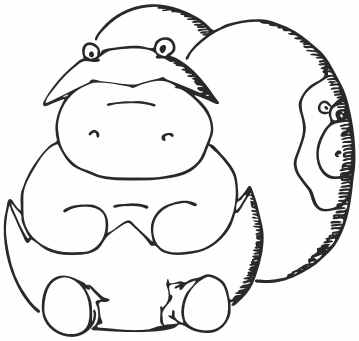
Milí řešitelé!

Vánoce jsou definitivně za námi. Stromeček dohořel, cukroví dojedli holubi a kapr zase přežil. Tak si oblékněte ten hnědý pletený svetr a ty vlněné ponožky, co jste našli pod stromečkem, a podívejte se, co vám ještě Ježíšek nadělil. Nemusíte pospíchat, protože na tuhle sérii máte skoro stejné množství času jako na přípravu pomlázky a barvení velikonočních vajíček.



Termín odeslání čtvrté série je tentokrát dne 21. března 2005. Řešení můžete odevzdávat jak elektronicky na *http://ksp.mff.cuni.cz/submit/*, tak klasickou poštou na známou adresu:

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

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

Zadání čtvrté série sedmnáctého ročníku KSP

17-4-1 Mandarinková zeď — 10 bodů

Veliký císař *Čching Ňamňam No-san* byl osvíceným vládcem Mandarínie. A jako osvícený vládce znal i zvyky a svátky jiných zemí. Ze všeho nejvíce se mu líbil jakýsi křesťanský svátek – Vánoce. Ten den totiž všichni dostávají mnoho dárků a on jako veliký osvícený panovník by jich určitě dostal opravdu mnoho. A tak se rozhodl, že se v Mandarínii budou Vánoce slavit také.

Prostí Mandaríni a Mandarínky nebyli ovšem jeho nápadem moc nadšeni, protože hlavní postava Vánoc *Santa Hood-san* měl podle *No-sana* chudým brát a jemu dávat. A proto se císař rozhodl, že si raději zkontroluje, jestli budou jeho poddaní Vánoce radostně slavit.

Kolem celé Mandarínie je postavena Velká Mandarinková zeď. Na této zdi jsou v pravidelných rozestupech strážní věže a v každé je jeden strážce. A *No-san* chce, aby právě tito strážci kontrolovali dodržování Vánoc. Práce strážců je ovšem velmi nudná, a tak každý strážce požaduje jistý počet *různých* medailí, aby byl se svou prací spokojen.

Císař chce všem strážcům vyhovět, ovšem rád by ušetřil, a tak se rozhodl použít co nejméně druhů medailí a rozdat je strážcům tak, aby každý dostal právě tolik různých medailí, o kolik si řekl, a navíc žádní dva sousední strážci neměli medaili stejného druhu (to, že nějaký strážce vidí, že jeho levý a pravý soused mají stejné druhy medailí, už císařovi nevadí). Poradíte?

Na vstupu dostanete počet strážních věží N a dále čísla a_1 až a_N, kde a_i reprezentuje počet medailí vyžadovaných strážcem číslo i. Vaším úkolem je zjistit, kolik nejméně druhů medailí je potřeba, aby každý strážce dostal, kolik chce různých druhů medailí, a aby žádní dva sousední strážci (sousedí spolu strážci i a $(i+1)$ a navíc ještě N a 1) nedostali ani jednu medaili stejného druhu.

Příklad: Pro 5 strážců a požadavky $2, 2, 2, 2, 2$ je minimální počet medailí 5.

17-4-2 Válicie — 10 bodů

Poté, co byl *Santa Hood-san* několikrát, zrovna když se jako na potvoru nedíval žádný strážce, zboulován, rozhodl se *No-san*, že bude muset prosazovat Vánoce ještě o něco důrazněji. A tak se rozhodl zavést v Mandarínii Vánoční policii, zvanou Válicie. (I když zlí jazykové tvrdí, že její název pochází spíš od toho, že si Válicisti stále válí šunky.)

Mandarínie je vlastně jedno velké město (obehnané zdí). A aby v něm císař udržel pořádek, rozhodl se postavit na některých křižovatkách stanice Válicie, a to tak, aby na konci každé ulice byla alespoň jedna stanice. Ale aby se nestalo, že na sebe v temných a strašidelných uličkách Mandarínie zaútočí Válicisté ze dvou stanic, které jsou na koncích jedné ulice, je třeba postavit stanice tak, aby na koncích každé ulice ve městě byla postavena právě jedna stanice. Císař je (jako obvykle) velký škudlil a minimalista, a tak by chtěl, aby stanic musel postavil co nejméně.

Na vstupu dostanete graf o N křižovatkách a M ulicích. Každá ulice je obousměrná a spojuje právě dvě křižovatky a žádné dvě ulice se mimo křižovatky nekříží (mimoúrovňově mohou). Vaším úkolem je zjistit, na kolika nejméně křižovatkách je třeba postavit Válicejní stanice tak, aby na koncích jedné ulice byla stanice právě jedna. Kromě počtu těchto křižovatek vypište i křižovatky, kde mají stanice stát. Pokud je řešení více, stačí vypsat libovolné řešení, pokud není řešení žádné, vypište odpovídající zprávu.

Příklad: Pro 6 křižovatek a 4 ulice spojující křižovatky $(1, 2)$, $(1, 5)$, $(3, 4)$ a $(1, 6)$ jsou potřeba dvě stanice na křižovatkách 1 a 3. Pro 3 křižovatky a 3 ulice $(1, 2)$, $(2, 3)$ a $(3, 1)$ stanice postavit nejde.

17-4-3 Phirma — 10 bodů

Poté, co dokázal *No-san* udržet v Mandarínii klid, se jeho pozornost přesunula k tomu, aby, když už Vánoce tak horko těžko zavedl, dostal odpovídající množství dárků. A protože mezi chudými už mnoho dárků hodných Velkého *No-sana* nebylo, rozhodl se je hledat jinde.

Snad nejznámější firmu v Mandarínii založili paní *Čestná* a pan *Jakobi*. A protože firma dělala čest svému jménu, byla také nejbohatší. Ledva to císař zvěděl, rozhodl, že mu Vánoční dárek zaplatí právě ona.

Firma *Jakobi-Čestná* musí dodat časově seřazený seznam výdajů a příjmů a císař určí daně podle toho, jak dlouhé bylo nejdelší časové období, kdy firma vykazovala zisk (takzvaný *kradit*). Ovšem účetní této firmy dokážou falšovat zisky opravdu bleskově, proto by *No-san* potřeboval zjistit požadované údaje co nejrychleji. A tak se obrátil na Vás.

Napište císaři program, který dostane na vstupu číslo N a dále posloupnost N celých čísel. Vaším úkolem je najít a vypsat nejdelší úsek (to je souvislá podposloupnost) takový, že součet čísel v tomto úseku je větší než nula. Pokud je takových úseků více, vypište libovolný s největším součtem.

Příklad: Pro posloupnost $(1, -2, -5, 1, 1, 1, 1, 1, 1, -1)$ má hledaný nejdelší úsek délku 7 a je to úsek $(1, 1, 1, 1, 1, 1, -1)$.

17-4-4 Antifrňákovník — 10 bodů

Mandarínům se nakonec *No-sanovy* klidné Vánoční svátky natolik znelíbily, že se rozhodli císaři utéct. Ovšem Mandarinková zeď obsazená strážemi s jejich záměry moc nesouhlasila. A tak si Mandaríni, aby se na útěk mohli pořádně připravit, založili Sportovní Klub Utek' & Utekl.

Po dlouhé debatě se členové SK Utek' & Utekl dohodli, že si postaví stroj *antifrňákovník,* který Mandarinkovou zeď rozbije, a oni budou moci utéct. Ale aby jejich práce nemohla být *No-sanovi* nikým z nich prozrazena, rozhodli se, že každý bude znát pouze část antifrňákovníku.

Jaké bylo nakonec jejich (ale ne naše) překvapení, když po sestavení celého přístroje zjistili, že nikdo neví, jak propojit jeho elektrické obvody. Dokonce ani neví, jaké konce drátů na jednom konci přístroje odpovídají koncům na straně druhé. A protože si *No-san* usmyslel, že právě Vy budete dalším sponzorem jeho Vánočních dárků, rozhodli jste se s dokončením antifrňákovníku pomoci.

Na obou koncích přístroje je N konců drátů očíslovaných 1 až N a dále zemnění, což je drát, o kterém jako jediném víte, jak je propojen. Můžete vlevo dráty na zemnění napojovat a odpojovat a na pravé straně můžete měřit, zda mezi koncem drátu a zemněním teče elektrický proud. Vaším úkolem je říci, jaký konec drátu na straně levé je spojen s jakým koncem na straně pravé. Bohužel se může stát i to, že některé dráty jsou přerušeny a nevedou nikam.

