané luminofor proud elektronů měnil na viditelné světlo.

Druhou technologií, která se začala kvůli ceně prosazovat až počátkem 90. let a k jejímuž masovému rozšíření došlo až po přelomu tisíciletí, byla technologie tekutých krystalů LCD. Pracovala na principu zastiňování světla. Za deskou z tekutých krystalů bylo osvětlovací těleso, produkující bílé viditelné světlo. Samotná deska z tekutých krystalů pak v závislosti na natočení krystalků buď světlo v daném bodě propouštěla, nebo ne. Natočení krystalků v jednotlivých bodech bylo řízeno pomocí slabého elektrického proudu.

„Teda, ti si s tím vyhráli," hvízdl obdivně John a odložil brožuru. Pak se podíval směrem ke vstupu do expozice, kde měla skupinka pracovníků problém s naprosto současnou zobrazovací technikou, s holografickými projektory.

24-4-5 Holografické projektory — 12 bodů

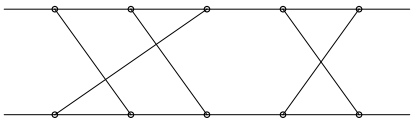
Do expozice muzea se vstupuje dlouhou chodbou, v níž jsou na jedné stěně instalovány holografické projektory. Každý projektor promítá na přesně určené místo na druhé stěně chodby.

Žádné dva promítané obrazy na stěně se sice nepřekrývají a žádné dva projektory nejsou na jednom místě, ale může se stát (a vzhledem k návrhům uměleckého designéra na uspořádání se stává dost často), že se paprsky nějakých dvou projektorů cestou kříží. A aby u holografických projektorů nedošlo k nechtěné interferenci a rozmazání obrazu, musí v takovém případě pracovat oba projektory na jiné frekvenci.

Technik, který už tak má bolení hlavy z návrhů designéra, zároveň chce, aby bylo použito co nejméně frekvencí, protože je to jednodušší na údržbu. Když se projektory nekříží, mohou mít klidně stejnou frekvenci. Ale žádné dva křížící se projektory nemůžou pracovat na stejné frekvenci.

Navrhněte tedy postup, jak co nejrychleji určit, na jakých frekvencích mají pracovat které projektory, tak, aby počet použitých frekvencí byl co nejmenší. Od designéra dostanete pouze rozmístění promítaných obrazů na stěně (například očíslované podle pořadí odpovídajících projektorů na druhé stěně).

Příklad: pro vstup `3 1 2 5 4` se paprsky na stejné frekvenci nekříží například při rozdělení: frekvence 1 – projektory 1, 2, 4, frekvence 2 – projektory 3, 5. Viz obrázek:



„Tak vidíte, že to šlo. A vypadá to pěkně." usmál se John na designéra, když mu konečně vymluvil některé jeho šílenější nápady s umístěním holografických projektorů. Rozloučili se a John se vydal dál, až úplně dozadu celého prostoru připravované expozice. Tam sídlila výstava netradičních výstupních zařízení.

Na podstavci u vstupu stál Braillský řádek. Jak říkal popisek, tato pomůcka pro nevidomé mohla zobrazovat až 80 znaků v Braillově písmu. Zobrazení jednotlivých znaků měly na starost většinou malé elektromagnety, které nadzvedly odpovídající výstupky. Nevidomý tedy mohl procházet jakýkoliv textový obsah na obrazovce a na Braillském řádku si ho přečíst.

Další zajímavostí byla ukázka technologie, která se začala rozvíjet na přelomu prvního a druhého desetiletí 21. století, takzvaných generátorů vůně. Vstupní data pro tyto věci mohla pocházet buď ze speciálního programu, nebo z chemického čidla na druhé straně komunikační linky. Generátor pachu pak z několika základních chemických vůní (podobně jako monitor z několika základních barev) poskládal pach nebo vůni, která se tomu co nejvíce blížila.

„Něco takového bych si domů asi nepořídil," řekl John a pak leknutím uskočil, neboť se hned vedle něj náhle rozsvítila velká obrazovka plná spousty znaků. „Promiňte pane, jenom tady procházíme stará záznamová média a koukáme, co by šlo zobrazit na těchto obrazovkách, hned to dám pryč."

„Počkejte!" vyhkrl John, v němž se probudila zvídavost. „Vždyť to je kus nějakého starého programového kódu, k čemu asi... hmm... počkejte, už je mi to asi jasné. Ale proč je to napsané takhle neefektivně?"

24-4-6 Starý kód — 9 bodů

Pracovníci muzea počítačů nalezli na jednom starém disku následující kód. Zkuste zjistit, co vlastně kód dělá, a zamyslete se nad tím, jestli by nešel přepsat nějak efektivněji.

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

#define MAX_H 1000000
#define MAX_V 1001

typedef struct {
  int x, y;
}H;
int N, M;
H h[MAX_H];
int v[MAX_V][MAX_V];
int p[MAX_V];
int f[2*MAX_V];
short b[MAX_V];

int main() {
  scanf("%d%d", &N, &M);
  if (N>MAX_V || M>MAX_H) {
    printf("Chybny vstup.\n");
    return 1;
  }
  for (int i=0; i<M; i++) {
    int x, y;
    scanf("%d%d", &x, &y);
    if (x>N || x<1 || y>N || y<1) {
      printf("Chybny vstup.\n");
      return 1;
    }
    h[i] = (H){x, y};
    v[x][p[x]++] = y;
    v[y][p[y]++] = x;
  }
  printf("Vysledny seznam:\n");
  for (int k=0; k<M; k++) {
    int a = 0;
    int z = 0;
    for (int i=1; i<=N; i++)
      b[i] = 0;
    b[h[k].x] = 1;
    f[z++] = h[k].x;
    while (a<z && b[h[k].y]==0) {
      int q = f[a++];
      for (int i=0; i<p[q]; i++) {
        if (b[v[q][i]]==0 && !(q==h[k].x
            && v[q][i]==h[k].y)) {
          f[z++] = v[q][i];
          b[v[q][i]] = 1;
        }
      }
    }
    if (b[h[k].y]==0)
      printf("%d %d\n", h[k].x, h[k].y);
  }
  return 0;
}
```

Vedle něj byl objeven druhý, podobný kód, u kterého byla poznámka, že až na nepodstatný rozdíl ve čtení vstupu dělá to samé:

```
import sys
MAX_H = 1000000
MAX_V = 1001
v = []; p = []; f = []; b = []; h = []
for j in range(MAX_V+1):
  v.append([]); b.append(0); p.append(0)
for j in range(2*MAX_V):
  f.append(0)

N, M = raw_input().split(' ')
N = int(N); M = int(M)
if N > MAX_V or M > MAX_H:
  sys.exit("Chybny vstup.")

for i in range(M):
  x, y = raw_input().split(' ')
  x = int(x); y = int(y)
  if(x>N or x<1 or y>N or y<1):
    sys.exit("Chybny vstup.")
  h.append((x, y))
  v[x].append(y); p[x] += 1
  v[y].append(x); p[y] += 1

print "Vysledny seznam:"
for k in range(M):
  a = 0; z = 0
  for i in range(1,N+1):
    b[i] = 0
  b[h[k][0]] = 1
  f[z] = h[k][0]; z += 1
  while(a<z and b[h[k][1]] == 0):
    q = f[a]; a += 1
    for i in range(p[q]):
      if b[v[q][i]] == 0 and not (
        q == h[k][0] and v[q][i] == h[k][1]
      ):
        f[z] = v[q][i]; z += 1
        b[v[q][i]] = 1