Máte tedy napsat program, který dostane na vstupu počet konců drátu N, vypisuje příkazy `+X` pro připojení levého konce X-tého drátu na zemnění, `-X` pro odpojení levého konce X-tého drátu od zemnění a `?X` pro změření napětí na pravém konci X-tého drátu a zemnění (na tyto dotazy odpovídá uživatel). Nakonec má vypsat, které levé konce drátů jsou připojeny na jaké pravé (nevodivé dráty nevypisujte vůbec, vodivé vypište všechny). S levým i pravým koncem drátu může být spojen nanejvýš jeden opačný konec. Na začátku není na zemnění připojen žádný konec levého drátu.

Příklad: Pro $N = 3$ a následující „rozhovor“

výstup programu	*uživatelský vstup*
`+1`	
`?1`	*Ano*
`-1`	
`+2`	
`?3`	*Ano*
`-2`	
`+3`	
`?2`	*Ne*

je správná odpověď $1 \to 1$, $2 \to 3$.

17-4-5 Jazykozpytec vrací úder — 15 bodů

Minule jsme si praktičtější úlohou odpočinuli od rozmanité teorie formálních jazyků, což nyní opět napravíme. Nastal čas, abychom od jednoduchých regulárních jazyků pokročili k složitějším *bezkontextovým jazykům.* Název těchto jazyků plyne ze souvislosti s gramatikami, jíž si ovšem ukážeme až v příštím díle seriálu. (Nezasvěceným doporučujeme, aby si prostudovali seriálové úlohy předchozích sérií.)

Zavedeme si podstatně mocnější výpočetní prostředek než byl konečný automat, tzv. *zásobníkový automat.* Ten vznikne tak, že starý známý konečný automat vybavíme zásobníkem, což je paměť potenciálně neomezené kapacity, ve které jsou naskládané symboly z nějaké pevné abecedy, ale je možno přistupovat vždy jen k symbolu, který je na vrcholu a buďto tento symbol odebrat a nebo přidat další nad něj.

Většina vlastností KA zůstane zachována, jen přechodová funkce se nyní bude počítat z kombinace aktuálního stavu, písmene na vstupu a symbolu na vrcholu zásobníku, tedy trojice (q, p, z). Funkce potom vrátí nový stav, do kterého má stroj přejít, a také posloupnost symbolů, kterými se nahradí dosavadní vrchol zásobníku. Oproti konečnému automatu, který v každém kroku musel ze vstupu přečíst právě jedno písmeno a z něj počítat přechodovou funkci, umí zásobníkový automat také načtení písmene vynechat, což si můžeme představovat jako načtení prázdného znaku λ (a přechodovou funkci tudíž počítat z trojice (q, λ, z)). Vše si zavedeme formálně.

Deterministický zásobníkový automat (DZA) M je sedmice

$$M = (Q, A, Z, \delta, q_0, z_0, F),$$

kde

- Q je konečná množina stavů,
- A je konečná vstupní abeceda,
- Z je konečná zásobníková abeceda (tedy symboly, které lze ukládat na zásobník),
- $\delta : Q \times (A \cup \{\lambda\}) \times Z \to Q \times Z^*$ je přechodová funkce,
- $q_0 \in Q$ je počáteční stav,
- $z_0 \in Z$ je počáteční zásobníkový symbol (čili symbol, který je při spuštění automatu uložen na zásobníku)
- $F \subseteq Q$ je množina přijímacích stavů.

Jeden výpočet přechodové funkce $\delta(q, a, z) = (q', w)$, neboli vykonání instrukce $(q, a, z) \to (q', w)$ pro $q, q' \in Q$, $a \in A \cup \{\lambda\}$, $z \in Z$ a $w \in Z^*$ znamená, že aktuální stav q se změní na q', ze vstupu se přečte písmeno a (anebo také nepřečte, v případě že $a = \lambda$) a aktuální symbol na vrcholu zásobníku z se nahradí posloupností w (třeba prázdnou či jednoprvkovou) zásobníkových symbolů (symboly zapsané vlevo se do zásobníku umístí níže než symboly vpravo). Pokud by bylo možné provést jak instrukci s $a = \lambda$, tak s konkrétním znakem, automat si vybere možnost s $a = \lambda$.

U zásobníkového automatu narozdíl od KA nepožadujeme, aby byla přechodová funkce δ definována pro všechny možné kombinace stavu, písmene a zásobníkového symbolu. Výpočet stroje se zastaví při dvou příležitostech: pro (q, a, z) není definována žádná instrukce nebo došlo k odstranění všech symbolů ze zásobníku. Všimněte si, že zásobníkový automat s (ne)vhodnou přechodovou funkcí (tedy vlastně programem) je již může zacyklit v nekonečné smyčce.

Zbývá si přesně říci, kdy je dané slovo přijato. Narozdíl od konečných automatů, u zásobníkových automatů můžeme stanovit hned dvě možnosti přijetí.

- Slovo $u \in A^*$ je přijímáno DZA M *koncovým stavem,* pokud se stroj M spuštěný na slovo u po konečném počtu kroků zastaví, celé slovo u je přečteno a M se nachází v přijímacím stavu. Jazykem DZA M přijímajícího stavem nazveme množinu všech slov, která M přijímá, značíme ji $L(M)$. Množině všech jazyků, které lze rozpoznávat DZA koncovým stavem (tedy všech takových jazyků L, že pro L existuje nějaký DZA M přijímající

stavem takový, že $L = L(M)$), se říká *deterministické bezkontextové jazyky,* budeme ji značit *BKS*.

- Slovo u je přijímáno DZA M *prázdným zásobníkem,* pokud se stroj M spuštěný na slovo u po konečném počtu kroků zastaví, celé slovo u je přečteno a zásobník stroje M je vyprázdněný. Jazykem DZA M přijímajícího zásobníkem nazveme množinu všech slov, která M přijímá, značíme ji $N(M)$. Množině všech jazyků, které lze rozpoznávat DZA prázdným zásobníkem (tedy všech takových jazyků L, že pro L existuje nějaký DZA M přijímající zásobníkem takový, že $L = N(M)$), se říká *bezprefixové bezkontextové jazyky,* budeme ji značit *BKZ*.

Příklad: Přijímat jazyk $L = \{0^n 1^n;\ n \in N\}$ zásobníkovému automatu nečiní potíže, narozdíl od konečného automatu (což jsme si dokázali v seriálové úloze druhé série). Setrojíme si tedy DZA M, který bude přijímat prázdným zásobníkem. Vstupní abeceda M bude $A = \{0, 1\}$, množina stavů $Q = \{l, p\}$, zásobníkové symboly $Z = \{z, 0\}$, počáteční stav bude l a počáteční zásobníkový symbol z, přijímací stavy F nejsou podstatné. Sadu instrukcí (čili přechodovou funkci δ) sestrojíme takto:

$$\delta(l, 0, z) = (l, 0) \quad \ldots \text{čte první symbol 0}$$
$$\delta(l, 0, 0) = (l, 00) \quad \ldots \text{čte další symbol 0}$$
$$\delta(l, 1, 0) = (p, \lambda) \quad \ldots \text{čte první symbol 1}$$
$$\delta(p, 1, 0) = (p, \lambda) \quad \ldots \text{čte další symbol 1}$$

Pokud bychom chtěli raději přijímat koncovým stavem $F = \{q_F\}$, pak první instrukci změníme na $\delta(l, 0, z) = (l, z0)$ (čili neodstraníme počáteční symbol hned na začátku) a přidáme ještě navíc jednu instrukci:

$$\delta(p, \lambda, z) = (q_F, \lambda) \quad \ldots \text{detekuje úspěšný konec}$$

Uvědomíme si, že každé DZA M přijímající prázdným zásobníkem lze převést na DZA přijímající koncovým stavem, jinými slovy tedy $BKZ \subseteq BKS$. Nový stroj bude mít jiný počáteční stav, řekněme q_0', a na začátku výpočtu původní počáteční symbol na zásobníku z podloží ještě jedním pomocným symbolem, řekněme z'. To se udělá například instrukcí $\delta(q_0', \lambda, z) = (q_0, z'z)$. Dále se pokračuje v původním programu, ale dodáme ještě speciální instrukce $\delta(q, \lambda, z') = (q_F, \lambda)$ pro každý $q \in Q$, které když uvidí na zásobníku z' (neboli zásobník původního stroje se vyprázdnil), přejdou do přijímacího stavu a vyprázdní zásobník (čímž skončí). Opačný převod však provést nelze.

Soutěžní úloha 1: Ukažte, že jazyk $L = \{0^n 1^m;\ 0 < n \leq m\}$ lze rozpoznávat DZA koncovým stavem, ale neexistuje DZA přijímající prázdným zásobníkem, který by L rozpoznával. Najděte příklad regulárního jazyka (tedy rozpoznatelného konečným automatem), který nelze rozpoznávat DZA prázdným zásobníkem (a pochopitelně zdůvodněte proč). *[6 bodů]*

Podobně jako u konečných automatů i u zásobníkových automatů můžeme velmi podobně zavést nedeterministickou verzi stroje (viz zadání druhé série). *Nedeterministický zásobníkový automat (NZA)* se od DZA liší tím, že přechodová funkce $\delta : Q \times (A \cup \{\lambda\}) \times Z \to P(Q \times Z^*)$, kde značením $P(X)$ rozumíme množinu všech podmnožin X, nyní vrací hned několik možností, jak může stroj zareagovat. Z nich si stroj jednu libovolnou vybere. Stejně tak pokud má stroj na výběr mezi čtením písmene a nebo jeho nečtením ($a = \lambda$), může si vybrat libovolnou možnost. Dané slovo u je přijato NZA M koncovým stavem resp. prázdným zásobníkem, pokud mezi všemi možnými výpočty nad u existuje alespoň jediný, po jehož konci je celé u přečteno a M se nachází v přijímacím stavu, resp. celé u je přečteno a zásobník je prázdný. DZA je tedy zjevně pouze speciálním případem takového NZA, kde přechodová funkce vrací vždy jednoprvkovou množinu.

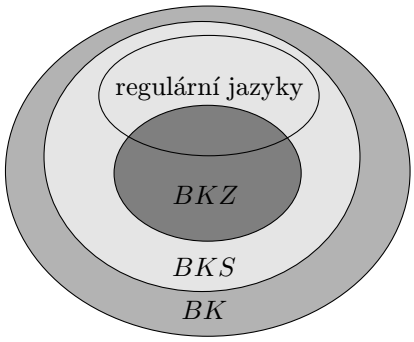
Soutěžní úloha 2: Narozdíl od DZA, přijímání koncovým stavem a přijímání prázdným zásobníkem je u nedeterministických zásobníkových automatů ekvivalentní. (Tedy ke každému NZA přijímajícímu prázdným zásobníkem lze zkonstruovat ekvivalentní NZA přijímajíci koncovým stavem a naopak.) Ukažte. *[3 body]*

Obě množiny jazyků přijímaných NZA stavem i NZA zásobníkem jsou tedy stejné. Těmto jazykům se říká *bezkontextové jazyky* a označíme si je *BK*.

Soutěžní úloha 3: Sestrojte nedeterministický zásobníkový automat, který umí rozpoznávat jazyk L všech palindromů nad abecedou $\{a, b\}$. (Palindrom je slovo, které se čte pozpátku stejně jako zepředu.) Také ukažte, že jazyk L nelze rozpoznávat DZA prázdným zásobníkem. Lze dokonce dokázat, že L se nedá rozpoznávat DZA koncovým stavem. To je ale o dost obtížnější a po vás to nechceme. Pokud ovšem někdo zašle *správný* důkaz i této varianty, štědrý bodový bonus ho jistě nemine. NZA jsou tedy výpočetně silnější stroje než DZA. *[6 bodů]*

Připomínáme, že vaše řešení by měla obsahovat matematicky správné a pokud možno i formální argumenty. Nicméně jelikož zásobníkové automaty jsou už poměrně složité stroje, nebudeme již takoví puntičkáři co se týká požadavků na matematický formalismus.

Po vyřešení všech soutěžních úloh vlastně sami ukážete tento vztah mezi bezkontextovými jazyky, deterministickými bezkontextovými jazyky a bezprefixovými bezkontextovými jazyky:



Recepty z programátorské kuchařky

Rozděl a panuj

Dnešní díl programátorské kuchařky se bude zabývat algoritmy založenými na metodě *Rozděl a panuj.* A tak by se slušelo začít tím, jaká je myšlenka této metody: Často se setkáme s úlohami, které lze snadno rozdělit na nějaké menší úlohy a z jejich výsledků zase snadno složit výsledek původní velké úlohy. Přitom menší úlohy můžeme počítat opět týmž algoritmem (zavoláme si ho rekurzivně), leda by již byly tak maličké, že dokážeme odpovědět triviálně bez jakéhokoliv počítání.

Zkrátka jak říkali staří římští císařové: Divide et impera. Uveďme si pro začátek jeden staronový příklad:

Quicksort

QuickSort (alias QS) je algoritmus pro třídění posloupnosti prvků. Už o něm byla jednou řeč v „třídící kuchařce“ v druhé sérii 16. ročníku KSP. Tentokrát se na něj podíváme trochu podrobněji a navíc nám poslouží jako ingredience pro další algoritmy.

QS v každém svém kroku zvolí nějaký prvek (budeme mu říkat *pivot*) a přerovná prvky v posloupnosti tak, aby napravo od pivota byly pouze prvky větší než pivot a nalevo pouze menší. Pokud se vyskytnou prvky rovné pivotu, můžeme si dle libosti vybrat jak levou, tak pravou stranu posloupnosti, funkčnost algoritmu to nijak neovlivní. Tento postup pak rekurzivně zopakujeme zvlášť pro prvky nalevo a zvlášť pro prvky napravo od pivota, a tak získáme setříděnou posloupnost.

Implementací QS je mnoho a mimo jiné se liší způsobem volby pivota. My si předvedeme jinou, než jsme ukazovali v třídící kuchařce (hlavně proto, že se nám od ní pak snadno budou odvozovat další algoritmy) a pro jednoduchost budeme jako pivota volit poslední prvek zkoumaného úseku:

```
{budeme třídit takováto pole}
type Pole=array[1..MaxN] of Integer;

{přerovnávací procedura pro úsek a[l..r]}
function prer(a:Pole; l,r:Integer):Integer;
var i,j,x,q:Integer;
begin
 {pivotem se stane poslední prvek úseku}
 x:=a[r];                          {hodnota pivota}
 i:=l-1;  {a[i] bude vždy poslední <= pivotovi}

 {samotné přerovnávání}
 for j:=l to r-1 do
  if a[j]<=x then  {právě probíraný prvek     }
  begin            {menší/rovný hodnotě pivota}
   Inc(i);         {pak zvyš ukazatel         }
   q:=a[j];        {a proveď přerovnání prvku }
   a[j]:=a[i];
   a[i]:=q;
  end;

 {nakonec přesuneme pivota za poslední <=}
 q:=a[r];
 a[r]:=a[i+1];
 a[i+1]:=q;
 prer:=i+1;      {vrátíme novou pozici pivota}
end;

{hlavní třídící procedura, třídí a[l..r]}
procedure QuickSort(a:Pole; l,r:Integer);
var m:Integer;
begin
 if l<r then            {máme ještě co dělat?}
 begin
  m:=prer(l,r);   {přerovnej, m pozice pivota}
  QuickSort(l,m-1);     {setřiď prvky napravo}
  QuickSort(m+1,r);      {setřiď prvky nalevo}
 end;
end;
```

Bohužel volit pivota právě takto je docela nešikovné, protože se nám snadno může stát, že si vybereme nejmenší nebo největší prvek v úseku (rozmyslete si, jak by vypadala posloupnost, ve které se to bude dít pokaždé), takže dostaneme-li posloupnost délky N, rozdělíme ji na úseky délek $N-1$ a 1, načež pokračujeme s úsekem délky $N-1$, ten rozdělíme na $N-2$ a 1 atd. Přitom pokaždé na přerovnání spotřebujeme čas lineární s velikostí úseku, celkem tedy $O(N+(N-1)+(N-2)+\ldots+1)=O(N^2)$.

Na druhou stranu pokud bychom si za pivota vybrali vždy *medián* z právě probíraných prvků (tj. prvek, který by se v setříděné posloupnosti nacházel uprostřed; pro sudý počet prvků zvolíme libovolný z obou prostředních prvků), dosáhneme daleko lepší složitosti $O(N \log N)$. To dokážeme snadno:

Přerovnávací část algoritmu běží v čase lineárním vůči počtu prvků, které máme přerovnat. V prvním kroku QS pracujeme s celou posloupností, čili přerovnáme celkem N prvků. Následuje rekurzivní volání pro levou a pravou část posloupnosti (obě dlouhé $(N-1)/2 \pm 1$); přerovnávání v obou částech dohromady trvá opět $O(N)$ a vzniknou tím části dlouhé nejvýše $N/4$. Zanoříme-li se v rekurzi do hloubky k, pracujeme s částmi dlouhými nejvýše $N/2^k$, které dohromady dají nejvýše N (všechny části dohromady dají prvky vstupní posloupnosti bez těch, které jsme si už zvolili jako pivoty). V hloubce $\log_2 N$ už jsou všechny části nejvýše jednoprvkové, takže se rekurze zastaví. Celkem tedy máme $\log_2 N$ hladin (hloubek) a na každé z nich trávíme lineární čas, dohromady $O(N \log N)$.

V tomto důkazu jsme se ale dopustili jednoho podvodu: zapomněli jsme na to, že také musíme medián umět najít. Jak z této nepříjemné situace ven?