  if b[h[k][1]] == 0:
    print h[k][0], h[k][1]
```

Když John dozkoumal kód, pozval ho ten stejný technik, který ho vylekal, dál. „Nechcete se podívat na ty staré helmy virtuální reality, co jsme zrovna vybalili?“

První helmy virtuální reality se začaly objevovat koncem 80. a počátkem 90. let. V podstatě šlo o helmu se dvěma malými obrazovkami, pro každé oko jedna. Ve spojení ještě například s rukavicemi poskytujícími hmatovou odezvu se tak člověk mohl ponořit do světa virtuální reality.

Helmy se ale díky své ceně a váze nikdy příliš neuplatnily. Na přelomu prvního a druhého desetiletí nového tisíciletí jejich funkci částečně převzaly technologie trojrozměrných brýlí a odpovídajících obrazovek, které byly mnohem dostupnější než drahé helmy.

„Nechcete si třeba vyzkoušet nějakou starou hru?“ zeptal se technik a aniž by čekal na odpověď, spustil hru s nejnápadnějším názvem.

24-4-7 Čtvercové bombardování — 13 bodů

⚠ Představte si, že máte velké město a chcete ho srovnat se zemí. Třeba protože se vám už nelíbí a chcete místo starých domů postavit nové, moderní.

Máte k dispozici bombardér se speciální demoliční bombou. Na demoličních bombách je zajímavé to, že jsou pečlivě sestrojené tak, aby srovnaly se zemí pouze přesně danou čtvercovou oblast. A protože jste nakoupili kvalitní demoliční bomby, bouchají navíc pouze směrem na východ a jih.

Tedy pokud shodíte demoliční bombu s rázem D do místa $[x, y]$, budou zdemolovány všechny budovy ve čtverci vymezeném body $[x, y]$ a $[x + D, y + D]$, ale nic jiného. Protože ale chcete demolovat efektivně, bude lepší si vše předem propočítat.

Pro zjednodušení budeme budovy považovat za body – na vstupu dostanete jejich počet $B < 250\,000$ a jejich souřadnice $[x_i; y_i]; -10^9 < x_i, y_i < 10^9$. Zkuste vymyslet program, kterého se budete moci ptát, kolik budov bude zbouráno, když do místa $[x, y]$ hodíte bombu s rázem $0 < D < 2 \cdot 10^9$. Všechny souřadnice jsou celočíselné.

Počítejte s tím, že těchto dotazů bude program dostávat řádově statisíce, takže se pokuste, aby odpovědi na dotazy byly rychlé i za cenu delšího úvodního předpočítání.

„To teda byla hra!“ smál se John, když sundaval helmu. „Děkuju.“

Technik s úsměvem převzal helmu a podíval se na obrazovku, kde svítilo „Úroveň New York dokončena, přejete si pokračovat?“

I nadešel poslední den před otevřením, do slavnostního přestřižení pásky zbývalo již jen několik hodin a vše konečně vypadalo připravené. Projektory svítily, panel se žárovičkami blikal, vozíky ve skladu opět jezdily a sledovací systém exponátů spokojeně předl.

Nestrhne se na poslední chvíli ještě nějaká pohroma? Bude konečně dopřáno Johnovi přestřihnout v klidu slavnostní pásku? Prozradím vám, že ano. Co se ale stane několik sekund po přestřižení pásky, to je už jiný příběh. Možná někdy příště...

Od klávesnice se s Vámi loučí váš dnešní průvodce muzeem počítačů

Jirka Setnička

24-4-8 O hrách a číslech — 15 bodů

Vítejte u dalšího dílu herního seriálu. Podrobněji rozvineme Conwayovu teorii her, konkrétně se naučíme hry porovnávat a některým přiřazovat čísla.

Zopakujme si nejdůležitější pojmy z minula. Zajímají nás hry dvou hráčů označovaných jako *Levý* a *pRavý*. Vše jsme si dosud ukazovali na *dominování*, které spočívá v pokládání dominových kostek do čtvercové mřížky. Levý pokládá svisLe, pravý vodoRovně.

Dále jsme rozdělili hry (přesněji řečeno pozice) do čtyř tříd dle vítěze:

- vyhrané pozice: vyhraje hráč, který bude táhnout jako první; třída V
- prohrané hry: začínající hráč prohraje; třída P
- pozice levého: levý vždy vyhraje, ať začne kdokoliv; třída L
- pozice pravého: analogicky k levému; třída R

Důležité bylo sčítání pozic, které jsou nezávislé (nelze zahrát do obou najednou), přičemž začínající hráč si vždy může vybrat, kam potáhne.

V tomto díle se nám bude hodit rovnost her: hry G a H se rovnají, tedy $G = H$, pokud dopadnou stejně, když k oběma přičteme libovolnou jinou hru. Rovněž budeme využívat i obrácené hry značené $-G$, v níž si hráči vymění možné

tahy, které od teď povedou do podobně obrácených pozic. V dominování odpovídá $-G$ otočení herního plánu o 90°.

Posledním úkolem v minulém díle bylo ukázat, že z $G = H$ vyplývá, že $G - H$ je prohraná hra. Tvrzení však platí i opačně: pokud $G - H$ je prohraná hra, pak $G = H$.

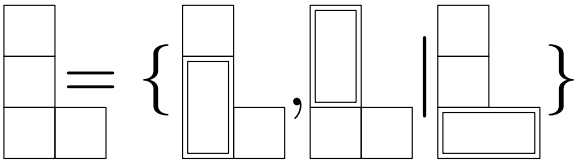
Když víme, které hry se rovnají, jak poznat, že nějaká hra je lepší pro levého než jiná? Opět budeme zkoumat hru $G-H$. Je-li to pozice levého (levý vyhraje, ať začíná kdokoliv), pak G je pro levého lepší než H, což se zapisuje jako $G > H$.

Pokud je pozice $G-H$ v třídě R, tak $G < H$. Dvojicím her, pro něž $G - H$ je pozice vyhraná pro začínajícího hráče, budeme říkat *neporovnatelné*, což se značí $G \parallel H$.

Tímto jsme si definovali částečné uspořádání na hrách. Stejně jako pro jiná uspořádání z $G < H$ a $H < I$ vyplývá $G < I$ (analogicky pro pravého).

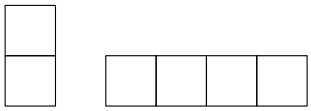
Číslování her

Než začneme číslovat hry, doplníme ještě abstraktní zápis her, v němž bude pozice G vypadat takto: $G = \{\mathcal{G}_L \mid \mathcal{G}_P\}$. $\mathcal{G}_L$ je množina pozic, kam může táhnout levý hráč, obdobně $\mathcal{G}_P$ jsou hry po tazích pravého hráče. Jelikož takto suchá definice může snadno zmást, podívejte se na obrázek:



Nyní se můžeme pustit do přiřazování čísel hrám. Ty budou vyjadřovat, jak moc velkou má levý nebo pravý výhodu v pozici oproti soupeři. Velikost čísla bude udávat, kolik tahů má jeden z hráčů navíc (kupodivu to vyjde někdy neceločíselně).

Kladná čísla budou znamenat výhodu levého a záporná výhodu pravého. Všimněte si, že pozici levého hráče nemůžeme přiřadit záporné číslo, jelikož v ní má alespoň malou výhodu levý. Obdobně pozice z třídy R nemohou dostat kladné číslo.

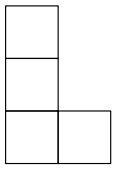


V pozici vlevo má levý hráč jeden tah navíc, hra dostane tedy číslo 1. V pozici vpravo má pravý dva tahy a levý nic, proto -2. V abstraktním zápise se hra s kladným číslem n dá zapsat jako $\{n-1 \mid \}$ (levý hráč může táhnout do hry s číslem $n-1$), hra $n < 0$ analogicky jako $\{ \mid n+1\}$.

Když máme kladná i záporná čísla, je logické se ptát, co bude 0. V souladu s definicí kladných i záporných her znamená 0, že žádný z hráčů nemá výhodu, ani ten, co je na tahu, čili je to prohraná pozice.

Nejjednodušší taková hra je $\{ \mid \}$ (žádný hráč nemá tah) a každá prohraná hra se rovná 0. (Pro prohranou hru G není těžké ověřit, že $G = 0$. Nejsnadněji asi důkazem, že $G - 0$ je prohraná hra.)

Na začátku jsme tvrdili, že čísla her mohou být neceločíselná. Konkrétně mohou být jen tzv. *diadická racionální*, tedy zlomky $n/2^m$ v základním tvaru, kde n je celé číslo a m přirozené (včetně 0).



Na následujícím obrázku je hra s hodnotou 1/2, v níž levý získá tah navíc, jen když začíná pravý:

Jak zjistit hodnotu dané pozice, když máme rozebrány hry, do nichž hráči mohou táhnout, nám řekne *pravidlo jednoduchosti*.

Mějme hru $G = \{\mathcal{G}_L \mid \mathcal{G}_R\}$, přičemž všechny tahy obou hráčů vedou do pozic, jež jsou čísla, a každý tah levého vede do pozice s číslem menším, než mají všechny pozice po tazích pravého hráče (formálně $\forall G_L \in \mathcal{G}_L$, $\forall G_R \in \mathcal{G}_R$: $G_L < G_R$). Potom G je nejjednodušší číslo x, nacházející se mezi hodnotami pozic levého a pravého ($G_L < x < G_R$).

Nejjednodušší v minulém odstavci znamená, že x je diadické, čili $n/2^m$ v základním tvaru, m je co nejmenší (preferují se celá čísla) a mezi čísly se stejným m se vybere to s menší absolutní hodnotou n.

Příklady:

- $\{-1 \mid 1\} = 0$
- $\{-100 \mid 10\} = 0$
- $\{-42 \mid -4\} = -5$
- $\{0 \mid 1\} = 1/2$
- $\{\{1 \mid -1\} \mid \{1 \mid -1\}\} = 0$ (je to prohraná hra)
- $\{1 \mid -1\}$ není číslo
- $\{-5 \mid -10\}$ není číslo (hra je však vyhraná pro pravého)
- $\{0 \mid 0\}$ není číslo

Poslední hra v seznamu, $\{0 \mid 0\}$, je nejjednodušší vyhranou hrou a značí se $*$.

Všimněte si, že některé hry levého či pravého jsou čísla a jiné ne. Žádná vyhraná hra však není číslo (výhodu má ten, kdo začne, ne vždy levý nebo pravý) a naopak každá prohraná hra se rovná 0, i když se to nemusí nahlédnout přes pravidlo jednoduchosti.

Her, které nejsou čísla, je hodně. Například $\uparrow = \{0 \mid *\}$, analogicky $\downarrow = \{* \mid 0\} = -\uparrow$. Hrám $\{a \mid b\}$, kde $a > b$, se říká přepínač, speciálně $\pm a = \{a \mid -a\}$ pro $a > 0$.

Co se týče uspořádání dle výhodnosti pro levého (či pravého), tak platí například (ověření necháme jako cvičení):

- $\{-10 \mid 10\} = \{-1 \mid 1\} = 0$,
- $\{1 \mid 2\} < \{1 \mid 6\}$,
- $\uparrow > 0$ a symetricky $\downarrow < 0$,
- $* \parallel 0$,
- $\pm 1 \parallel 0$,
- $\pm 10 \parallel \pm 5$.

Bylo by nyní potřeba ukázat, že hry, jež se rovnají, mají stejná čísla (mají-li vůbec nějaká). Nebo že součet her G a H má číslo rovné součtu čísel G a H.

Celkově by však důkazy zabraly možná celou sérii, takže případné zájemce odkáži na literaturu a internet (viz níže). Jejich hlavním důsledkem je, že lze libovolně zaměňovat hru a k ní příslušející číslo.

Místo důkazů zkusíme hry zjednodušovat. Podívejme se třeba na hru $G = \{3, 2, *, 0 \mid 4, 6, 8, 10\}$. Levý nemá žádný důvod táhnout do 2, $*$ nebo 0, podobně pravý potáhne určitě do 4, tedy $G = \{3 \mid 4\} = 3,5$.

Obecně lze v možnostech levého vyškrtat pozice horší pro levého než nějaká jiná pozice a analogicky mezi tahy pravého škrtáme pozice větší než nějaká jiná. Formálně zapsáno (pro možnosti levého): je-li $G = \{A, B, \ldots \mid C, D \ldots\}$, přičemž $A > B$ nebo $A = B$, pak $G = \{A, \ldots \mid C, D \ldots\}$ (nerovnosti mezi hrami jsou ty samé, co byly definovány na začátku tohoto dílu).