- *Naučit se počítat medián.* Ale jak? Haf?
- *Spokojit se se „lžimediánem“:* kdybychom si místo mediánu vybrali libovolný prvek, který bude v setříděné posloupnosti v prostřední polovině (čili alespoň čtvrtina prvků bude větší a alespoň čtvrtina menší než on), získáme také složitost $O(N \log N)$, neboť úsek délky N rozložíme na úseky, které budou mít délky nejvýše $(1-1/4)\cdot N$, takže na k-té hladině budou úseky délek $\leq (1-1/4)^k \cdot N$, čili hladin bude maximálně $\log_{1-1/4} N = O(\log N)$. Místo $1/4$ by dokonce fungovala libovolná jiná konstanta, ale ani to nám nepomůže k tomu, abychom uměli lžimedián najít.
- *Recyklovat pravidlo* typu „vezmi poslední prvek“ a jen ho trochu vylepšit. To bohužel nebude fungovat, protože pokud budeme při výběru pivota hledět jenom na konstantní počet prvků, bude poměrně snadné přijít na vstup, pro který toto pravidlo bude dávat kvadratickou složitost, i když obvykle půjde dokázat, že takových vstupů je „málo“. [Také se to tak často dělá.]
- *Volit pivota náhodně* ze všech prvků zkoumaného úseku. K náhodné volbě samozřejmě potřebujeme náhodný generátor a s těmi je to svízelné, ale zkusme na chvíli věřit, že jeden takový máme nebo alespoň že máme něco s podobnými vlastnostmi. Jak nám to pomůže? Náhodně zvolený pivot nebude sice přesně uprostřed, ale s pravděpodobností $1/2$ to bude lžimedián, takže po průměrně dvou hladinách se ke lžimediánu dopracujeme (rozmyslete si, proč, nebo nahlédněte do loňského seriálu o pravděpodobnostních algoritmech). Proto časová složitost takovéhoto randomizovaného QS bude v průměru 2-krát větší, než lžimediánového QS, čili v průměru také $O(N \log N)$.

Jednoduše řečeno, zatímco fixní pravidlo nám dalo dobrý čas pro průměrný vstup, ale existovaly vstupy, na kterých bylo pomalé, randomizování nám dává dobrý průměrný čas pro všechny možné vstupy.

Hledání k-tého nejmenšího prvku

Nad QuickSortem jsme zvítězili, ale současně jsme při tom zjistili, že neumíme rychle najít medián. To tak nemůžeme nechat, a proto rovnou zkusíme vyřešit obecnější problém: najít k-tý nejmenší prvek (medián je to pro $k = \lfloor N/2 \rfloor$).

První řešení této úlohy se nabízí samo. Načteme posloupnost do pole, prvky pole setřídíme nějakým rychlým algoritmem a kýžený k-tý nejmenší prvek nalezneme na k-té pozici v nyní již setříděném poli. Má to však jeden háček. Pokud prvky, které máme na vstupu, umíme pouze porovnat, pak nedosáhneme lepší časové složitosti (a to ani v průměrném případě) než $O(N \log N)$ – rychleji prostě třídit nelze, důkaz můžete najít například v třídící kuchařce.

O něco rychlejší řešení je založeno na výše zmíněném algoritmu QuickSort (často se mu proto říká QuickSelect). Opět si vybereme pivota a posloupnost rozdělíme na prvky menší než pivot, pivota a prvky větší než pivot (pro jednoduchost budeme předpokládat, že žádné dva prvky posloupnosti nejsou stejné). Pokud se pivot nalézá na k-té pozici, je to hledaný k-tý nejmenší prvek posloupnosti, protože právě $k-1$ prvků je menších. Zbývají dva případy, kdy tomu tak není. Pakliže je pozice pivota v posloupnosti větší než k, pak se hledaný prvek nalézá nalevo od pivota a postačí rekurzivně najít k-tý nejmenší prvek mezi prvky nalevo. V opačném případě, kdy je pozice pivota menší než k, je hledaný prvek v posloupnosti napravo od pivota. Mezi těmito prvky však nebudeme hledat k-tý nejmenší prvek, ale $(k-p)$-tý nejmenší prvek, kde p je pozice pivota v posloupnosti.

Časovou složitost rozebereme podobně jako u QuickSortu. Nešikovná volba pivota dává opět v nejhorším případě kvadratickou složitost. Pokud bychom naopak volili za pivota medián, budeme nejprve přerovnávat N prvků, pak jich zbude nejvýše $N/2$, pak nejvýše $N/4$ atd., což dohromady dává složitost $O(N + N/2 + N/4 + \ldots + 1) = O(N)$. Pro lžimedián dostaneme rovněž lineární složitost a opět stejně jako u QS můžeme nahlédnout, že náhodnou volbou pivota dostaneme v průměru stejný čas jako se lžimediánem.

Program bude velmi jednoduchý, využijeme-li přerovnávací proceduru od QS:

```
function kty(var a:Pole; l,r,k:Integer):Integer;
var x,z:Integer;
begin
 x:=prer(a,l,r); {přerovnej, x je pozice pivota}
 z:=x-l+1;     {pozice pivota vzhledem k [l..r]}
 if k=z then
   kty:=a[x]             {k-tý nejmenší je pivot}
 else if k<z then
   kty:=kty(a,l,x-1,k) {k-tý nejmenší je nalevo}
 else
   kty:=kty(a,x+1,r,k-z);              {napravo}
end;
```

k-tý nejmenší podruhé, tentokrát lineárně a bez náhody

Existuje však algoritmus, který řeší naši úlohu lineárně, a to i v nejhorším případě. Je založený na ďábelském triku: zvolit vhodného pivota (jak ukážeme, bude to jeden ze lžimediánů) rekurzivním voláním téhož algoritmu. Zařídíme to takto:

- Pokud jsme dostali méně než 6 prvků, použijeme nějaký triviální algoritmus, například si posloupnost setřídíme InsertSortem (opět viz třídící kuchařka) a vrátíme k-tý prvek setříděné posloupnosti.
- Rozdělíme prvky posloupnosti na pětice; pokud není počet prvků dělitelný pěti, poslední pětici necháme nekompletní.
- Spočítáme medián každé pětice. To můžeme provést například rekurzivním zavoláním celého našeho algoritmu, čili v důsledku InsertSortem. (Také bychom si mohli pro 5 prvků zkonstruovat rozhodovací strom s nejmenším možným počtem porovnání, což je rychlejší, ale jednak pouze konstanta-krát, jednak je to daleko pracnější.)
- Máme tedy $N/5$ mediánů. V nich rekurzivně najdeme medián m (označíme mediány pětic za novou posloupnost a na ni začneme opět od prvního bodu).
- Přerovnáme vstupní posloupnost po quicksortovsku a jako pivota použijeme prvek m. Po přerovnání je pivot, podobně jako v předchozím algoritmu, na $(z+1)$-ní pozici v posloupnosti, kde z je počet prvků s menší hodnotou, než má pivot.
- Opět, podobně jako u předchozího algoritmu, pokud je $k = z+1$, pak je právě pivot m k-tým nejmenším prvkem posloupnosti. V případě, že tomu tak není a $k < z+1$, budeme hledat k-tý nejmenší prvek mezi prvními z členy posloupnosti, v opačném případě, kdy $k > z+1$, budeme hledat $(k-z+1)$-ní nejmenší prvek mezi posledními $n-z-1$ prvky.

Řečeno s panem Pascalem:

```
{potřebujeme přerovnávací funkci, která
 dostane pozici pivota jako parametr}
function prerp(var a:Pole;
               l,r,m:Integer):Integer;
var q:Integer;
begin
 {pivota prohodíme s posledním prvkem}
 q:=a[m]; a[m]:=a[r]; a[r]:=q;
 {a zavoláme původní přerovnávací fci}
 prerp := prer(a,l,r);
end;

{hledání k-tého nejmenšího prvku z a[l..r],}
{vracíme pozici prvku, nikoliv jeho hodnotu}
function kth(var a:Pole; l,r,k:Integer):Integer;
var medp:Pole;          {pole pro mediány pětic}
    i,j,q,x,pocet,m,z:Integer;
begin
 pocet:=r-l+1;        {s kolika prvky pracujeme}

 if pocet<=1 then           {pouze jeden prvek?}
  kth:=l          {výsledek ani nemůže být jiný}
 else if pocet<6 then begin   {méně než 6 prvků}
  for j:=l+1 to r do begin       {=> InsertSort}
   q:=a[j];
   i:=j-1;
   while (i>=l) and (a[i]>q) do begin
    a[i+1]:=a[i];
    Dec(i);
    end;
   a[i+1]:=q;
   end;
  kth:=l+k;
  end
```

```
 else begin         {mnoho prvků, jde to tuhého}
  {rozdělíme prvky do pětic}
  q:=1;                {zatím máme jednu pětici}
  i:=l;                {levý okraj první pětice}
  j:=i+4;             {pravý okraj první pětice}
  while j<=r do begin   {procházíme celé pětice}
   medp[q]:=kth(a,i,j,2);        {medián pětice}
   Inc(q);                     {zvyš počet pětic}
   Inc(i,5);           {nastav levý okraj pětice}
   Inc(j,5);          {nastav pravý okraj pětice}
   end;
  if i<=r then begin      {zbyla neúplná pětice}
   medp[q]:=kth(a,i,r,(r-i+2) div 2);
   Inc(q);
   end;

  {najdeme medián mediánů pětic, je na pozici m}
  m:=kth(medp,1,q-1,q div 2);

  {přerovnej a zjisti, kde skončil pivot}
  x:=prer(a,l,r,m);
  z:=x-l+1;           {pozice vzhledem k [l..r]}
  if k=z then
   kth:=m               {k-tý nejmenší je pivot}
  else if k<z then
   kth:=kth(a,l,x-1,k)    {k-tý nejmenší nalevo}
  else
   kth:=kth(a,x+1,r,k-z);              {napravo}
 end;
end;
```

Zbývá dokázat, že tato dvojitá rekurze opravdu má lineární složitost. Zkusme se proto podívat, kolik prvků posloupnosti po přerovnání je větších než prvek m. Všech pětic je $N/5$ a alespoň polovina z nich (tedy $N/10$) má medián menší než m. V každé takové pětici pak navíc najdeme dva prvky menší než medián pětice, takže celkem existuje alespoň $3/10 \cdot N$ prvků menších než m. Větších tedy může být maximálně $7/10 \cdot N$. Symetricky ukážeme, že i menších prvků může být nejvýše $7/10 \cdot N$.

Rozdělení na pětice, hledání mediánů pětic a přerovnávání trvá lineárně, tedy nejvýše cN kroků pro nějakou konstantu $c > 0$. Pak už algoritmus pouze dvakrát rekurzivně volá sám sebe: nejprve pro $N/5$ mediánů pětic, pak pro $\leq 7/10 \cdot N$ prvků před/za mediánem. Pro celkovou časovou složitost $t(N)$ našeho algoritmu tedy platí:

$$t(N) \leq cN + t(N/5) + t(7/10 \cdot N).$$

Nyní zbývá tuto rekurzivní nerovnici vyřešit, což provedeme drobným úskokem: uhodneme, že výsledkem bude lineární funkce, tedy že $t(N) = dN$ pro nějaké $d > 0$. Dostaneme:

$$dN \leq (c + 1/5 \cdot d + 7/10 \cdot d) \cdot N.$$

To platí například pro $d = 10c$, takže opravdu $t(N) = O(N)$.

Násobení dlouhých čísel

Dalším pěkným příkladem na rozdělování a panování je násobení dlouhých čísel – tak dlouhých, že se už nevejdou do integeru, takže s nimi musíme počítat po číslicích (ať už v jakékoliv soustavě – my volíme desítkovou, často se hodí třeba 256-ková). Klasickým „školním“ algoritmem pro násobení na papíře to zvládneme na kvadratický počet operací, zde si předvedeme efektivnější způsob.

Libovolné $2N$-ciferné číslo mužeme zapsat jako $10^N A + B$, kde A a B jsou N-ciferná. Součin dvou takových čísel pak bude $(10^N A + B) \cdot (10^N C + D) = (10^{2N} AC + 10^N (AD + BC) + BD)$. Sčítat dokážeme v lineárním čase, násobit mocninou deseti také (dopíšeme příslušný počet nul na konec čísla), N-ciferná čísla budeme násobit rekurzivním zavoláním téhož algoritmu. Pro časovou složitost tedy bude platit $t(N) = cN + 4t(N/2)$. Nyní tuto rovnici můžeme snadno vyřešit, ale ani to dělat nebudeme, neboť nám vyjde, že $t(N) \approx N^2$, čili jsme si oproti původnímu algoritmu vůbec nepomohli.

Přijde trik. Místo čtyř násobení čísel poloviční délky nám budou stačit jen tři: spočteme AC, BD a $(A+B)\cdot(C+D) = AC+AD+BC+BD$, přičemž pokud od posledního součinu odečteme AC a BD, dostaneme přesně $AD + BC$, které jsme předtím počítali dvěma násobeními. Časová složitost nyní bude $t(N) = c'N + 3t(N/2)$. (Konstanta c' je o něco větší než c, protože přibylo sčítání a odčítání, ale stále je to konstanta. My si ovšem zvolíme jednotku času tak, aby bylo $c' = 1$, a ušetříme si tak spoustu psaní.)

Jak naši rovnici vyřešíme? Zkusíme ji dosadit do sebe samé a pozorovat, co se bude dít:

$$\begin{aligned} t(N) &= N + 3(N/2 + 3t(N/4)) = \\ &= N + 3/2 \cdot N + 9t(N/4) = \\ &= N + 3/2 \cdot N + 9/4 \cdot N + 27t(N/8) = \ldots = \\ &= N + 3/2 \cdot N + \ldots + 3^{k-1}/2^{k-1} \cdot N + 3^k t(N/2^k). \end{aligned}$$

Pokud zvolíme $k = \log_2 N$, vyjde $N/2^k = 1$, čili $t(N/2^k) = t(1) =$ nějaká konstanta d. To znamená, že:

$$t(N) = N \cdot (1 + 3/2 + 9/4 + \ldots + (3/2)^{k-1}) + 3^k d.$$

Výraz v závorce je součet prvních k členů geometrické řady s kvocientem $3/2$, čili $((3/2)^k - 1)/(3/2 - 1) = O((3/2)^k)$. Tato funkce však roste pomaleji než zbylý člen $3^k d$, takže ji klidně můžeme zanedbat a zabývat se pouze oním posledním členem: $3^k = 2^{k \log_2 3} = 2^{\log_2 n \cdot \log_2 3} = (2^{\log_2 n})^{\log_2 3} = n^{\log_2 3} \approx n^{1.58}$. Konstanta d se nám „schová do O-čka“, takže algoritmus má časovou složitost přibližně $O(n^{1.58})$. Umí se to i lépe – $O(n \log n)$, ale to je mnohem ďábelštější a pro malá n se to sotva vyplatí.

Program si pro dnešek odpustíme, šetřímeť naše lesy.

Poznámky na ubrousku aneb Rozmyslete si

- Při hledání k-tého nejmenšího prvku jsme předpokládali, že všechny prvky jsou různé. Prohlédněte si algoritmy pozorně a rozmyslete si, že budou fungovat i bez toho. Opravdu?
- Proč jsme zvolili zrovna pětice? Jak by to dopadlo pro trojice? A jak pro sedmice? Fungoval by takový algoritmus? Byl by také lineární?
- Ve výpočtu $t(N)$ jsme si nedali pozor na neúplné pětice a také jsme předpokládali, že pětic je sudý počet. Ono se totiž nic zlého nemůže stát. Jak se to snadno nahlédne? Proč nestačí na začátku doplnit vstup „nekonečny“ na délku, která je mocninou pěti?
- Kdybychom neuhodli, že $t(N)$ je lineární, jak by se na to dalo přijít?
- Ještě jednou QS: Představte si, že budujete binární vyhledávací strom vkládáním prvků v náhodném pořadí. Obecně nemusí být vyvážený, ale v průměru v něm půjde vyhledávat v čase $O(\log N)$. Žádný div: stromy, které nám vzniknou, odpovídají přesně možným průběhům QuickSortu.

Dnešní menu Vám servírovali
David Matoušek & Martin Mareš

17-2-1 Prasátko Květomil

Jak si mnozí řešitelé správně povšimli, tato úložka byla zaměřena převážně na procvičení hešování – to se nám bude hodit dokonce dvakrát.

Nejprve si povšimněme, že se po nás chce pouze spočítat počet výrazů v programu, jejichž hodnota je různá – výrazy se stejnou hodnotou bychom nevyhodnocovali dvakrát, ale poprvé uložili do pomocné proměnné a podruhé použili tuto uloženou hodnotu. Upřesněme si ještě, co to znamená „mít stejnou hodnotu“. Představme si, že bychom za proměnné ve výrazech postupně dosazovali jejich definice tak dlouho, dokud by alespoň jedna proměnná neměla svou počáteční hodnotu. Pak dva výrazy E_1 a E_2 jsou si rovny, pokud

- $E_1 = E_2 = v$, kde v je nějaká proměnná, nebo
- $E_1 = E_1' \; op \; E_1''$, $E_2 = E_2' \; op \; E_2''$, kde op je buď $+$ nebo $*$ a buď
  - E_1' je rovno E_2' a E_1'' je rovno E_2'', nebo
  - E_1' je rovno E_2'' a E_1'' je rovno E_2'.

Samozřejmě ověřovat rovnost přímo podle této definice je nevhodné (už proto, že takto rozexpandované výrazy mohou mít i exponenciální velikost). Místo toho každému výrazu přiřadíme číslo, které bude reprezentovat jeho hodnotu – tj. dva výrazy dostanou stejné číslo právě tehdy, pokud jsou si rovny, jinak dostanou různá čísla.

První hešovací tabulka A bude jménu proměnné přiřazovat číslo hodnoty, která je aktuálně v této proměnné uložena. Ve druhé hešovací tabulce B si pak budeme pamatovat čísla hodnot výrazů, které se v programu vyskytují – klíčem této tabulky budou trojice (operátor, číslo hodnoty levého operandu, číslo hodnoty pravého operandu), a jim bude přiřazeno číslo hodnoty tohoto výrazu. Na konci stačí vypsat počet různých čísel hodnot v tabulce B, protože to bude právě počet různých hodnot výrazů v programu.

Čísla hodnot výrazů určujeme takto:

- Když zpracováváme nějakou proměnnou poprvé, přiřadíme jí nové číslo hodnoty.
- Když zpracováváme přiřazení $var_1 = var_2$, pak proměnné var_1 přiřadíme stejné číslo hodnoty, jaké má proměnná var_2.
- Když zpracováváme přiřazení $var_1 = var_2 \; op \; var_3$, pak si nejprve zjistíme čísla hodnot v proměnných var_2 a var_3 – nechť to jsou n_2 a n_3. Pak se podíváme do hešovací tabulky B, zda v ní je uložen výraz (op, n_2, n_3). Je-li tomu tak, pak jeho číslo hodnoty přiřadíme proměnné var_1. Jinak tento výraz přidáme do tabulky B s novým číslem hodnoty, a toto číslo přiřadíme proměnné var_1.