Úkol 1 [4b]: Mějme hromádku n sirek. Je-li n sudé, levý na tahu odebírá dvě sirky a pravý jednu. Je-li n liché, vezme levý jednu sirku a pravý dvě. Určete číslo hry pro každé $n \geq 0$.

Úkol 2 [5b]: Výše jsme si ukazovali pozici v dominování s hodnotou 1/2 (označme ji G). Ověřte, že $G + G = 1$. Dále nalezněte v dominování pozici H, jež není číslo, ale $H + H = 1$.

Úkol 3 [6b]: Hra *Padající domino* se hraje s bílými a černými dominovými kostkami postavenými v řadě za sebou. Tah hráče spočívá ve výběru jedné své kostky a jejím shození doleva nebo doprava, přičemž díky tomu spadnou všechny kostky ve směru, kam padala.

Levý hraje s bíLými kostkami, pravý s čeRnými a opět platí, že prohrál ten, kdo nemůže táhnout. Pro jednoduchost budeme posloupnost kostek zapisovat jako řetězec s písmeny B a C zastupujícími bílé a černé kostky. Například z pozice CBC má levý dva tahy, oba vedoucí do pozice C.

Pro hry BCBBBBC a BBCCBC najděte co nejjednodušší abstraktní zápis, jež neobsahuje konkrétní pozice, ale jen čísla, $*$, $\uparrow$ apod. (tedy třeba $\{\{4 \mid 2\} \mid -6\}$). Vyškrtáváte-li nějakou možnost, je třeba zdůvodnit, proč.

Tím jsme zakončili spíše neformální úvod do Conwayovy teorie her. Toto odvětví matematiky je však podstatně košatější, vynechali jsme dost důkazů (i zajímavých!), teploty a teploměry her (určování, jak moc výhodné je do hry zahrát) a mnoho dalších zajímavých věcí.

Co se týče praktické využitelnosti teorie, je na ní založený algoritmus pro řešení koncovek v Go nazvaný *Decomposition search*.

Prohloubit své znalosti teorie si můžete mimo jiné přečtením *Winning Ways for your Mathematical Plays* od Berlekampa, Conwaye a Guye a *On Numbers and Games* od Conwaye (ani jedna z nich bohužel nemá český překlad).

Mimochodem, hry v zápise $\{A, B, C, \ldots \mid P, Q, R, \ldots\}$ jsou tzv. *nadreálná čísla* (jejich speciálním případem jsou i reálná čísla), o nichž se dočtete více na Wikipedii.[2]

Hračičkům doporučujeme program CG Suite,[3] v němž lze zadávat pozice z různých her (nebo zapsané nadreálným číslem) a nechat určit jejich hodnotu, teplotu a další vlastnosti. (To může sloužit i pro kontrolu řešení úkolů, bude však třeba vše zdůvodnit.)

Příště se můžete těšit na návrat výpočetní části seriálu započaté v první a druhé sérii (algoritmy Minimax a Alfa-beta ořezávání). Nově nabyté znalosti si budete moct vyzkoušet na analýze jedné deskovky, kterou bude možné hrát online.

Pavel „Paulie“ Veselý

Recepty z programátorské kuchařky

Řetězec je v podstatě jakákoli posloupnost symbolů zapsaná za sebou a s nimi budeme v této kapitole pracovat. Každého napadne „vyhledávání v textu“ nebo „hledání jmen v telefonním seznamu“, ale řetězce najdeme i na nižších úrovních informatiky. Například celé číslo zakódované v binární soustavě, které dostaneme na vstupu programu, je také jen řetězec nul a jedniček.

Jiný příklad použití řetězců (a jejich algoritmů) najdeme v biologii. DNA není o mnoho více, než chytré uložení posloupnosti čtyř znaků/nukleových bazí – a chceme-li hledat vzory anebo konkrétní podposloupnosti, bude se nám hodit znalost základních algoritmů pro práci s řetězci.

Nemáme bohužel šanci vysvětlit *všechny* algoritmy s řetězci, protože je příliš mnoho možných věcí, co s řetězci dělat. Převáděním řetězců na čísla (hešováním) jsme se věnovali v jiné kuchařce, v této se budeme soustředit na algoritmy, které se objevují spíše v práci s textem.

Kromě úvodu popíšeme dva *stavební kameny* textových algoritmů, což bude jedna datová struktura pro adresáře (trie) a jedno vyhledání v textu s předzpracováním hledaného slova (a jeho rozšíření pro více slov). S jejich znalostí se pak mnohem snáze vymýšlí řešení složitějších, reálnějších problémů.

Jak řetězce chápat

Když programátor dělá první krůčky, často moc netuší, co s těmi řetězci vlastně může a nesmí dělat. V programovacím jazyce to je jasné – něco mu jazyk dovolí a na něco nejsou prostředky. Ale jak to je na úrovni ryze teoretické?

Jak jsme si řekli na začátku, řetězec bude posloupnost nějakých symbolů, kterým říkáme *znaky*. Tyto znaky jsou z nějaké množiny, které říkáme *abeceda*. Abeceda může být jen `01` pro čísla v binárním zápisu, klasické `A-Za-z` pro anglickou abecedu anebo plný rozsah univerzální znakové sady Unicode, která má až 2^{31} znaků. Nezapomínejme, že nejenom písmena a číslice, ale i mezery a interpunkce jsou znaky!

Vidíme, že zanedbat velikost abecedy při odhadu složitosti by bylo příliš troufalé, a tak budeme velikost abecedy označovat $|\Sigma|$. Abeceda samotná se v textech o řetězcích často značí řeckým Σ.

O znacích samotných předpokládáme, že jsou dostatečně malé, abychom s nimi mohli pracovat v konstantním čase, podobně jako s celými čísly v ostatních kapitolách.

Nyní hlavní otázka – máme chápat řetězec jako pole znaků, nebo jako spojový seznam? Šalamounská odpověď: můžeme s ním pracovat tak i tak. Když budeme potřebovat převést řetězec na spojový seznam (protože se nám hodí rychlé přepojování řetězců), tak si jej převedeme. Tento převod nás samozřejmě bude stát čas lineárně závislý na *délce* řetězce. Budeme ji značit dále L; časová složitost převodu bude $\mathcal{O}(L)$.

Standardně se ale počítá s tím, že řetězec je uložen v poli někde v paměti (již od začátku algoritmu), takže ke každému znaku můžeme přistupovat v konstantním čase.

Jelikož jsme řetězce definovali jako posloupnosti, nesmíme zapomínat ani na *prázdný řetězec* ε. A když už máme řetězec, určitě máme i *podřetězec* – souvislou podposloupnost znaků jiného řetězce. Například `BAR`, `RET`, ε i `KABARET` jsou podřetězce slova (řetězce) `KABARET`; `KAT` však podřetězcem není.

[2] `http://cs.wikipedia.org/wiki/Nadre%C3%A1ln%C3%A9_%C4%8D%C3%ADslo`

[3] `http://www.cgsuite.org/`

Často nás budou zajímat dva zvláštní druhy podřetězců. Pokud ze slova odstraníme nějaký souvislý úsek na konci, vznikne podřetězec, kterému říkáme *prefix* (česky předpona), a pokud odstraníme nějaký souvislý úsek ze začátku, dostaneme *suffix* neboli příponu. `RET` je suffix slova `KABARET`, `KABA` je zase jeho prefixem.

Terminologie dovoluje zepředu i zezadu odstranit prázdný řetězec – to znamená, že slovo je samo sobě prefixem i suffixem.

Pokud chceme mluvit o prefixech, suffixech nebo obecně podřetězcích, kde jsme museli alespoň jeden znak odtrhnout, označíme takové podřetězce jako *vlastní*.

Pro některá použití řetězců je důležité, abychom je mohli porovnávat – když máme řetězce R a S, tak rozhodnout, který je menší, a který je větší. Jaké přesně toto uspořádání bude, závisí na naší aplikaci, ale mnohdy se používá tzv. *lexikografické uspořádání*.

Pro lexikografické uspořádání potřebujeme nejprve zadané (lineární) uspořádání na znacích (kromě binárního `0` < `1` se často používá „telefonní“ `A` = `a` < `B` = `b` < ... < `Z` = `z`, které je ovšem lineární až na velikost znaků).

Když máme zadané uspořádání na znacích, na všechny řetězce jej rozšíříme následovně: nejkratší je prázdný řetězec a ostatní řetězce třídíme podle znaků od začátku do konce. Zvláštnost je v tom, že řetězec je větší než jeho každá vlastní předpona (neboli *prefix*). Řetězec `A` tedy bude menší než `AUTO`, které samo bude menší než `AUTOBUS`.

Adresář pomocí trie

Typický problém v oblasti textu je, že máme seznam nějakých řetězců (často třeba jmenný adresář), můžeme si jej nějak předzpracovat, a pak bychom rádi efektivně odpovídali na otázku: „Je řetězec S obsažen v adresáři?“ Můžeme také po předzpracování chtít přidávat nové položky i odebírat staré.

Pokud bychom nemuseli odebírat jména, můžeme použít hešování, které je rychlé a účinné. Více o něm najdete v kuchařce o hešování.[4] Má však tu nevýhodu, že při velkém zaplnění se začne chovat pomaleji a mírně nepředvídatelně.

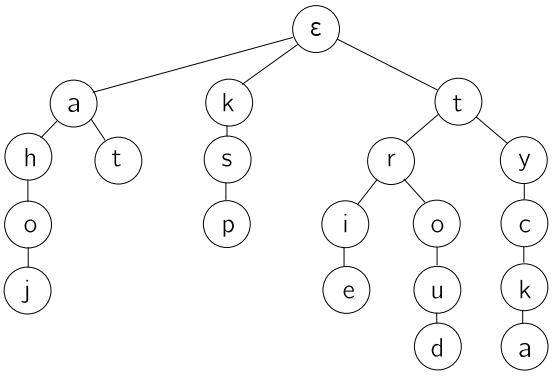
Ukážeme si jiné řešení, které je také asymptoticky rychlé a není ani příliš náročné na paměť. Využívá stromové struktury a říká se mu *trie* (vyslovujeme česky „tryje“ a anglicky jako část slova „retrieval“, z něhož slovo trie vzniklo). V češtině se občas používá také označení „písmenkový strom“.

Trie bude zakořeněný strom, budeme jej stavět pro nějaký adresář A. Kořen bude odpovídat prázdnému slovu ε. Každá hrana, která z něj povede, odpovídá jednomu ze znaků, kterým slovo z adresáře A začíná, a to bez opakování (tedy jsou-li v A čtyři slova začínající na `A`, hranu vedeme jen jednu).

Na koncích těchto hran z kořene nám vznikly vrcholy, které odpovídají všem jednoznakovým prefixům slov z A, a už je celkem jasné, jak struktura dále pokračuje – z každého vrcholu odpovídajícímu prefixu P vede hrana se znakem c právě tehdy, když slovo $P + c$ (za P přilepíme znak c) je také prefixem některého slova z A.

[4] `http://ksp.mff.cuni.cz/viz/kucharky/hesovani`

Obrázek vydá za tisíc definic, zde je postavená trie pro slova `AHOJ`, `AT`, `KSP`, `TRIE`, `TROUD`, `TYC`, `TYCKA`:



Jak bychom takovou trii postavili algoritmem? Přesně, jak jsme ji definovali: každé slovo z adresáře budeme procházet znak po znaku a bude-li nějaká hrana chybět, tak ji vytvoříme a pokračujeme dále podle slova.

Z takto popsané trie bohužel nepoznáme, kde končí slovo z adresáře a kde končí jen jeho prefix. Standardní způsoby, jak to vyřešit, jsou dva: buď si do každého vrcholu přidáme informaci o tom, je-li koncem celého slova nebo ne, anebo si rozšíříme abecedu o speciální znak, který se v ní předtím nevyskytoval – třeba `$` – a pak všem slovům z A přilepíme tento `$` na konec.