Zbývá si rozmyslet, jak ošetřit komutativitu operací. To je ale snadné – před prací s tabulkou B stačí čísla hodnot v trojici seřadit tak, aby druhé z nich bylo menší nebo rovno třetímu.

Časová složitost na operaci s tabulkou A je v průměrném případě $O(k)$, kde k je délka názvu proměnné. Protože pro každý výskyt proměnné v programu provedeme právě jednu operaci s touto tabulkou, dohromady bude časová složitost pro práci s ní $O(n)$, kde n je délka vstupu. Časová složitost pro práci s tabulkou B je $O(1)$ na operaci, a počet operací s ní je roven počtu přiřazení ve vstupu, tj. celková časová složitost je $O(n)$ – toto je složitost v průměrném případě, v nejhorším případě, kdy by docházelo ke všem možným kolizím, by časová složitost byla $O(n^2)$. Paměťová složitost je zřejmě $O(n)$.

Poznámka na závěr – zde popsaná metoda identifikace redundantních výpočtů se s mírnými vylepšeními skutečně používá v kompilátorech. Anglický název je Value Numbering.

Zdeněk Dvořák

17-2-2 Bobr Béďa

Všichni správně uhodli, že jde o hledaní minimální kostry grafu a že předem dané hrany tvořící cykly nijak nevadí. Mnoho z vás ale pak zvolilo buď zbytečně pomalý algoritmus (no comment :–)) nebo použili více či méně rychlou implementaci DFU. Ta sice běží v čase $O(N \cdot \alpha(N))$, ale pro tuto úlohu je zbytečně složitá.

Naše řešení bude založeno na Jarníkově algoritmu pro hledání kostry. Budeme postupně budovat kostru tak, že začneme s jedním libovolným vrcholem a vybereme si jeho souseda takového, že ještě v kostře není, a přitom je k současné kostře nejblíže. Takto pokračujeme, dokud nepřidáme vrcholy všechny.

Popsaný algoritmus naimplementujeme pomocí haldy tak, že na začátku začneme s libovolným vrcholem a do haldy přidáme všechny hrany, které z tohoto vrcholu vedou. V každém kroku pak z haldy vezmeme hranu s nejmenším ohodnocením a podíváme se, jestli nějaký její konec ještě není v kostře. Pokud ne, přidáme ho do ní (je momentálně nejblíž vytvářené kostře) a do haldy vložíme všechny hrany z tohoto vrcholu vedoucí. Pokud už oba konce hrany v kostře jsou, neděláme nic. Tento celý postup se opakuje, dokud je v haldě nějaká hrana.

Jaká bude časová složitost? Každou hranu přidáme do haldy maximálně dvakrát a na každý vrchol sáhneme nanejvýš čtyřikrát (tolik z něj vede hran). Pokud by naše operace s haldou byly v konstantním čase, bude celková časová složitost $O(N \cdot M)$.

V haldě budeme mít naštěstí jenom hrany s ohodnocením 0 (dopředu vyryté kanálky), 1 (svislé kanálky) a 2 (vodorovné kanálky). Můžeme si tedy pamatovat hrany ve třech polích podle jejich ohodnocení a pokud hledáme hranu s nejmenším ohodnocením, zkusíme pole s ohodnocením 0, a pokud je prázdné, tak 1 nebo 2. Každopádně všechny tyto operace (přidat hranu a odebrat hranu s nejmenším ohodnocením) zvládneme v konstantním čase.

Můj program bude mít v poli $h[i]$ vrcholy, ke kterým vede z už propojené části hrana s ohodnocením i. Vrchol v nich může tak být až 4× (přidán z různých stran), ale to mi nijak nevadí. Do polí fv a fs si na začátku načtu již hotové hrany.

Tomáš Gavenčiak

17-2-3 Krkavec Kryšpín

Těžiště trojúhelníku je zároveň středem kružnice opsané, je to bod, který je od všech vrcholů stejně vzdálen. Souřadnice těžište trojúhelníku lze tedy můžeme spočítat podle vzorce $x = (x_1 + x_2 + x_3)/3$ a $y = (y_1 + y_2 + y_3)/3$.

Další vzorec, který budeme potřebovat, je z fyziky: těžiště soustavy hmotných bodů lze spočítat jako vážený průměr

souřadnic těch bodů. Tedy

$$x = (x_1m_1 + x_2m_2 + \ldots + x_nm_n)/(m_1 + m_2 + \ldots + m_n)$$
$$y = (y_1m_1 + y_2m_2 + \ldots + y_nm_n)/(m_1 + m_2 + \ldots + m_n)$$

kde x_i, y_i jsou souřadnice bodů a m_i je jeho hmotnost. Pro naše účely na jednotce hmotnosti nezáleží. Vzorec bude fungovat i pro záporné „hmotnosti“, což se nám bude hodit pro nekonvexní útvary.

Ještě potřebujeme znát plochu jednoho trojúhelníku, tu lze získat jednoduše jako $S = 1/2 \cdot |AB \times AC|$ neboli $S = 1/2 \cdot |(B_x - A_x) \cdot (C_y - A_y) - (C_x - A_x) \cdot (B_y - A_y)|$.

Pro konvexní útvary by stačilo rozdělit mnohoúhelník na trojúhelníky A_1, A_i, A_i+1, a těžiště získat pomocí tří vzorců nahoře.

Pokud při výpočtu plochy použijeme vzorec $S = 1/2 \cdot ((B_x - A_x) \cdot (C_y - A_y) - (C_x - A_x) \cdot (B_y - A_y))$ (tedy bez absolutní hodnoty), budeme dostávat kladné a záporné výsledky podle toho, jestli trojúhelník ABC je orientován po nebo proti směru hodinových ručiček. Toho lze obratně využít: vybereme libovolný bod O (třeba počátek systému souřadnic) a použijeme vzorce nahoře pro trojúhelníky O, A_i, A_i+1. Každý bod, který je uvnitř mnohoúhelníku bude součástí lichého počtu dílčích trojúhelníků, a díky tomu se bude počítat do celkového výsledku. Body, které jsou mimo mnohoúhelnik, budou součástí sudého počtu trojúhelníků, a díky opačným orientacím se navzájem odečtou a celkový výsledek neovlivní.

Časová složitost je lineární vzhledem k počtu vrcholů, a lépe to nejde, protože výsledek záleží na všech bodech. Paměťová slozitost je konstantní, body jsou zpracovávány hned, jak přichází ze vstupu.

S díky Janu Pelcovi.

Pavel Machek

17-2-4 Mravenec Ferda

Ferda se dlouho probíral vašimi programy, ale naštěstí pro něj a naneštěstí pro Ptáčka Sáčka nalezl mezi řešeními kýžený algoritmus.

Hlavní myšlenkou algoritmu je zpracovávat úlohu postupně. Bude nás zajímat součet horních čísel na kostkách. Pokud otočíme kostku, změní se horní číslo na kostce, a tedy i součet horních čísel. Naším cílem je najít takové otočení kostek, aby součet horních čísel byl co nejbližší součtu dolních čísel. Víme, že součet čísel je menší než $K \cdot N$. Zajímáme se tedy, pro které součty čísel s existuje otočení kostek, jehož součet horních čísel na kostkách je právě s. Dalším parametrem součtu čísel s je minimální počet otočených kostek $O[s]$, který je potřeba, aby jsme dosáhli součtu s. Pole O budeme budovat postupně. Označme O_i pole O vytvořené z i prvních kostek. Zřejmě $O_0[0] = 0$ a pro ostatní součty s má hodnotu *Null*, která znamená, že tohoto součtu nelze dosáhnout. Jakmile máme vytvořené pole O_i, pak pole O_{i+1} naplníme následovně: Označme čísla na $i+1$-ní kostce h a d. Projdeme všechny možné součty s od 0 do $K \cdot N$. Pokud $O_i[s]$ není *Null*, zkusíme k součtu přidat kostku. Pokud je $O_i[s] < O_{i+1}[s+h]$, nastavíme $O_{i+1}[s+h] := O_i[s]$ a podobně pokud $O_i[s]+1 < O_{i+1}[s+d]$, nastavíme $O_{i+1}[s+d] := O_i[s] + 1$. Samozřejmě v případě hodnoty *Null* uložíme novou hodnotu. Tímto způsobem sice ztratíme některé možnosti otočení kostek, ale určitě si uchováme ty součty, které potřebují nejmenší otočení kostek. Jakmile naplníme pole O_n, jsme hotovi. Stačí už jen vybrat nejbližší součet a najít kostky, které musíme otočit. Otočení jednotlivých kostek si uložíme už při vytváření pole O – do pole $T_i[s]$ si poznamenáme, zda jsme při vytváření součtu s z prvních i kostek otočili i-tou kostku. Pak už stačí jen vystopovat všechny otočené kostky z finálního součtu prostým odečítáním horních či dolních čísel kostek.

Algoritmus používá k uložení součtů N polí o velikosti $K \cdot N$, takže jeho paměťová složitost je $O(N^2 \cdot K)$. Časově nejnáročnější operací je právě naplnění těchto polí, tedy časová složitost je také $O(N^2 \cdot K)$. V algoritmu jsme použili myšlenku spočítat si řešení pro část vstupu, uložit si ho a pak ho znovu použít pro další výpočet. Tomuto způsobu řešení úloh se říká dynamické programování.

Petr Škoda

17-2-5 Jazykozpytcova pomsta

Správných řešení první úlohy, ke kterým jsem neměl žádnou připomínku, tentokrát došlo poskrovnu. Nejběžnější chyba byla následující: V podstatě všichni řešitelé přišli na správnou myšlenku, že automat nutně nějak musí umět rozlišovat hloubku vnoření závorek. Bohužel už málokdo to uměl i správně dokázat. To přeci vůbec není na první pohled zřejmé! Stejně tak bychom mohli tvrdit, že když rozpoznáváme jazyk $\{a^i;\ i \in N\}$, tak je nutné si počítat ono i, ve skutečnosti to samozřejmě potřeba není. Proč by třebas nemohl existovat automat, který používá nějakou úplně jinou metodu? Řešitelům jsem uděloval body podle toho, nakolik myšlenku důkazu dotáhli do konce.

Ale teď už si předveďme jeden z možných správných důkazů. Jeho myšlenka je jednoduchá: automat se nemůže obejít bez rozlišování úrovně vnoření závorek. Jak to ovšem ukázat formálně?

Budeme postupovat sporem, nechť existuje konečný automat $M = (Q, A, q_0, \delta, F)$, který má k stavů a rozpoznává jazyk U správně uzávorkovaných výrazů. Vezmeme vhodné správně uzávorkované slovo a ukážeme, že z něj lze vynechat úsek tak, že slovo přestane být dobře uzávorkované, ale automat to vůbec nepozná a prohlásí ho za správné. Tím ukážeme, že žádný konečný automat M, který by měl umět rozpoznávat U, ve skutečnosti nikdy nemůže dobře pracovat.

Když má tedy automat k stavů, uvažme slovo $(^{k+1})^{k+1}$. Při načítání levých závorek automat nějak mění stavy, ale protože levých závorek je $k+1$, alespoň jedním stavem se musí projít dvakrát. Existují tedy dva indexy i a j, $i < j$, že po přečtení j-té levé závorky se automat ocitl ve stejném stavu q jako po přečtení i-té levé závorky. Jinými slovy, automat ve své „paměti“ považuje pozice i a j za nerozlišitelné. V obou případech se totiž stroj nachází ve stavu q, a následný výpočet tudíž musí mít úplně stejný průběh. A co se tedy stane, když automatu podstrčíme slovo, kde vynecháme levé závorky na pozicích $i+1$ až j? Automat to vůbec nepozná a prohlásí, že slovo je správné!

Dokonce bychom mohli tento úsek ne vynechat, nýbrž zdvojit, ztrojit, zkrátka libovolně mnohokrát znásobit. Když se nad tím zamyslíme hlouběji, podobnou vlastnost musí mít všechny nekonečné regulární jazyky. Stačí vzít dostatečné dlouhé slovo, a pak už se v něm nutně musí vyskytovat úsek, který lze beztrestně odmazat či libovolněkrát „nafouknout“. Tato skutečnost se dá v literatuře najít pod názvem *Pumping lemma.*

→ ∗ ←

Druhá úloha byla myšlenkově jednodušší, zato bylo potřeba být pečlivější a dát si pozor na některé zrady, které mohly nastat. V podstatě všichni, kdo úlohu odeslali, správně přišli na to, že stačí vzít automaty $M_1 = (Q_1, A_1, q_1^0, \delta_1, F_1)$ a $M_2 = (Q_2, A_2, q_2^0, \delta_2, F_2)$, které jsou dle definice regulárního jazyka schopné rozpoznávat jazyky L_1 a L_2, a vhodně je sériově zapojit do nového stroje M, který bude rozpoznávat jazyk $L_1.L_2$. Například můžeme na každý přijímací stav f_i stroje M_1 „přivěsit“ kopii stroje M_2 tak, že ztotožníme stav f_i s počátečním stavem stroje M_2. Počátečním stavem zvolíme počáteční stav q_1^0 stroje M_1 a jako koncové stavy zvolíme koncové stavy F_2 strojů M_2.

Z přijímacích stavů M_1 se však ještě výpočet může vrátit zpět do vnitřních stavů M_1, a tyto zpětné šipky nemůžeme vynechat. Po napojení stroje M_2 tudíž v propojovacích stavech vznikne více šipek pro jediné písmenko. To je ale přesně nedeterministický konečný automat. Jenže definice regulárního jazyka je taková, že pro něj musí existovat *deterministický* konečný automat. Tehdy však stačí použít větu dokázanou v zadání, podle které umíme k nedeterministickému automatu M sestrojit ekvivalentní deterministický automat.

Ještě je potřeba vyřešit několik důležitých technických detailů. Pokud bychom spojení obou automatů realizovali ztotožněním přijímacího a počátečního stavu, mohlo by se ještě stát, že se výpočet, který již přešel do M_2 přes propojovací stav, vrátí zpět do stroje M_1, což my určitě nechceme. Napojení se tudíž musí řešit rafinovaněji: Vezmeme stroj M_1 a pouze jednu kopii stroje M_2. Z každého stavu stroje M_1, ze kterého vede šipka pro písmeno a do některého přijímacího stavu z F_1, natáhneme ještě jednu šipku pro písmeno a navíc do počátečního stavu q_2^0 stroje M_2. Také je třeba ošetřit, když při práci stroje M_1 přijde znak z abecedy stroje M_2 a naopak. Zavedeme proto „odpadní“ stav q_{err}, ze kterého už nebude úniku a směřovat do něj budou šipky pro všechny špatné znaky.

Následujícím poněkud odpudivým formálním zápisem ještě výsledný nedeterministický stroj

$$M = (Q_1 \cup Q_2 \cup \{q_{err}\}, A_1 \cup A_2, P, \delta, F_2)$$

přesně definujeme. Pokud je stav $q_1^0 \in F_1$, bude množina počátečních stavů $P = \{q_1^0, q_2^0\}$, v opačném případě $P = \{q_1^0\}$. Přechodová funkce δ bude

$$\delta(q,a) = \begin{cases} \{\delta_1(q,a)\} & q \in Q_1,\ a \in A_1,\ \delta_1(q,a) \notin F_1 \\ \{\delta_1(q,a), q_2^0\} & q \in Q_1,\ a \in A_1,\ \delta_1(q,a) \in F_1 \\ \{\delta_2(q,a)\} & q \in Q_2,\ a \in A_2. \\ \{q_{err}\} & q = q_{err},\ a \in A_1 \cup A_2 \\ \{q_{err}\} & q \in Q_1,\ a \in A_2 \setminus A_1 \text{ nebo} \\ & q \in Q_2,\ a \in A_1 \setminus A_2 \end{cases}$$

Na tento výsledný NKA M nyní aplikujeme větu o převodu na DKA a důkaz je hotov.

Ještě bychom měli věnovat pár slov několika málo řešitelům, který svůj důkaz založili na zřetězení gramatik speciálního tvaru $X \to aY$, $X \to a$ pro a terminální a X, Y neterminální, jež ke každému regulárnímu jazyku existují. Myšlenka je to samozřejmě dobrá, ale trpí stejnými neduhy při spojování jako automaty. Je opět třeba zajistit, aby se expanze gramatiky z pravidel pro L_2 nevrátila zpět do pravidel pro L_1. Navíc my jsme si ekvivalenci automatu a gramatik výše uvedeného tvaru celou nedokazovali. Druhý směr, tedy konstrukci deterministického automatu z gramatiky, jsme schválně ve vzorovém řešení první série odbyli, neboť ony „detaily, které si každý rozmyslí sám“ znamenají právě konstrukci nedeterministického konečného automatu a jeho následný převod na deterministický například pomocí věty o ekvivalenci DKA a NKA.