Budeme-li se později ptát, bylo-li slovo v adresáři, po průchodu trií zkontrolujeme ještě, jestli z konečného vrcholu vede hrana odpovídající znaku `$`.

Ještě jsme si nerozmysleli, jak budeme v jednotlivých vrcholech trie reprezentovat hrany do delších prefixů. Abychom mohli vyhledávat skutečně lineárně, potřebovali bychom umět v konstantním čase odpovědět na otázku „má vrchol P potomka přes hranu se znakem c?“.

Abychom zajistili konstantní čas odpovědi, museli bychom mít v každém vrcholu pole indexované znaky abecedy. To ovšem znamená, že takové pole budeme muset vytvořit, a tedy alokovat $|\Sigma|$ políček v každém znaku.

To zvýší paměťovou náročnost trie (a časovou náročnost) na $\mathcal{O}(D \cdot |\Sigma|)$, kde D značí velikost vstupu, čili součet délek všech slov v adresáři. To je naprosto přijatelné pro malé abecedy, ale už pro `A-Za-z` je tento faktor roven 52 a pro Unicode je už taková alokace nemyslitelná.

Pokud tedy pracujeme s velkou abecedou, může se nám vyplatit oželet konstantní rychlost dotazu a použít v každém vrcholu vlastní binární vyhledávací strom pro znaky, kterými aktuální prefix může pokračovat. To zmírní časovou složitost konstrukce na $\mathcal{O}(D \cdot \log |\Sigma|)$ a zhorší časovou složitost dotazu na slovo délky L na $\mathcal{O}(L \cdot \log |\Sigma|)$.

A jsme hotovi! S trií můžeme v lineárním čase odpovídat na dotazy „Vyskytuje se dané slovo v adresáři?“, přidávat a odebírat další položky za běhu a nejen to – víc o tom ve cvičeních.

Poznámky

- Chcete-li algoritmus konstrukce trie vidět napsaný v Pascalu, podívejte se do knihy *Algoritmy a programovací techniky*.
- Triím se také říká *prefixové stromy*, což popisuje, že každý vrchol odpovídá prefixu některého slova v adresáři. (Slovo *prefixové* je však v matematice hodně nadužívané (prefixová notace, prefixové kódy), a tak to může vést ke zmatení.)

- Kdybychom chtěli, mohli bychom pomocí trie vyhledávat v českém textu v lineárním čase. Můžeme přeci postavit adresář ze všech slov v daném textu, a pak procházet tu trii. Má to ale pár háčků: jednak je často hledaný řetězec krátký, ale text se nevejde do paměti. Druhak, pokud bychom použili jako oddělovač mezery, mohli bychom hledat jen jednotlivá slova, a nikoli jejich konce nebo delší kusy věty.
- Asi se po poslední poznámce ptáte – existuje nějaká modifikace trie, která umí hledat libovolnou část textu? Ano, jmenuje se *suffixový strom* a jdou s ní dělat spousty krásných kousků. Říká se, že každou řetězcovou úlohu lze řešit v lineárním čase pomocí suffixových stromů. Víc se o nich dočtete třeba v knížce *Grafové algoritmy*.[5]

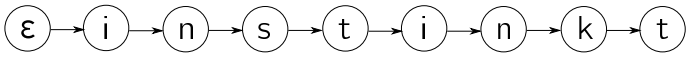
Cvičení

- Řekněme, že chceme adresář na vstupu setřídit v lexikografickém pořadí (definovaném v sekci „Jak řetězce chápat“). Můžeme použít nějaký klasický třídicí algoritmus, ale bohužel musíme počítat s tím, že porovnání dvou řetězců není konstantně rychlé. Vymyslete způsob, jak setřídit takový adresář pomocí trie.
- *Komprese trie.* Co kdybychom chtěli odstranit přebytečné vrcholy trie, tedy ty, v nichž se slova nevětví? Rozmyslete si, jestli by něčemu vadilo místo takovýchto cest mít jen jednotlivé hrany. Zesložití se konstrukce nebo vyhledávání? Mimochodem, je celkem jasné, že takováto *komprimovaná trie* přinese jen konstantní zrychlení dotazů i prostoru, a tak na soutěžích apod. stačí použít základní variantu.

Vyhledávání v textu

Začátek situace je asi zřejmý – máme na vstupu zadán dlouhý text a krátké slovo. Chceme si slovo zpracovat, načež projdeme co nejrychleji text a zahlásíme jeden nebo všechny jeho výskyty. Zajímají nás při tom i výskyty, které se navzájem překrývají: v textu `NANANA` se slovo `NANA` vyskytuje dvakrát. Často se hovoří o „hledání jehly v kupce sena“, a tedy se textu přezdívá *seno* a hledanému slovu *jehla*. Délku jehly označíme D a délku textu H.

Představme si nejdříve hledané slovo jako spojový seznam, třeba slovo `INSTINKT`:



Mohli bychom text začít procházet znak po znaku a kontrolovat, zda se text shoduje s naším slovem/spojovým seznamem. Pokud by si znaky odpovídaly, skočíme na další znak z textu a i na další znak v seznamu. Co když se ale neshodují? Pak nemůžeme jen skočit na další znak textu – co kdybychom v textu narazili na slovo `INSTINSTINKT`?

Musíme se tedy vrátit nejen na začátek spojového seznamu, ale i zpátky v textu na druhý znak, který jsme označili jako odpovídající, a zkoušet porovnávat s jehlou znovu od začátku. To už naznačuje, že takto získaný algoritmus nebude lineární, protože se musí vracet zpět v textu o délku jehly.

Sice je předchozí popis skutečně v nejhorším případě složitý $\mathcal{O}(H \cdot D)$, avšak stačí malá úprava a složitost přejde na lineární $\mathcal{O}(H + D)$. Ve skutečnosti algoritmus nezpomalovalo vracení se – za špatnou složitost mohl fakt, že jsme se vraceli *příliš zpátky*.

[5] `http://mj.ucw.cz/vyuka/ga/`

Třeba v našem příkladu s textem `INSTINSTINKT` se nemusíme vracet ve spojovém seznamu na začátek, jakmile načteme `INSTINS`. Mohli jsme se vrátit jen na druhý znak, tedy do prvního `N`, a pak kontrolovat, jaký znak pokračuje dál. Když následuje `S` jako v našem případě, můžeme pokračovat dále v čtení a nevracíme se v textu. Kdyby text byl jiný, třeba `INSTINB`, vrátili bychom se po načtení `B` na začátek spojového seznamu a v textu bychom pokračovali dále bez zastavení.

Pro každý znak ve spojovém seznamu si tedy určíme políčko spojového seznamu, na které skočíme, pokud se následující znak v textu liší od toho očekávaného. Pořadové číslo tohoto políčka nám poradí tzv. *zpětná funkce* F, což bude funkce definovaná pomocí pole, kde $F[i]$ bude pořadové číslo políčka, na které se má skočit z políčka číslo i. Porovnávat pak budeme s následujícím znakem. Pokud $F[i] = 0$, znamená to, že máme začít porovnávat úplně od prvního znaku jehly.

Pokud máte rádi grafovou terminologii, můžete se na náš spojový seznam dívat jako na graf a hovořit o *zpětných hranách*.

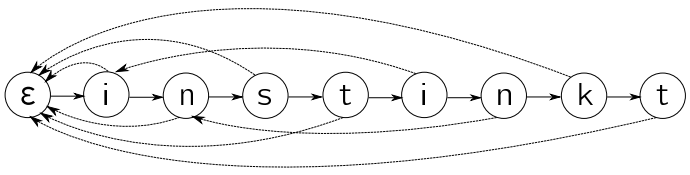
Zatím jsme ale přesně nepopsali, na které políčko přesně bude zpětná funkce ukazovat. Nechť chceme určit zpětné políčko pro druhé `N` ve slově `INSTINKT`. Pracujeme teď s prefixem `INSTIN`. Selsky řečeno, chceme najít „konec slova `INSTIN` takový, že je stejný, jako začátek slova `INSTIN`“.

Abychom náš požadavek upřesnili, zamyslíme se nad zpětným políčkem pro jiné slovo. Co kdyby jehlou bylo slovo `ABABABC` a my určovali zpětné políčko pro `ABABAB`? Kdybychom ukázali na první písmenko `B`, nebylo by to správně, protože pak bychom pro text `ABABABABC` nezahlásili výskyt jehly, což je jasná chyba. Musíme se vrátit už na `ABAB`!

Zajímá nás tedy ne libovolný suffix, který je stejný jako začátek, ale nejdelší takový konec/suffix. A ještě navíc ne jen ten nejdelší, ale nejdelší „netriviální“ – slovo `INSTIN` je samo sobě prefixem a suffixem, ale zpětná funkce pro `N` by se neměla cyklit, měla by vést zpátky.

Řekněme to tedy znova, zcela formálně: pokud bychom právě určovali hodnotu zpětné funkce pro znak číslo i, kterému odpovídá prefix P, pak její hodnota bude *délka nejdelšího vlastního suffixu* slova P, pro který ještě platí, že *je zároveň prefixem* P.

Pro slovo `INSTINKT` vypadá spojový seznam se zpětnou funkcí (zakreslenou pomocí ukazatelů) takto:



Nyní vyvstávají dvě otázky: Jakou to má celé časovou složitost? Jak spočítat zpětnou funkci?

Poperme se nejdříve s tou první. Pro každý znak vstupního textu mohou nastat dva případy: Buď znak rozšiřuje aktuální prefix, nebo musíme použít zpětnou funkci. První případ má jasně konstantní složitost, druhý je horší, neboť zpětná funkce může být pro jeden znak volána až D-krát.

Při každém volání však klesne pořadové číslo aktuálního stavu (políčka) alespoň o jedna, zatímco kdykoliv stav prodlužujeme, roste jen o jeden znak. Proto všech zkrácení dohromady může být nejvýše tolik, kolik bylo všech prodloužení, čili kolik jsme přečetli znaků textu. Celkem je tedy počet kroků automatu lineární v délce textu, $\mathcal{O}(H)$.

Konstrukci zpětné funkce provedeme malým trikem. Všimněme si, že $F[i]$ je přesně číslo stavu, do nějž se dostaneme při spuštění našeho vyhledávacího algoritmu na řetězec, který tvoří prefix délky i z jehly bez prvního znaku.

Proč to tak je? Zpětná funkce říká, jaký je nejdelší vlastní suffix daného stavu, který je také stavem, zatímco políčko, ve kterém po i krocích skončíme, označuje nejdelší suffix textu, který je stavem. Tyto dvě věci se přeci liší jen v tom, že ta druhá připouští i nevlastní suffixy, a právě tomu zabráníme odstraněním prvního znaku.

Takže F získáme tak, že spustíme vyhledávání na část samotné jehly. Jenže k vyhledávání zase potřebujeme funkci F. Jak z toho ven? Budeme zpětnou funkci vytvářet postupně od nejkratších prefixů. Zřejmě $F[1] = 0$. Pokud již máme $F[i]$, pak výpočet $F[i+1]$ odpovídá spuštění automatu na slovo délky i a při tom budeme zpětnou funkci potřebovat jen pro stavy délky i nebo menší, pro které ji již máme hotovou.

Navíc nemusíme pro jednotlivé prefixy spouštět výpočet vždy znovu od začátku – $(i+1)$-ní prefix je přeci prodloužením i-tého prefixu o jeden znak. Stačí tedy spustit algoritmus na celou jehlu bez prvního znaku a sledovat, jakými stavy bude procházet, a to budou přesně hodnoty zpětné funkce.

Vytvoření zpětné funkce se nám tak nakonec zredukovalo na jediné vyhledávání v textu o délce $D-1$, a proto poběží v čase $\mathcal{O}(D)$. Časová složitost celého algoritmu tedy bude $\mathcal{O}(H+D)$. Dodáme už jen, že tento algoritmus poprvé popsali pánové Knuth, Morris a Pratt a na jejich počest se mu říká KMP. Naprogramovaný bude vypadat následovně (čtení vstupu jsme si odpustili):

```
var
  Slovo: array[1..D] of char;   { jehla }
  Text: array[1..H] of char;    { seno }
  F: array[1..D] of integer; { zpětná fce }
  function Krok(I: integer; C: char): integer;
  begin
    if (I < D) and (Slovo[I+1] = C) then
      Krok := I + 1
    else if I > 0 then
      Krok := Krok(F[I], C)
    else
      Krok := 0;
  end;
var I, J: integer;  { pomocné proměnné }
begin
  { konstrukce zpětné funkce }
  F[1]:= 0;
  for I:= 2 to D do
    F[I]:= Krok(F[I-1], Slovo[I]);
  { procházení textu }
  J:= 0;
  for I:= 1 to H do begin
    J:= Krok(J, Text[I]);
    if J = D then writeln(I);
  end;
end.
```

Poznámky

- Pro anglický nebo český text je použití takto sofistikovaného algoritmu skoro škoda, protože v obou jazycích se stává jen málokdy, že bychom měli několik slov spojených dohromady. Prakticky bude stačit i na začátku zmíněný naivní algoritmus. Na soutěžích a olympiádách ale pište raději algoritmus KMP.
- Hešování lze použít i na vyhledávání řetězce v textu. Obzvláště vhodné jsou na to *rolling hash functions* („okénkové hešovací funkce“), které umí v konstantním čase přepočítat heš, ubereme-li nějaký znak na začátku a přidáme-li jiný na konci – jako kdybychom se dívali na text skrz posouvající se okénko.

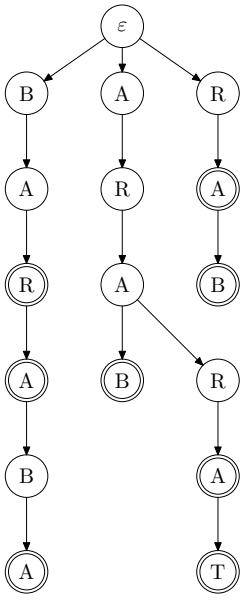
Cvičení

- Rozmyslete si, že když vyhledáváme více slov, ne jen jedno, a algoritmus musí vypsat všechny výskyty na výstup, můžeme se dobrat vyšší než lineární složitosti v závislosti na vstupu. Na čem potom taková časová složitost také záleží?
- Vymyslete nějakou vhodnou okénkovou hešovací funkci pro vyhledávání jedné jehly.

Vyhledávání jehelníčku

Co kdybychom neměli jen jednu jehlu/hledané slovo, ale celý jehelníček, čili seznam hledaných slov? I to lze řešit podobnou metodou, jako jsme řešili jedno slovo. Tento algoritmus se nazývá po tvůrcích *algoritmus Aho-Corasicková* a spočívá v tom, že jednoduchý spojový seznam nahradíme trií a do trie opět přidáme zpětné hrany.

Budeme postupovat podobně jako u KMP. Nejprve naskládáme jehelníček do trie. Pro příklady v této kuchařce použijeme jehelníček `ARAB`, `ARARA`, `ARARAT`, `BAR`, `BARA`, `BARABA`, `RA` a `RAB`.



Dalším krokem v KMP bylo sestrojení zpětných hran. Nejprve jsme sestrojili zpětnou hranu pro první znak slova, pak pro druhý atd. Ve trii to bude o něco složitější.

Na první pohled se může zdát, že bychom mohli automat sestrojit tak, že bychom vyrobili KMP pro první slovo, pak KMP pro druhé slovo s využitím struktury prvního atd., ale to má háček.

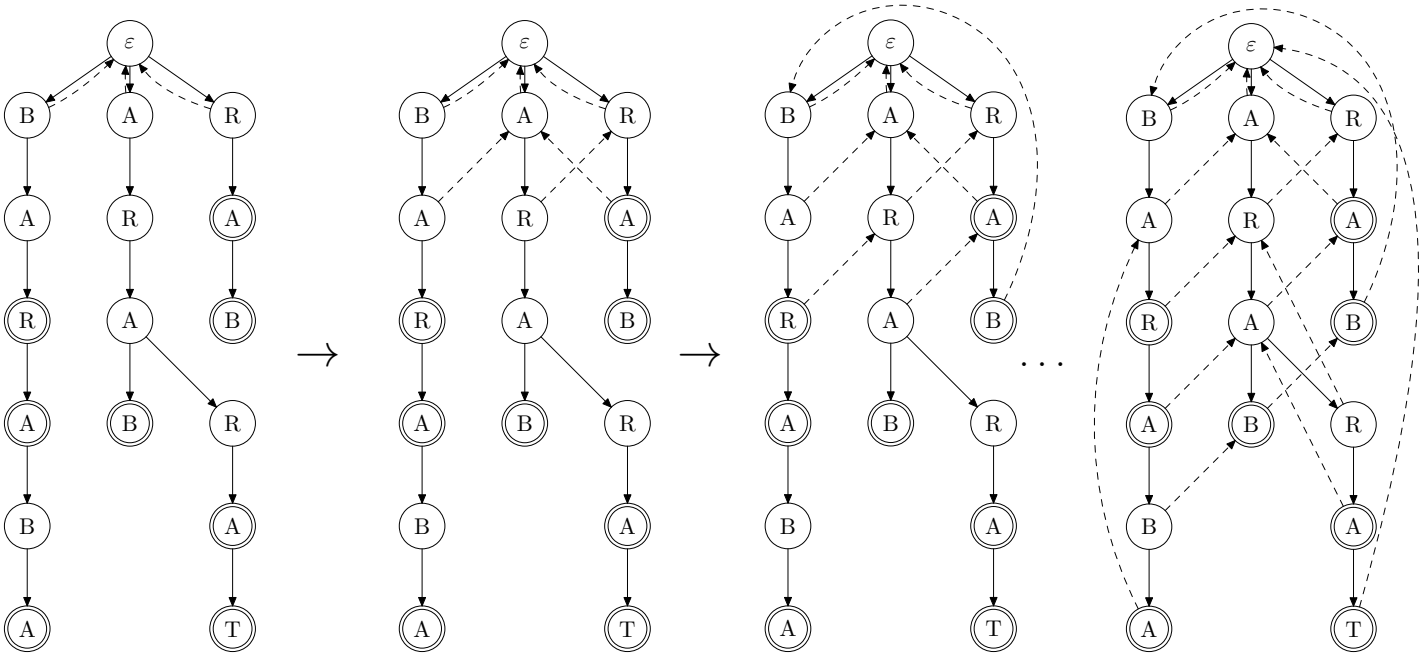
Zpětné hrany totiž nemusí vést do předka. Například pro slovo `BARAB` povede zpětná hrana do slova `ARAB`, z toho do slova `RAB` a z toho do `B`. Kdybychom ale zkonstruovali automat výše popsaným způsobem (a začali slovem `BARAB`), nebude existovat v trii ani `ARAB`, ani `RAB`, takže bychom vedli zpětnou hranu chybně do `B`.

Můžeme se ale opřít o trik z konstrukce KMP – vyhledání svého nejdelšího vlastního suffixu. Kam dojde výpočet po jeho vyhledání, tam povede zpětná hrana.

Zkusíme tedy nejprve sestrojit celou trii a pak postupně vyhledat nejdelší vlastní suffix pro každé slovo. Ouha, to také nefunguje. Když začneme slovem `BARABA` a budeme tedy vyhledávat `ARABA`, nalezneme v trii úspěšně prefix `ARAB`, ale

`ARABA` již v trii není. Měli bychom přejít ze slova `ARAB` po zpětné hraně, ale tu ještě nemáme zkonstruovanou.

Rozdělíme si trii na *vrstvy* – první znaky slov budou první vrstva, druhé znaky budou tvořit druhou vrstvu atd., až i-té znaky slov budou tvořit i-tou vrstvu.



Zpětná hrana jistě povede do kratšího slova. Z i-té vrstvy tedy povede do vrstvy s nižším pořadovým číslem. Pokud tedy budeme zpětné hrany konstruovat po vrstvách, dojdeme kýženého výsledku.

Ještě zbývá otázka, jak konstruovat zpětné hrany efektivně, když je musíme vyrábět po vrstvách. Mohli bychom prostě vzít slovo, pro které hledáme zpětnou hranu, utrhnout mu první znak a vyhledat. Jenže to budeme dělat spoustu práce zbytečně.

Například pro slovo `BARABA` bychom mohli vyhledávat `ARABA` v již zkonstruované části automatu, ale proč to dělat celé, když jsme při konstrukci předchozí vrstvy vyhledávali `ARAB` při konstrukci zpětné hrany pro `BARAB`?

Najdeme tedy akorát, kde jsme minule skončili, a odtamtud pokračujeme dál. Jak to najdeme? Z otce našeho vrcholu tam přece vede zpětná hrana. Takže můžeme postup shrnout do bodů:

1. c = poslední znak slova (znak stavu P, pro který hledáme zpětnou hranu);
2. přesuneme se do otce;
3. přesuneme se po zpětné hraně;
4. dokud neexistuje syn se znakem c nebo nejsme v kořeni, přesouváme se po zpětných hranách;
5. pokud existuje syn se znakem c, natáhneme do něj zpětnou hranu z P, jinak ji natáhneme do kořene.

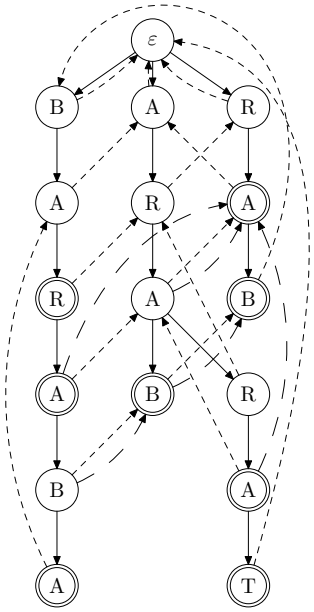
Automat je zkonstruován. Časová složitost konstrukce sestává z konstrukce trie v $\mathcal{O}(D \cdot |\Sigma|)$, resp. $\mathcal{O}(D \cdot \log |\Sigma|)$ (pokud použijeme binární strom ve vrcholech) a z předpočítání zpětných hran. Při předpočítávání uděláme nějaký konstantní počet operací pro každý vrchol (celkem tedy $\mathcal{O}(D)$) a také paralelně vyhledáváme všechny jehly z jehelníčku, jejichž vyhledání nás stojí $\mathcal{O}(D)$, resp. $\mathcal{O}(D \cdot \log |\Sigma|)$.

Tedy konstrukce trvá celkem $\mathcal{O}(D \cdot |\Sigma|)$, resp. $\mathcal{O}(D \cdot \log |\Sigma|)$, paměťová náročnost je stejná jako u trie – $\mathcal{O}(D \cdot |\Sigma|)$, resp. $\mathcal{O}(D)$, přidali jsme jen $\mathcal{O}(D)$ zpětných hran.

Projdeme tedy automatem text `BARABARARAT`. Ohlásí postupně nález slov `BAR`, `BARA`, `BARABA`, `BAR`, `BARA`, `ARARA` a `ARARAT`.

Nenalezl však všechno. Chybí mu např. `ARAB`, který začíná druhým znakem a končí pátým. Dále chybí několik výskytů `RA` a jeden `RAB`.

Když byl na pátém znaku, byl ve stavu `BARAB`, jehož suffixem je `ARAB`. Obecně na suffixy zapomínáme – narozdíl od KMP, kde suffix aktuálního stavu nikdy nebyl jehla, tady jehlou být může.



V každém stavu bychom tedy měli projít všechny suffixy a zkontrolovat je, jestli náhodou nejsou jehlami. Jak najdeme všechny suffixy? Projdeme postupně po zpětných hranách až do kořene. Má to jen jeden problém – je to pomalé.

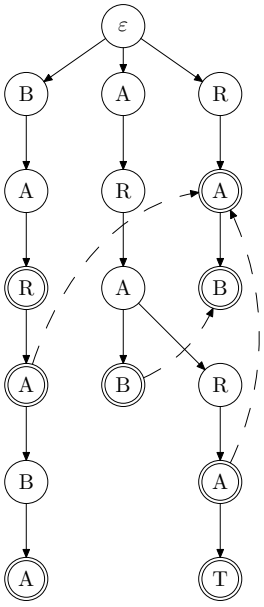
Představme si například slovník obsahující `A` a `AAAA...A` (délky $D-1$). Budeme-li jím vyhledávat v textu `AAAA...A` délky $H > D$, projdeme prakticky pro každý znak až $D-1$ zpětných hran, čímž složitost naroste až na nepoužitelných $\mathcal{O}(H \cdot D)$.

Všimněme si však, že většinou zpětných hran jsme prošli úplně zbytečně. Předpočítáme si tedy *zkratky* – z vrcholu vede zkratka do nejdelšího jeho suffixu, který je jehlou. Na obrázku jsou vyznačeny dlouze čárkovanými šipkami.

Předpočítání zpětných hran časovou složitost konstrukce automatu jistě nezhorší, neboť vyžaduje v nejhorším případě projít všechny zpětné hrany ještě jednou.

Potřebujeme-li ohlásit všechny výskyty slov včetně pozice, kde se nacházejí, jsme hotovi. Výsledná časová složitost prohledávání bude $\mathcal{O}(H+O)$, resp. $\mathcal{O}(H \cdot \log|\Sigma| + O)$, kde O je velikost výstupu – počet výskytů všech slov.

Celková časová složitost prohledávání včetně stavby automatu tedy bude $\mathcal{O}(O + H + D \cdot |\Sigma|)$, resp. $\mathcal{O}(O + (H + D) \cdot \log|\Sigma|)$.



Jak velký může být výstup? Obecně až $\mathcal{O}(H^2)$). Extrémně velký výstup je možné vygenerovat například slovníkem obsahujícím všechny prefixy slova `AAAA...A` délky H a senem taktéž `AAAA...A` délky H. Automat pak hlásí výskyt pro každé podslovo, kterých je $\mathcal{O}(H^2)$.

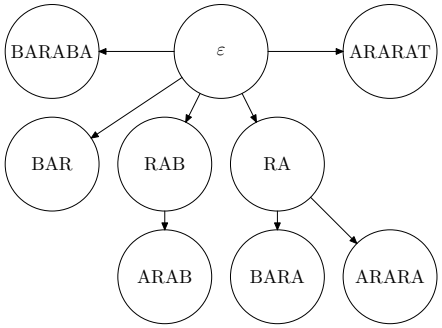
Pokud nám stačí u každého slova jen počet výskytů, nemusíme zoufat – závislost na počtu výskytů umíme odstranit.

Použijeme trik – na každé pozici započítáme pouze nejdelší jehlu, která tam končí (u každé jehly si budeme udržovat čítač). Nebudeme tedy v každém kroku poskakovat po zkratkách až do aleluja, ale maximálně jednou. Díky tomu nám z časové složitosti zmizí velikost výstupu.

V našem příkladu se senem `BARABARARAT` tedy na konci budeme mít uloženo, že `ARAB` se vyskytnul 1×, `ARARA` 1×, `ARARAT` 1×, `BAR` 2×, `BARA` 2× a `BARABA` 1×. `RA` a `RAB` nemají hlášený žádný výskyt.

Nyní si zkonstruujeme strom jenom ze zkratek a pro každý vrchol spočítáme součet celého jeho podstromu.

Tedy po přepočtu bude mít `RA` 3 výskyty a `RAB` 1 výskyt; celkový počet výskytů pak bude 12.



Poznámky

- Dalším krokem po KMP a Aho-Corasickové jsou konečné automaty a regulární výrazy, o kterých jsme měli seriál ve 23. ročníku.
- Není moc rozumné snažit se implementovat Aho-Corasickovou v rozumné době například při soutěži, pokud tento algoritmus nemáte opravdu pod kůží. Radši zkuste použít hešování, pokud budete něco takového potřebovat.

Cvičení

- Redukci o velikost výstupu můžeme provést i pro případ, kdy výstup nebudeme vypisovat, ale stačí nám mít jej uložený v paměti. Vymyslete vhodnou úpravu triku s čítačem.
- Zkuste si implementovat Aho-Corasickovou vlastnoručně ve svém oblíbeném jazyce, abyste si byli jisti, že doopravdy chápete všechny záludnosti tohoto algoritmu.

Martin Böhm, Jan Matějka, Martin Mareš a Petr Škoda

Vzorová řešení třetí série čtyřiadvacátého ročníku KSP

24-3-1 Intervalové duplicity

Kuchařka na intervalové stromy, intervaly v názvu úlohy, dokonce nám i chodí dotazy na intervaly. „To prostě musí být intervalové stromy!“ Ale nejsou. Tato shoda náhod je jeden velký chyták. Je možné, že na tuto úlohu nějakým způsobem jdou napasovat intervalové stromy, ale rozhodně to nepatří k těm jednodušším řešením. Jaký tedy byl vzorový postup?

Celé řešení této úlohy je vlastně jen jeden velký trik. Nejdříve si všimneme, že pro dvojici čísel se stejnou hodnotou nás zajímá především interval, který má na krajích čísla z této dvojice… Pak o libovolném intervalu $[X, Y]$ řekneme, že je špatný, pokud v sobě obsahuje některý z těchto minimálních intervalů.

My dostaneme dotaz na interval $[L, P]$ a vše, co nás zajímá, je, jestli se v něm vyskytuje některý z minimálních špatných intervalů. Co kdybychom si pro každý možný pravý kraj intervalu $[L, P]$ předpočítali pozici A začátku nejbližšího levého minimálního intervalu $[A, B]$, který zároveň splňuje $B \leq P$? Pak bychom jen porovnali A a L. Pokud by $A < L$, tak by interval byl dobrý a v opačném případě by byl špatný. To bychom měli vyhráno!

Předpočítat si tyto hodnoty ale vůbec není těžké. Posloupnost projdeme zleva doprava a pro každou hodnotu si budeme pamatovat, kdy naposled jsme ji viděli. To si můžeme pamatovat například pomocí hešovací tabulky, nebo binárního vyhledávacího stromu. Ať už to bude cokoliv, říkejme tomu *mapa*. Dále si chceme pamatovat *poslední* minimální špatný interval nalevo od nás. Nyní pro všechny pozice k v posloupnosti zavoláme tyto příkazy:

```
poslední[k] = max(poslední[k-1], mapa[pole[k]])
mapa[pole[k]] = k
```

A to už vlastně máme hodnoty předpočítané. Teď jen stačí odpovědět na všechny dotazy.

Při použití binárního vyhledávacího stromu potřebujeme čas $\mathcal{O}(N \log N)$ na předpočítání a čas $O(Q)$ na zodpovězení dotazů, kde N je délka posloupnosti a Q je počet dotazů. Celkem tedy $\mathcal{O}(N \log N + Q)$. Při použití hešovací tabulky časová složitost závisí na použité hešovací funkci a průměrném množství vzniklých kolizí v tabulce. Tento rozbor složitosti zde vynecháme.

Paměťová složitost řešení je $\mathcal{O}(N)$. Potřebujeme si pamatovat konstantní množství informací ke každé hodnotě posloupnosti.

Ve vzorovém zdrojovém kódu je použita mapa z C++ knihovny STL, se kterou se pracuje stejně jako s hešovací tabulkou a vlastně stejně jako s asociativním polem, které používáte například v PHP.

Závěrem bych chtěl poznamenat, že na našich testovacích

vstupech prošla na plný počet bodů i některá řešení s kvadratickou časovou složitostí s dostatečným množstvím heuristik. Tohoto faktu si všiml Ondra Hübsch a my mu tímto děkujeme za upozornění.

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

Karel Tesař

24-3-2 Nemnoho počítačů

Nejdříve si uvědomíme, že rychleji než v $\mathcal{O}(N)$ čísla počítačů nesetřídíme, protože je potřebujeme alespoň načíst a vypsat, což rychleji než lineárně nejde. Současně určitě umíme čísla počítačů setřídit v $\mathcal{O}(N \log N)$, protože v tomto čase umíme setřídit obecnou posloupnost čísel pomocí třídících algoritmů, jako jsou Quicksort nebo Mergesort. Nejde to však rychleji?

V došlých řešeních se objevovaly dva možné přístupy, popíšeme si tedy oba. Prvním z nich je použití *asociativního pole*, neboli hešování.

Řešení pomocí hešování

V rychlosti tu popíšeme pouze základní principy, koho by to zaujalo, může se podívat do odpovídající kuchařky o hešování.

Základem fungování heše je *hešovací funkce*. Ta přiřazuje *klíčům* (textových řetězcům nebo velkým číslům podle toho, jakými hodnotami chceme heš indexovat) nějaká malá čísla z rozsahu 0 až $K-1$, pomocí nichž se již dá indexovat normální pole. Dobrým příkladem hešovací funkce pro velká čísla je například zbytek po dělení číslem K.

Když ale hešovací funkce přiřadí dvěma klíčům stejnou hodnotu, nastává *kolize*. V takovém případě je někdy nutné projít až celé pole a to trvá lineárně dlouho. Pro větší detaily se opět podívejte do kuchařky o hešování.[6]

Pokud se rozhodneme použít hešování, vytvoříme si asociativní pole o velikosti K, sloužící pro ukládání počtů jednotlivých typů počítačů. Číslo K zvolíme jako nějaké prvočíslo mezi $2 \log N$ a $4 \log N$ (takové prvočíslo mezi číslem a jeho dvojnásobkem určitě existuje, ale to si zde nebudeme dokazovat).

Proč právě takhle? Rozsah zhruba dvojnásobku počtu klíčů je rozumná volba z hlediska minimalizování počtu kolizí, ale současně pole ještě není příliš velké. A volba prvočísla je šikovná z hlediska hešovací funkce, která bude vracet zbytek po dělení K.

Poté již postupně procházíme vstupní posloupnost. Pokud se klíč odpovídající typu počítače v heši ještě nenachází, založíme ho, jinak ke stávajícímu počtu počítačů tohoto typu pouze přičteme jedničku.

Po načtení celého vstupu pak pole setřídíme, což nám vzhledem k jeho velikosti $\mathcal{O}(\log N)$ bude s použitím například Mergesortu trvat $\mathcal{O}(\log N \log\log N)$, což je méně než $\mathcal{O}(N)$. Poté již stačí jenom setříděné pole projít a u každého typu ho vypsat tolikrát, kolikrát byl na vstupu. To nám zabere lineární čas vzhledem k velikosti vstupu.

Paměťová složitost je úměrná velikosti pole, tedy $\mathcal{O}(\log N)$. Je ale časová složitost skutečně $\mathcal{O}(N)$? Vše záleží na volbě hešovací funkce a na číslech, která se vyskytnou na vstupu. V nejhorším případě může nastat u všech prvků heše kolize a zpracování kolize může stát až lineárně vzhledem k velikosti heše, tedy $\mathcal{O}(\log N)$ Časová složitost by tedy byla až $\mathcal{O}(N \log N)$.

Řešení pomocí vyhledávacích stromů

Druhým přístupem, který nám zajistí dobrou časovou složitost ve všech případech (i když to nebude tak dobré, jako $\mathcal{O}(N)$ u hešování v nejlepším případě), je použití vyhledávacích stromů. Vyvážený vyhledávací strom nám zaručuje přístup ke všem jeho prvkům v logaritmickém čase vzhledem k jeho velikosti.

Vyhledávací strom je strom s nějakou hodnotou v každém vrcholu. Pro každý vrchol platí, že všechny hodnoty v jeho levém podstromu jsou menší, než hodnota v daném vrcholu, a všechny hodnoty v pravém podstromu jsou zase větší, než hodnota v daném vrcholu.

Budeme potřebovat pouze přidávání do stromu, proto si popíšeme pouze to. Pro více detailů se podívejte do kuchařky o vyhledávacích stromech.[7] Přidávání je stejné jako vyhledávání. Začneme v kořeni a postupně se zanořujeme do levého nebo pravého podstromu (podle toho, jestli je hledaná hodnota menší nebo větší, než hodnota ve vrcholu), dokud nenarazíme na vrchol s hledanou hodnotou.

Nebo, pokud takový vrchol neexistuje a my ho chceme přidat, vytvoříme ho na daném místě jako levého nebo pravého syna vrcholu, kde jsme skončili. Při takovém přidání nám ale může strom degenerovat a může nám vzniknout až strom tvaru dlouhé lineární cesty. Pro zajištění podmínky přístupu ke všem prvkům v logaritmickém čase je nutné strom vyvažovat.

Vyvažování se provádí pomocí *rotací*. Prostě překořeníme nevyvážený podstrom za nějaký jeho vrchol tak, aby se hloubka levého a pravého podstromu vždy lišila maximálně o jedna. Zároveň samozřejmě nesmíme porušit uspořádání hodnot ve vrcholech – výsledkem rotace je opět binární vyhledávací strom.

Takovým stromům se říká *AVL stromy*, koho rotace zajímají více, nechť se opět začte do kuchařky o vyhledávacích stromech.

Nám stačí vědět pouze to, že jedna rotace trvá konstantně dlouho a při jednom vyvažování provedeme maximálně tolik rotací, kolik je hloubka stromu, tedy logaritmicky vzhledem k jeho velikosti.

Nyní tedy máme datovou strukturu, do které můžeme v logaritmickém čase přidávat libovolné hodnoty. A navíc, pokud poté budeme strom procházet zleva (v každém vrcholu nejdříve zpracujeme levý podstrom, pak vrchol a nakonec pravý podstrom), dostaneme rovnou setříděnou posloupnost těchto hodnot. Procházení stromu zleva trvá lineárně vzhledem k jeho velikosti.

Základní idea programu tedy bude stejná jako v předchozím případě. Budeme načítat posloupnost na vstupu a každý prvek se pokusíme vložit do stromu. Pokud se už ve stromu tento typ počítače nachází, jenom ke stávajícímu počtu počítačů tohoto typu přičteme jedničku, jinak tento typ počítače založíme.

Velikost stromu bude stejná, jako je počet typů počítačů, tedy $\mathcal{O}(\log N)$ a přidání prvku do stromu včetně násled-

[6] `http://ksp.mff.cuni.cz/viz/kucharky/hesovani`
[7] `http://ksp.mff.cuni.cz/viz/kucharky/stromy`

ného vyvážení bude stát $\mathcal{O}(\log \log N)$. Zpracování vstupu nám tedy zabere $\mathcal{O}(N \log \log N)$. Nakonec už pouze projdeme strom zleva a vypíšeme každý typ tolikrát, kolikrát se objevil na vstupu.

Paměťová složitost je v tomto případě také $\mathcal{O}(\log N)$. Nejvíce času nám zabere zpracování celého vstupu do stromu, tedy celková časová složitost je $\mathcal{O}(N \log \log N)$.

Program (C++):
`http://ksp.mff.cuni.cz/viz/24-3-2.cpp`

Jirka Setnička

24-3-3 Párování znalců

Našou úlohou je zistiť počet hrán v maximálnom párovaní v strome.

Hranu, ktorá obsahuje vrchol, ktorý má len jedného suseda (a to druhý vrchol, ktorý patrí tej istej hrane) nazvyme *listová hrana*.

Algoritmus je jednoduchý. Vezmeme si ľubovoľnú listovú hranu a odoberieme zo stromu oba vrcholy patriace tejto hrane (odobrať vrchol znamená aj odobrať všetky hrany ktorým patrí). Za takýto krok si započítáme jednu hranu do párovania (tú listovú), tie čo sme odobrali spolu s vrcholom, ktorý v nájdenej listovej hrane nebol list, do párovania samozrejme nepočítame. Skončíme, keď už nemáme čo odobrať. To, že nám pri odoberaní listových hrán môže vzniknúť les, ničomu nevadí.

Prečo to funguje? Tak, predpokladajme, že e je listová hrana a M je nejaké maximálne párovanie v strome. Takže platí, že ak do M pridáme ľubovoľnú hranu, ktorá v M ešte nie je, tak M už nebude párovanie. Nech M neobsahuje listovú hranu e. Ak hranu e do M pridáme, tak práve jeden vrchol bude obsiahnutý v dvoch hranách párovania (lebo e je listová). Takže môžme odobrať nejakú z hrán v M a dostať maximálne párovanie, ktoré obsahuje e.

Môžeme si to predstaviť takto: vždy, keď nájdeme nejakú listovú hranu, tak na základe predchádzajúceho odstavca vieme, že existuje maximálne párovanie (v tom, čo nám zo stromu na vstupe ešte zostalo) také, že obsahuje nájdenú hranu a preto môžeme vrcholy tejto hrany odobrať. A teda správnosť algoritmu je dokázaná, pretože vieme, že v každom kroku nič nepokazíme.

Algoritmus môžeme implementovať ako prehľadávanie stromu do hĺbky metódou postorder (s vrcholom niečo vykonáme až potom, čo sme dokončili prácu s jeho synmi): vždy, keď sa pozrieme na vrchol (to je už po tom, čo sme sa pozreli na všetkých jeho synov), tak zistíme, či nemá nejakého nespárovaného syna. Ak áno, tak vrchol spárujeme s ľubovoľným nespárovaným synom.

Čo sa časovej zložitosti týká, tak tá je lineárna vzhľadom na počet vrcholov, čiže $\mathcal{O}(n)$, kde n je počet vrcholov. Pamäťová zložitosť je na tom rovnako.

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

24-3-4 Návrat do podposloupnosti

Po přečtení zadání není těžké si uvědomit, že naším úkolem je vyškrtnout souvislou část posloupnosti tak, abychom dostali co nejdelší souvislou rostoucí podposloupnost.

Než začneme cokoliv vymýšlet, tak si uvědomíme, že by se nám pro každý prvek mohlo hodit znát, jak dlouhá souvislá rostoucí podposloupnost v něm končí a jak dlouhá souvislá rostoucí podposloupnost v něm začíná. Těmto hodnotám budeme říkat prefixy a sufixy prvků. Prefixy spočítáme tak, že posloupnost projdeme zleva doprava a budeme si průběžně pamatovat, jak dlouhá je poslední rostoucí část. Obdobně, při průchodu zprava doleva, spočítáme sufixy.

Teď teprve nad úlohou začneme přemýšlet. Pokud bychom nic neškrtli, tak odpovědí bude nejdelší prefix. Pokud vyškrtneme nějaký úsek $[a, b]$, tak se nám situace může zlepšit pouze v místě škrtání, pokud spolu můžeme spojit prefix končící v bodě $a - 1$ se sufixem začínajícím v bodě $b + 1$. Nikde jinde se nám situace nezmění.

Pro kvadratické řešení nám tedy stačí jen vyzkoušet všechny možné úseky a pro každý se podívat, jak dlouhá posloupnost vznikne po jeho vyškrtnutí. Pak jen vezmeme maximum ze všech hodnot, které jsme takto našli, a všech prefixů a řešení vypíšeme. Toto řešení má časovou složitost $\mathcal{O}(n^2)$. Zkoušíme $\mathcal{O}(n^2)$ úseků a z každého získáme zlepšení v konstantním čase.

Jde to ale řešit i lépe. Pokud si určíme, kde škrtaný úsek bude končit, tak budeme chtít efektivně zjistit, kde má škrtaný úsek začínat, abychom vytvořili co nejdelší souvislý rostoucí úsek. Jinými slovy nás zajímá, k jakému nejdelšímu prefixu, umístěnému směrem nalevo, jsme schopní tento sufix napojit.

K tomu využijeme datovou strukturu jménem intervalový strom, který je popsán v kuchařce ke třetí sérii. Konkrétně se nám bude hodit intervalový strom pro maxima, na začátku inicializovaný na samé 0. Jak přesně nám pomůže?

Hodnoty v posloupnosti si seřadíme podle velikosti od nejmenších po největší. Nyní postupně od nejmenších prvků budeme provádět tyto operace (pozor, první operace bez druhé nedává smysl):

1) Zeptáme se stromu na maximum na intervalu jedna až původní pozice prvku.
2) Do intervalového stromu na původní pozici prvku uložíme velikost rostoucího prefixu končícího tímto prvkem.

Vždy, když se intervalového stromu ptáme na maximum nějakého intervalu, tak v něm máme uložené prefixy všech menších prvků, tedy jediných prvků, na které má smysl se ptát. Tedy dostáváme správné odpovědi. A to je vlastně celé.

Toto řešení má časovou složitost $\mathcal{O}(n \log n)$. Prvky třídíme a pokládáme $\mathcal{O}(n)$ dotazů intervalovému stromu, kde každý zabere čas $\mathcal{O}(\log n)$.

Řešení pomocí intervalového stromu můžete najít ve zdrojovém kódu. Pro jednoduchost zápisu program jen zjišťuje, jak dlouhou posloupnost umíme vytvořit. Konkrétní posloupnost dostaneme tak, že intervalový strom upravíme, aby si pamatoval i to, odkud maximum pochází.

Program (C++):
`http://ksp.mff.cuni.cz/viz/24-3-4.cpp` *Karel Tesař*

24-3-5 Součin zlomků

Ukážeme si celkem tři postupně se zlepšující řešení. Stojí možná za poznámku, že jen pár řešitelů přišlo na první z nich a nikdo na druhé ani na třetí. Navíc se objevil netriviální počet řešitelů, kteří vzali příklad s jednobajtovými čísly za součást zadání, což je pochopitelně stálo nemalou část bodů. A nyní už k věci.

První varianta

Lze snadno nahlédnout, že rozložením čitatelů a jmenovatelů na prvočinitele a následným pokrácením dostaneme zlomek v základním tvaru. K rozkladu (neboli faktorizaci) použijeme pole prvočísel předpočítané známým algoritmem Eratosthenova síta. Ten ostatně budeme potřebovat i v následujících dvou řešeních.

Nejdříve tedy přečteme celý vstup a vybereme maximum M ze všech čitatelů a jmenovatelů. M nastavíme jako horní mez pro Eratosthenovo síto. Sítem získáme pole prvočísel, kde si následně u každého prvočísla budeme udržovat hodnotu jeho exponentu ve výsledku. Tu zjistíme tak, že znovu procházíme vstup a každého z čitatelů, resp. jmenovatelů rozkládáme na prvočinitele a podle toho zvyšujeme, resp. snižujeme příslušný exponent o jedničku.

Tento algoritmus běží v čase $\mathcal{O}(M \log\log M + N \cdot K)$. Z toho $\mathcal{O}(M \log\log M)$ nás stojí síto (viz dodatek), $\mathcal{O}(N \cdot K)$ trvá rozklad na prvočinitele (K označíme počet prvočísel menších než M a pro každé z $2N$ čísel na vstupu projdeme v nejhorším všechna prvočísla). Pokud jako výstup chceme skutečného čitatele a jmenovatele, nejen jeho faktorizaci, musíme ještě započíst čas na umocňování. Ten se dá snadno shora odhadnout logaritmem maximální hodnoty D datového typu, tedy $\mathcal{O}(\log D)$.

Časová složitost je tedy $\mathcal{O}(M \log\log M + N \cdot K + \log D)$.

Druhá varianta

Eratosthenova síta se nejspíš nezbavíme, takže se zaměříme na druhou část algoritmu, a to na prvočíselný rozklad. Upravíme síto tak, aby si u složených čísel pamatovalo nejen to, že jsou vyškrtnutá, ale také některé z prvočísel, jimiž jsme je škrtli. Přesněji řečeno, když v sítu vyškrtáváme k-tý násobek prvočísla p, poznamenáme si do prvku pole $P[k \cdot p]$ číslo p.

K čemu je nám to dobré? Ve chvíli, kdy potřebujeme faktorizovat nějaké číslo i, podíváme se do $P[i]$ a tam najdeme jeden z faktorů. Tím i vydělíme a proces opakujeme tak dlouho, až v $P[i]$ bude 0. Faktorizace každého čísla nám nyní zabere $\mathcal{O}(\log M)$ (zkuste si rozmyslet, proč). Celková časová složitost tudíž klesla na $\mathcal{O}(M \log\log M + N \log M + \log D)$.

Třetí, nejlepší varianta

Opět využijeme vhodného předpočítání. Než spustíme síto, spočítáme si pro každé číslo od 1 do M hodnotu $C[i]$, která se rovná rozdílu počtu výskytů i v čitatelích a jmenovatelích. Kdykoliv pak v sítu vyškrtáváme násobky nějakého prvočísla p, posčítáme $C[i]$ všech vyškrtaných čísel a hned víme, kolikrát se prvočíslo vyskytuje ve výsledku. Jen přitom musíme dávat pozor na ta i, která jsou dělitelná vyšší mocninou p. Ta musíme započítat vícekrát.

Kdybychom neošetřili vyšší mocniny, trval by celý algoritmus $\mathcal{O}(N)$ pro výpočet pole C a $\mathcal{O}(M \log\log M)$ pro síto. Vyšší mocniny nám ale ve skutečnosti algoritmus nezpomalí. Pokud vyškrtáváme násobky prvočísla p, budou mě první mocniny stát n/p, druhé n/p^2, atd., což není nic jiného, než geometrická řada se součtem $\mathcal{O}(n/p)$. Celkově tedy dostáváme časovou složitost $\mathcal{O}(M \log\log M + N + \log D)$.

Paměťová složitost všech tří řešení je $\mathcal{O}(M + N)$.

Program (C):
`http://ksp.mff.cuni.cz/viz/24-3-5.c`

Jan Bok

Složitost Eratosthenova síta

Eratosthenovo prvočíselné síto je jedním z vůbec nejstarších známých algoritmů (Eratosthenés z Kyrény žil ve 3. století př. n. l. a objevil ledacos zajímavého, například docela přesně spočítal velikost Země). Ovšem teprve v historicky nedávné době se matematici naučili spočítat, jakou má toto síto časovou složitost. Pojďme to také zkusit.

Uvažujme následující přímočarou implementaci síta:

```
for (int p=2; p<=n; p++)
   if (!sito[p])
      for (int j=2*p; j<=n; j+=p)
         sito[j] = 1;
```

Většinu času jistě trávíme ve vnitřním cyklu. Pokud zrovna vyškrtáváme násobky prvočísla p, projdeme jich $\lfloor n/p \rfloor$. Označíme-li všechna nalezená prvočísla $p_1 < p_2 < \ldots < p_k$, můžeme složitost celého síta zapsat jako $\mathcal{O}(n/p_1 + \ldots + n/p_k) = \mathcal{O}(n \cdot s)$, kde $s = 1/p_1 + \ldots + 1/p_k$, tedy součet převrácených hodnot všech prvočísel od 1 do n.

Přesný vzorec pro s není znám, ale ukážeme, jak hodnotu s omezit shora.

Nejprve to zkusíme poměrně hrubě: doplníme do součtu i převrácené hodnoty ostatních čísel. Tedy $s \leq 1/1 + 1/2 + 1/3 + \ldots + 1/n$. Tomuto součtu se říká n-té harmonické číslo a značí se H_n. Za chvíli dokážeme, že $H_n = \mathcal{O}(\log n)$, takže síto doběhne v čase $\mathcal{O}(n \log n)$.

Aby se nám H_n počítalo snáz, budeme předpokládat, že n je mocnina dvojky. (Pokud by nebylo, prostě ho zaokrouhlíme na nejbližší vyšší mocninu dvojky n', čímž nevzroste víc než dvojnásobně. Dostaneme $H_n \leq H_{n'} = \mathcal{O}(\log 2n) = \mathcal{O}(\log n)$.)

Zlomky v harmonickém součtu rozdělíme na bloky po mocninách dvojky:

$$H_n = \left(\frac{1}{1}\right) + \left(\frac{1}{2}\right) + \left(\frac{1}{3} + \frac{1}{4}\right) + \left(\frac{1}{5} + \frac{1}{6} + \frac{1}{7} + \frac{1}{8}\right) + \ldots.$$

V i-tém bloku se tedy nacházejí čísla od $1/(2^{i-1}+1)$ do $1/2^i$. Blok tudíž obsahuje 2^{i-1} čísel a všechna jsou menší než $1/2^{i-1}$, takže součet bloku je nejvýše 1. (To platí i pro nultý blok 1/1, který jinak z pravidelné struktury vybočuje.)

Jelikož každý blok přispěje nejvýše jedničkou a bloků je $\mathcal{O}(\log n)$, platí $H_n = \mathcal{O}(\log n)$.

Mimochodem, podobně můžeme dokázat, že každý blok přispěje aspoň 1/2, takže H_n můžeme logaritmem omezit i zespoda.

Logaritmický odhad součtu s je sice pěkný, ale ještě jsme vůbec nevyužili toho, že součet obsahuje jen prvočíselné členy. Podobně jako předtím budeme předpokládat, že n je mocnina dvojky, součet rozdělíme na bloky a omezíme shora součet jednoho bloku, řekněme toho mezi $n/2$ a n.

Nejprve spočítáme, kolik mezi $n/2$ a n leží prvočísel. Označme P množinu všech těchto prvočísel, tedy:

$$P = \{p \mid p \text{ je prvočíslo } \wedge\ n/2 < p \leq n\}.$$

Bude se nám hodit následující kombinační číslo:

$$C = \binom{n}{n/2} = \frac{n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot (n/2+1)}{(n/2) \cdot (n/2-1) \cdot \ldots \cdot 2 \cdot 1}.$$

Dokážeme následující nerovnosti:

$$(n/2)^{|P|} \leq \prod_{p \in P} p \leq C \leq 2^n.$$

Třetí nerovnost platí, jelikož libovolná n-prvková množina má celkem 2^n podmnožin a číslo C udává počet jejích $(n/2)$-prvkových podmnožin, takže musí být menší.

Druhou nerovnost dostaneme z toho, že každé prvočíslo $p \in P$ je dělitelem našeho C: v prvočíselném rozkladu čitatele se p vyskytuje právě jednou a ve jmenovateli ani jednou. A jelikož je C dělitelné všemi prvočísly z P, musí být dělitelné i jejich součinem, takže C je aspoň tak velké, jako tento součin.

První nerovnost je nejsnazší: všechna $p \in P$ jsou větší nebo rovna $n/2$.

Nyní nerovnosti složíme:

$$(n/2)^{|P|} \leq 2^n$$

a zlogaritmováním získáme:

$$(\log_2 n - 1) \cdot |P| \leq n,$$

z čehož vyjádříme počet prvočísel v množině P:

$$|P| \leq n/(\log_2 n - 1) = \mathcal{O}(n/\log n).$$

Dokázali jsme tedy, že mezi $n/2$ a n leží nejvýše $\mathcal{O}(n/\log n)$ prvočísel. Součet převrácených hodnot těchto prvočísel už omezíme snadno:

$$\sum_{p\in P} \frac{1}{p} \leq \sum_{p\in P} \frac{2}{n} \leq \mathcal{O}(n/\log n) \cdot \frac{2}{n} = \mathcal{O}(1/\log n).$$

Vraťme se k původní otázce, totiž k součtu převrácených hodnot všech prvočísel mezi 1 a n. Ta mezi $n/2$ a n, čili v posledním bloku, jsme už započítali, teď stejným způsobem započteme i bloky předcházející:

$$\begin{aligned} s &= \mathcal{O}\left(\frac{1}{\log n} + \frac{1}{\log n/2} + \frac{1}{\log n/4} + \ldots + \frac{1}{\log 2}\right) \\ &= \mathcal{O}\left(\frac{1}{\log n} + \frac{1}{(\log n) - 1} + \frac{1}{(\log n) - 2} + \ldots + \frac{1}{1}\right). \end{aligned}$$

To je ovšem až na konstantu skrytou v $\mathcal{O}$ rovno $(\log n)$-tému harmonickému číslu, čili $\mathcal{O}(\log\log n)$.

Dokázali jsme tedy, že Eratosthenovo síto doběhne v čase $\mathcal{O}(n \log\log n)$.

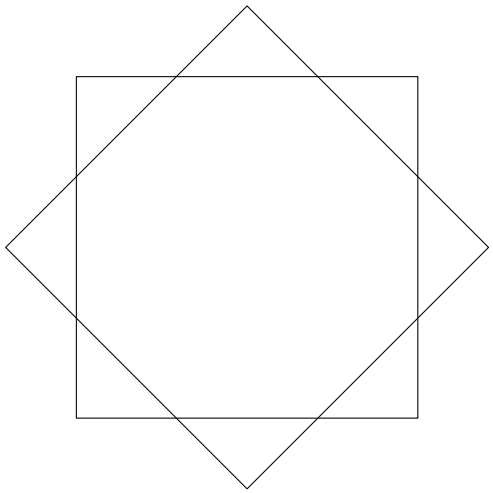
Martin „Medvěd“ Mareš

24-3-6 Průnik plánů

Úloha nebyla tak těžká, jak se na první pohled zdála. Stačilo nebát se a nenechat se ukolébat jednoduchostí vzorového obrázku.

Nejpřímočařejším řešením je uvědomit si, které vrcholy budou ohraničovat hledaný průnik. Vrchol jednoho mnohoúhelníka, který je uvnitř nebo na hranici druhého, bude určitě vrcholem průniku. Stejně tak průsečík stěn mnohoúhelníků bude vrcholem jejich průniku. Žádný jiný bod jistě nebude vrcholem průniku.

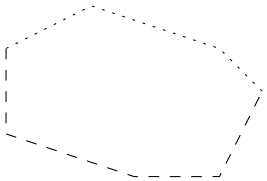
Na tomto místě většina z řešitelů zajásala a řekla, že maximální počet průsečíků stěn bude nějaká konstanta, nejčastěji čtyři. To ale není pravda. Obou typů vrcholů průniku může být $\mathcal{O}(n)$, kde n je počet vrcholů na vstupu. Lineární počet vrcholů uvnitř jednoho mnohoúhelníku si lze představit snadno – druhý mnohoúhelník bude celý uvnitř prvního. Lineární počet průsečíků stěn mají například dva soustředné pravidelné n-úhelníky. Pro šest průsečíků je to známá Davidova hvězda, pro osm dva pootočené čtverce.



I na základě této myšlenky by šel vymyslet hezký program. My si však ukážeme daleko jednodušší algoritmus.

Napřed nastiňme jeho myšlenku. Horní hranice průniku nebude výš než minimum z horních hranic obou mnohoúhelníků. Obdobně dolní hranice nebude níž než maximum.

Pro snazší popis si rozdělme konvexní obal na *horní* a *dolní obálku*. To jsou části, které vedou od nejlevějšího k nejpravějšímu vrcholu „horem“ a „spodem“. Pokud by byly dva vrcholy se stejnou x-ovou souřadnicí, berme vždy ten z nich, který má větší y-ovou souřadnici. Obálky si pamatujme v poli jako vrcholy seřazené podle x-ové souřadnice.



Rozdělení na horní a dolní obálky zvládneme snadno v lineárním čase, pokud máme vrcholy zadané už seřazené podle x-ové souřadnice nebo po obvodu konvexního obalu. Kdybychom měli vrcholy zadané jako nesetříděnou množinu, potřebovali bychom ještě třídit. Tento čas nebudeme počítat do výsledného času.

Kolmý průmět množiny bodů M na osu x je množina bodů na ose x takových, že když jimi vedeme kolmici, tak tato kolmice má neprázdný průnik s množinou M.

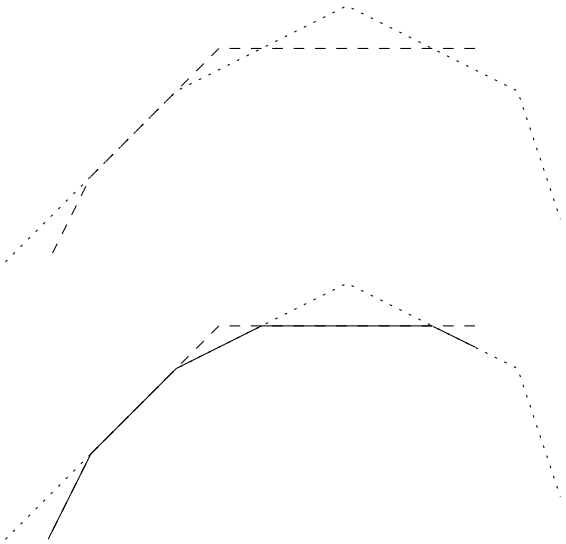
Pomocí horních obálek sestrojme horní lomenou čáru, která bude jejich minimem. Její kolmý průmět na osu x bude průnikem kolmých průmětů horních obálek. Na postup tvorby horní lomené čáry se mohou zkušení řešitelé geometrických úloh a znalci košťat dívat jako na zametání roviny.

Z obou horních obálek si udržujme jednu úsečku, se kterou budeme pracovat. Na začátku to budou první úsečky z obálek. Dokud mají prázdný průnik kolmých průmětů na osu x (tedy dokud neexistuje přímka kolmá na osu x, která má s oběma úsečkami společný aspoň jeden bod), nahradíme úsečku s menší x-ovou souřadnicí za následující v její obálce.

Dokud je průnik kolmých průmětů pracovních úseček neprázdný, přidáváme do horní lomené čáry hraniční body části jedné úsečky, která je pod druhou, nebo s ní splývá. Jakmile dojdeme na konec některé z našich pracovních úseček, vezmeme z její obálky další. Zjišťování, která část jedné úsečky je pod druhou, nebo s ní splývá, zabere konstantní čas.

Snadným rozborem případů nahlédneme, že do horní lomené čáry přidáme nejvýš dvě úsečky v každém pásu kolmém na osu x a vyhraničeném průnikem jejich kolmých průmětů. Takových úseků je lineárně s počtem úseček v obou obálkách.

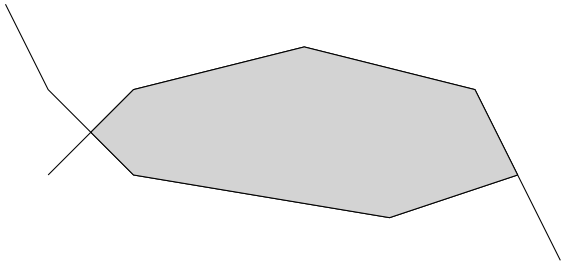
Lomenou čáru si ještě „uklidíme“ – odebereme z ní vrcholy, které jsou na spojnici dvou sousedních vrcholů. Jak výrobu, tak uklízení lomené čáry stihneme v čase $\mathcal{O}(n)$. Obdobně vyrobíme i dolní lomenou čáru, ale nesmíme zapomenout, že v tomto případě hledáme lomenou čáru, která vede po maximu z obálek.



Obdobným zametením, jako když jsme tvořili lomené čáry, určíme hranice oblasti, kde je horní lomená čára nad dolní. Můžeme si všimnout, že to bude souvislá oblast. Průnik konvexních množin je totiž konvexní množina. Případný důkaz plyne z definice – množina bodů M je konvexní, právě když pro každé dva body $x, y \in M$ leží i celá úsečka xy v M. Pokud x, y náleží průniku konvexních množin, náleží tam i jimi daná úsečka, protože x, y leží v průniku, takže musely ležet ve všech konvexních množinách, stejně jako jimi daná úsečka.

Obě lomené čáry musí mít stejnou x-ovou souřadnici začátku (a symetricky i konce). Je to dáno tím, že jejich kolmý průmět na osu x je roven průniku kolmých průmětů zadaných konvexních mnohoúhelníků na osu x. Lomené čáry se musí dvakrát protnout nebo dotknout. Pokud by se nedotkly, znamenalo by to, že minimum z horních obálek je větší než maximum z dolních. Ale horní obálka se v konvexním mnohoúhelníku vždy dotýká dolní.

Postupujeme po lomených čarách a část, kde horní je nad spodní, si zapamatujeme a vypíšeme. Musíme vypsat i případné průsečíky úseček tvořících lomené čáry. Opět stihneme v lineárním čase.



Dokažme ještě korektnost. Pokud leží bod v námi vypsané oblasti, jeho x-ová souřadnice je z průniku kolmých průmětů zadaných konvexních mnohoúhelníků na osu x. Navíc leží pod minimem z horních obálek a nad maximem z dolních obálek. Tedy leží v průniku oněch konvexních mnohoúhelníků. Naopak, pokud bod leží v průniku, musí ležet i ve vypsané oblasti.

Celkem tedy časová složitost algoritmu je $\mathcal{O}(n)$ (bez třídění, které není potřeba, pokud jsou vstupem body v jejich pořadí na konvexním obalu). Paměťová složitost je také lineární, protože si nepamatujeme víc než konstantně mnoho lineárně velkých polí.

Karel Král

24-3-7 Mazání závorek

Predpokladajme, že N je párne, pretože v opačnom prípade nemá význam uvažovať o správnosti uzátvorkovania a podobne musí platiť, že $K \leq N/2$.

Základnou myšlienkou pri riešení tejto úlohy je použiť zásobník k overeniu správnosti uzátvorkovania. Otváracie zátvorky postupne ukladáme do zásobníka. Ak narazíme na uzavieraciu zátvorku, tak ak je zásobník prázdny (momentálne v ňom nie sú otváracie zátvorky) alebo typ otváracej zátvorky na vrchu zásobníka sa nezhoduje s typom uzavieracej zátvorky, potom uzátvorkovanie určite nie je správne.

V prípade, že nám v zásobníku po vyčerpaní zátvoriek ostanú ešte nejaké otváracie, je uzátvorkovanie nesprávne. Inak ho môžeme prehlásiť za správne.

Budeme teda postupne čítať vstup. Ak je zátvorka na vstupe otváracia, tak ju vložíme na vrch zásobníka. Ak je uzavieracia, tak rozlíšime nasledujúce možnosti:

- Zásobník je prázdný.
- Na vrchu zásobníka je príslušná otváracia zátvorka.
- Na vrchu zásobníka je otváracia zátvorka iného typu.

V prvom prípade je nutné skontrolovať správnosť uzátvorkovania, v ktorom odignorujeme zátvorky typu práve spracovávanej uzavieracej zátvorky (je jednoduché si rozmyslieť, že stačí odignorovať len tento jeden typ). Podobne spravíme v treťom prípade, avšak musíme navyše skontrolovať správnosť uzátvorkovania, v ktorom odignorujeme zátvorky typu otváracej zátvorky na vrchu zásobníka. (opäť je jednoduché si rozmyslieť, že stačí kontrolovať tieto dva typy). V druhom prípade odoberieme otváraciu zátvorku z vrchu zásobníka.

Ak nakoniec ostane zásobnik prázdny, vieme, že je všetko v poriadku a môžeme prehlásiť uzátvorkovanie za správne. Inak môžeme skúsiť skontrolovať správnosť uzátvorkovania, v ktorom odignorujeme zátvorky typu otváracej zátvorky na vrchu zásobníka. Časová zložitosť je linéarna vzhľadom na dĺžku vstupu.

Program (C++):
`http://ksp.mff.cuni.cz/viz/24-3-7.cpp`

Peter Zeman

24-3-8 Sčítáme hry s panem Conwayem

Úkol 1: Maze

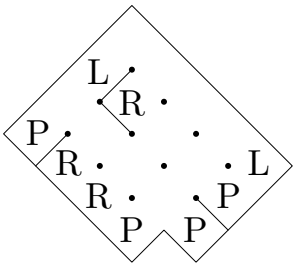
Jednoduchá hra Maze spočívající v posouvání žetonu po plánu byla prostým cvičením na definice her prohraných, vyhraných, her levého a pravého (tedy tříd P, V, L a R). Řešení úkolu potěšila, chyb nebylo mnoho a většinou zřejmě z nepozornosti.

Aby nějaká počáteční pozice žetonu mohla být označena správnou třídou, je třeba určit, jak dopadnou pozice, kam

z ní lze táhnout (ne vždy nutně všechny, ale hodí se to). Proto bylo vhodným postupem označovat políčka odspodu.

Jako první bylo možné zařadit do třídy P políčka, v nichž nemá žádný hráč žádný tah. Dále pozice žetonu, v nichž jeden hráč nemá tah a druhý může zahrát do prohrané pozice, patří jasně do třídy L nebo R (podle toho, kdo má tah).

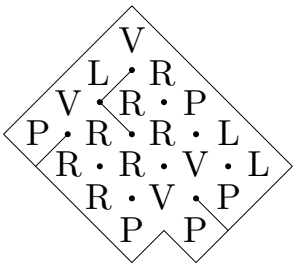
Dostaneme se tak k tomuto částečnému mezivýsledku (písmena tříd vkládáme pro jednoduchost přímo na políčka ve hře, jak to ostatně dělala většina řešitelů):



Pak se odspodu určují políčka tak, že na každém se pro oba hráče zjistí, jestli mohou z této pozice vyhrát tahem do prohrané pozice nebo do jejich pozice (tj. pro levého do pozice L). Podle toho se určí třída, do níž náleží políčko.

Například tedy políčko prohrané pro začínajícího je to, z něhož vedou všechny tahy levého do pozic pravého nebo do pozic vyhraných a všechny tahy pravého do pozic levého nebo vyhraných. Na pozici levého má levý tah, kterým vyhraje, a pravý ne.

Výsledný plán se zařazenými políčky vypadá takto:



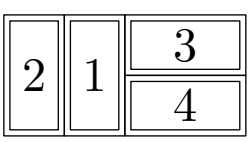
Úkol 2: dlouhé dominování

Prázdná mřížka o rozměrech $2 \times 4k$ (pro každé přirozené k) je vždy vyhraná pro pravého hráče pokládajícího vodorovná domina. (V tomto řešení se uvažuje mřížka se 2 políčky na výšku a se $4k$ na šířku. Pokud jste ve svém řešení měli mřížku otočenou, nevadilo to, jen je třeba prohodit L a R, levého a pravého, tedy prostě celou hru obrátit.)

Jednou z možností, jak to ukázat (či vůbec zjistit výsledek), bylo najít pro pravého vyhrávající strategii, když začíná i když nezačíná. My si ukážeme jednodušší argument založený na sčítání her, který také dává pravému strategii vedoucí k výhře.

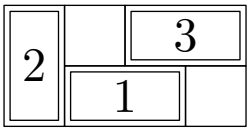
Nejprve rozebereme nejmenší případ, mřížku 2×4. Začne-li levý, může zahrát do sloupečku u kraje, nebo ve prostředku (ostatní dvě možnosti jsou symetrické k těmto). V obou případech pravý položí někam své domino, nyní má levý jen jeden tah a pravý také, jenže levý je na tahu, takže prohraje.

Na obrázku je jeden z případů, druhý si lze snadno domyslet:



Pokud začne pravý, zahraje doprostřed (je jedno, jestli nahoru nebo dolu). Levý položí domino doleva, nebo doprava (opět symetrické případy), načež pravý mu druhou možnost sebere položením posledního volného domina přes poslední volný sloupec (viz obrázek), čímž vyhraje.

Jelikož pravý vyhrál, není třeba zkoumat další možnosti jeho prvního tahu.



Mřížka 2×4 tedy náleží do třídy R. Mřížky $2 \times 4k$ pro $k > 1$ vyřešíme sčítáním tak, že je rozdělíme na k nepřekrývajících se bloků 2×4. Všimneme si, že levý nemůže zahrát do obou bloků současně, pravý však ano.

My ovšem chceme dokázat, že pravý vždy vyhraje, takže si můžeme dovolit ho omezit (pokud i s omezením stále vyhraje). Zakážeme mu tahy do dvou bloků současně, díky čemuž se bloky 2×4 stávají nezávislými hrami. Celá mřížka $2 \times 4k$ je pak jejich součtem.

Všechny bloky má vyhrané pravý, takže i jejich součet má vyhraný pravý (formálně použijeme indukci dle k, přičemž indukční krokem je sečtení mřížek $2 \times 4(k-1)$ a 2×4, jež obě náleží do R). A je to dokázáno.

Navíc doplníme strategii pro druhého na mřížce $2 \times 4k$. Začne-li levý, hraje pravý vždy do stejného bloku jako levý dle strategie pro mřížku 2×4. Pokud začne pravý, táhne doprostřed nějakého bloku (opět dle své vyhrávající strategie pro jeden blok) a pak hraje do stejného bloku jako předtím levý.

Úkol 3: rovnající se hry

Tento úkol se nakonec ukázal býti nejtěžším, soudě dle počtu správných řešení. Kdo nepřišel na následující vcelku jednoduchý důkaz, pustil se do rozboru případů podle toho, do jaké třídy náleží hry G a H. Jenže ten obsahuje spoustu skrytých záludností kvůli tomu, že G a H mohou vypadat o dost jinak, proto se mu budeme stručně věnovat.

Celkem zřejmě náleží G i H do stejné třídy (je to vidět z definice rovnosti, když přičteme prohranou hru, v níž nikdo nemá tah). Pokud je H ve třídě V nebo P, je $-H$ ve stejné třídě. Je-li H hra levého, je $-H$ hra pravého (a opačně pro hru pravého).

Pokud je G prohraná nebo vyhraná hra, pak máme součet dvou prohraných her, respektive dvou vyhraných (z jedné lze tahem udělat buď prohranou hru, nebo hru hráče, co táhl) a lze použít následující část (součet her z L a R) nebo důkaz ze seriálu (přičtení prohrané hry nemění výsledek).

Důkaz v seriálu však obsahoval chybu, za níž se hluboce omlouvám – vyhranou hru lze totiž tahem změnit nejen na prohranou hru, ale i na hru toho hráče, co táhl (je to vidět například v dominování na mřížce 2×2). Toto jsem tedy v řešeních toleroval a v seriálu opravil.

Nejtěžší byl rozbor, když G je hra levého (a analogicky pravého). Asi nejlepší bylo argumentovat stejnou převahou levého či pravého v G a H, neboli stejným počtem tahů pro levého i pravého, což však není lehké obecně spočítat (k úvahám těžších případů se místo dominování hodí spíše abstraktní zápis her).

Tolik v krátkosti k řešením rozborem případů. Obecně se nad ním bylo potřeba pořádně zamyslet, jestli je opravdu v pořádku.

Nyní o poznání jednodušší řešení. Z definice rovnosti G a H dopadnou hry $G + X$ a $H + X$ pro libovolnou hru X stejně. Speciálně to platí pro hru $-H$, tedy $G - H$ dopadne stejně jako $H - H$.

Rozbor, jak dopadne $H - H$ je už o dost jednodušší než rozbor $G - H$. Druhý hráč použije tzv. *zrcadlící* strategii. Táhne-li první do H, zahraje druhý do $-H$ ten samý tah, který tam z definice obrácené hry musí být. Obdobně, po tahu prvního do $-H$ hraje druhý do H.

Takto se druhý po prvním pořád opičí. Zároveň prvnímu musí dojít tahy dříve než druhému, díky čemuž druhý vyhrává. Tedy $H - H$ je prohraná hra a $G - H$ také.

Tím je hotovo. Nezbývá nic jiného než vám popřát hodně štěstí do dalšího řešení.

Pavel „Paulie“ Veselý

Výsledková listina třetí série dvacátého čtvrtého ročníku KSP

	řešitel	*škola*	*ročník*	*sérií*	*2431*	*2432*	*2433*	*2434*	*2435*	*2436*	*2437*	*2438*	*série*	*celkem*
0.					12	10	10	12	10	13	8	14	61,0	177,0
1.	Vojtěch Hlávka	GŠlapanice	3	13	12	7,5	10	12	5,5	13	5	12	58,5	163,8
2.	Martin Raszyk	G_Karvina	2	8	12	10		5	6	11	7,5		48,3	146,7
3.	Lukáš Ondráček	GVolgogrOS	3	3	12	9	8	5	2,5				43,8	129,8
4.	Jiří Eichler	SlovanGOL	4	10	12	8,5	10		2,5		8		41,5	128,0
5.	Michal Pokorný	SŠkybernHK	4	8	3	8,5	10	3	5,5	13	8	1	46,5	127,7
6.	Jerguš Greššák	ŠPMNDaGB	3	8	12	9	9		4			14	49,6	110,9
7.	Mark Karpilovsky	GJarošeBO	3	3	12	9	10				7		39,4	103,7
8.	Dominik Macháček	GLanškroun	3	3	1	4,5	8	2	4,5	4			34,6	100,2
9.	Alexander Mansurov	GNVPláníPH	3	7	9	9,5			6		5		33,0	95,7
10.	Michal Punčochář	GJírovcČB	2	4	12	10	10				8		40,0	89,5
11.	Jan Knížek	G_Strakon	1	4	6	4		2			7	7	36,5	84,2
12.	Martin Mirbauer	PORGPha	4	3	0					4			7,2	77,1
13.	Vojtěch Sejkora	SPSE_Pard	3	8		9	9		1,5			6	28,1	72,4
14.	Aneta Šťastná	GOmskPha	2	3		5		2	5			3,5	25,7	70,5
15.	Matej Lieskovský	GOmskPha	2	3		9	7		2,5	1		5	34,0	66,1
16.	Vojtěch Vašek	GHli	3	2	6	1	7		0	1	1	13	36,8	65,3
17.	Jan-Sebastian Fabík	GJarošeBO	2	3	12								12,0	64,0
18.	Štěpán Trčka	GSlavičín	1	2	6	7	5		2		5		37,1	61,1
19.	Ondřej Mička	GJírovcČB	3	11			10		4			12,5	26,5	58,0
20.	Lukáš Folwarczný	GKomHavíř	4	8	12								12,0	56,6
21.	Rastislav Rabatin	GJHroncaBA	3	2									0,0	51,6
22.	Dalimil Hájek	GKepleraPH	1	3		8			4	1	4		24,2	49,6
23.	Joel Jančařík	MensaG	4	1									0,0	38,8
24.	David Bernhauer	GZborovPH	4	4									0,0	38,5
25.	Martin Španěl	ArcibisGPH	3	1	3	4		2	3	2	6	6	37,1	37,1
26.	Jonatan Matějka	GJírovcČB	2	9									0,0	34,3
27.	Ondřej Cífka	GNAlejíPH	3	7	0	9	10					14	33,4	33,4
28.	Ondřej Hübsch	GArabskáPH	2	13	12								12,0	33,0
29.	Jan Hadrava	GZborovPH	4	5									0,0	32,6
30.	Jindřich Pilař	GBroumov	4	8									0,0	30,2
31.	Tereza Hulcová	GKlatovy	3	7		3	5				4		15,6	28,5
32.	Pavel Salva	VOŠŠumperk	2	3	3								5,7	19,4
33.	Pavel Kratochvíl	VOŠGSvětlá	4	14	6				3		7,5		14,3	17,4
34.	Jitka Fürbacherová	GKlatovy	3	5									0,0	15,7
35.	Jan Žárský	VSSKopř	1	1									0,0	14,1
36.	Zuzana Vozárová	GJHroncaBA	4	1									0,0	13,8
37.	Josefina Mádrová	GDobruška	4	2		6			2,5				13,4	13,4
38.	Vojtěch Polívka	GMikulášPL	4	1	6								9,5	9,5
39.	František Zajíc	G_Nymburk	-1	1									0,0	9,2
40.	Štěpán Šimsa	GJungmanLT	3	13									0,0	9,0
41.	Michal Hruška	GJirsíkaČB	4	1									0,0	7,6
42.	Matěj Židek	GBroumov	4	7									0,0	6,2
43.	Břetislav Hájek	GČesBrod	-2	3							1		2,1	5,9
44.	Juda Kaleta	GKlatovy	3	8									0,0	5,2
45.	Jan Pavlík	VOŠŠumperk	4	1									0,0	1,3