Tomáš Valla

Úloha 17-2-1 – Prasátko Květomil – program

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

#define MAX_VARS 107
#define MAX_EQS 107
#define MAX_LINE_LENGTH 100

struct var_hash_elt
{
    char *var_name;
    unsigned var_value;
};

struct expr_hash_elt
{
    char operator;
    unsigned left_val, right_val;
    unsigned expr_value;
};

struct var_hash_elt var_hash[MAX_VARS];
struct expr_hash_elt expr_hash[MAX_EQS];

unsigned n_vars, n_values;

static struct var_hash_elt *
get_var (char *var_name)
{
    unsigned hash = 0;
    char *t;

    for (t = var_name; *t; t++)
```

```
    hash = (76 * hash + *t) % MAX_VARS;
  while (var_hash[hash].var_name
         && strcmp (var_hash[hash].var_name, var_name))
    {
      hash++;
      if (hash == MAX_VARS)
        hash = 0;
    }
  if (!var_hash[hash].var_name)
    {
      var_hash[hash].var_name = strdup (var_name);
      var_hash[hash].var_value = n_values++;
      n_vars++;
    }
  return var_hash + hash;
}
static struct expr_hash_elt *
get_expr (unsigned left_val, char op, unsigned right_val)
{
  unsigned hash, t;
  if (left_val > right_val)
    {
      t = left_val;
      left_val = right_val;
      right_val = t;
    }
  hash = (76 * left_val + 777 * op + right_val) % MAX_EQS;
  while (expr_hash[hash].operator
         && (expr_hash[hash].operator != op
             || expr_hash[hash].left_val != left_val
             || expr_hash[hash].right_val != right_val))
    {
      hash++;
      if (hash == MAX_EQS)
        hash = 0;
    }
  if (!expr_hash[hash].operator)
    {
      expr_hash[hash].operator = op;
      expr_hash[hash].left_val = left_val;
      expr_hash[hash].right_val = right_val;
      expr_hash[hash].expr_value = n_values++;
    }
  return expr_hash + hash;
}
static int
id_char (char ch)
{
  return ( ('a'<= ch && ch <= 'z')
           || ('A'<= ch && ch <= 'Z')
           || ch == '_');
}
static void
skip_blanks (char **buffer)
{
  while (**buffer == '␣')
    (*buffer)++;
}
static struct var_hash_elt *
parse_var (char **buffer)
```

```
{
  char *var_end, ech;
  struct var_hash_elt *ret;

  skip_blanks (buffer);
  for (var_end = *buffer; id_char (*var_end); var_end++)
    continue;
  ech = *var_end;
  *var_end = 0;

  ret = get_var (*buffer);

  *var_end = ech;
  *buffer = var_end;
  skip_blanks (buffer);

  return ret;
}

static char
parse_eq (char *buffer, struct var_hash_elt **tgt, unsigned *left_val, unsigned *right_val)
{
  char ret;

  skip_blanks (&buffer);
  if (!*buffer)
    return '␣';

  *tgt = parse_var (&buffer);

  if (*buffer++ != '=')
    abort ();

  *left_val = parse_var (&buffer)->var_value;
  if (*buffer == ';')
    ret = '=';
  else
    {
      ret = *buffer++;
      if (ret != '+' && ret != '*')
        abort ();

      *right_val = parse_var (&buffer)->var_value;
    }

  if (*buffer++ != ';')
    abort ();
  skip_blanks (&buffer);
  if (*buffer)
    abort ();

  return ret;
}

int
main (void)
{
  char buffer[MAX_LINE_LENGTH];

  while (gets (buffer))
    {
      unsigned left_val, right_val;
      struct var_hash_elt *tgt;
      char eq_type = parse_eq (buffer, &tgt, &left_val, &right_val);

      switch (eq_type)
        {
        case '*':
        case '+':
          tgt->var_value = get_expr (left_val, eq_type, right_val)->expr_value;
          break;

        case '=':
          tgt->var_value = left_val;
```

```
            break;
          default:
            break;
          }
      }
    printf ("%d\n", n_values − n_vars);
    return 0;
}
```

Úloha 17-2-2 – Bobr Béďa – program

```
var hp:array[0..2] of integer;              {počty hran s ohodnocením 0..2}
    hx:array[0..2,1..2*M*N] of integer;     {počáteční souřadnice}
    hy:array[0..2,1..2*M*N] of integer;
   hxo:array[0..2,1..2*M*N] of integer;     {cílové souřadnice}
   hyo:array[0..2,1..2*M*N] of integer;
     f:array[1..M,1..N] of boolean;         {značky navštívení}
    fv:array[1..M,1..N] of boolean;         {je kanálek z [i,j] doprava už vyroben}
    fs:array[1..M,1..N] of boolean;         {je kanálek z [i,j] dolů už vyroben}

procedure pridej(xo,yo,x,y:integer);        {přidá do haldy hranu odkud-kam}
var v:integer;
begin
  if xo>x and fv[x,y] then v:=0             {zjistím cenu}
    else if xo<x and fv[xo,y] then v:=0
      else if yo>y and fs[x,y] then v:=0
        else if yo<y and fs[x,yo] then v:=0
          else if yo!=y then v:=1
            else v:=2;
  inc(hp[v]);
  hx[v,hp[v]]:=x;
  hy[v,hp[v]]:=y;
  hxo[v,hp[v]]:=xo;
  hyo[v,hp[v]]:=yo;
end;

begin
  {nactu uz hotové do fv,fs}

  for i:=0 to N do
     for j:=1 to M do
      f[i,j]:=false;
  hp[0]:=1;
  hp[1]:=0;   hp[2]:=0;
  hx[0,1]=1; hy[0,1]:=1;
  while (hp[0]>0) or (hp[1]>0) or (hp[2]>0) do
  begin
    if hp[0]>0 then                         {vyberu hranu z haldy}
    begin
      x:=hx[0,hp[0]]; y:=hy[0,hp[0]];
      xo:=hx[0,hp[0]]; yo:=hy[0,hp[0]]; dec(hp[0]);
    end else
      if hp[1]>0 then
      begin
        x:=hx[1,hp[1]]; y:=hy[1,hp[1]];
        xo:=hx[1,hp[1]]; yo:=hy[1,hp[1]]; dec(hp[1]);
      end else
        begin
          x:=hx[2,hp[2]]; y:=hy[2,hp[2]];
          xo:=hx[2,hp[2]]; yo:=hy[2,hp[2]]; dec(hp[2]);
        end;
    if(not f[x,y])
    begin                                   {přidám [x,y] do kostry?}
      f[x,y]:=true;
      writeln("(",x,",",y,")-(",xo,",",yo,")");
```

```
      if y>1 then pridej(x,y,x,y-1);
      if x>1 then pridej(x,y,x-1,y);
      if y<N then pridej(x,y,x,y+1);
      if x<M then pridej(x,y,x+1,y);
    end;
  end;
end.
```

Úloha 17-2-3 – Krkavec Kryšpín – program

```
#include <stdio.h>

#define maxP 100
#define maxN 100
#define MAX (a, b)  ( ( (a)> (b)) ? (a): (b))

int main () {
        int N;                                          /* kolik zloděj unese */
        int P;                                          /* počet předmětů */
        int m[maxP];                                    /* hmotnosti předmětů */
        float c[maxP];                                  /* ceny předmětů */

        /* maximální hodnota lupu z prvních p předmětů v batohu s kapacitou n */
        float batoh[maxP][maxN];

        scanf ("%d %d",  &N,  &P);
        for  (int i=1; i<=P; i++)
                scanf ("%d %f",  &m[i],  &c[i]);

        for  (int j=0; j<=N; j++)
                batoh[0][j] = 0;

        /* postupně přidáváme předměty */
        for  (int i=1; i<=P; i++) {
                for  (int j=0; j<m[i]; j++)             /* nízké hmotnosti jenom zkopírujeme */
                        batoh[i][j] = batoh[i-1][j];
                for  (int j=m[i]; j<=N; j++)            /* pokud lze, zlepšíme i-tým předmětem */
                        batoh[i][j] = MAX (batoh[i-1][j],  batoh[i-1][j-m[i]]+c[i]);
        }

        printf ("cena: %f\n",  batoh[P][N]);
        printf ("předměty:");
        for  (int i=P,  j=N; i>0; i--)
                if  (batoh[i][j] > batoh[i-1][j]) {     /* předmět i je v batohu, vylepšil celkovou cenu */
                        printf ("␣%d",  i);
                        j -= m[i];
                }

        return 0;
}
```

Úloha 17-2-4 – Mravenec Ferda – program

```
program Ferda;
const
  MaxN = 1000;
  MaxK = 10;
  Null = -1;
var
  O: array[0..MaxN, 0..MaxN * MaxK] of Integer;
  T: array[0..MaxN, 0..MaxN * MaxK] of Boolean;
  H, D: array[0..MaxN] of Integer;
  K, N, R: Integer;

  I, S: Integer;
begin
  Readln(N, K);
  for I:= 1 to N do
  Readln(H[I], D[I]);

  for I:= 0 to N do
```

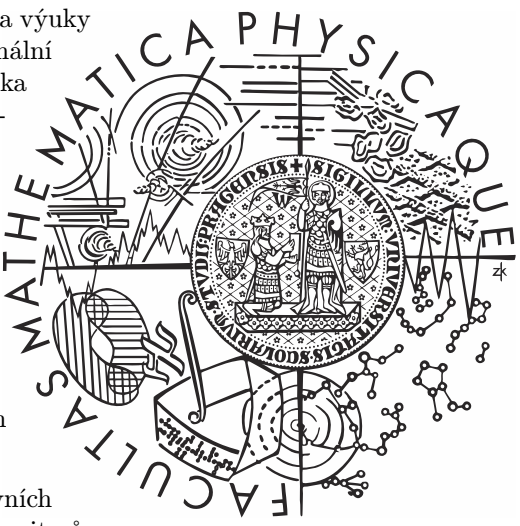
```
  for S:= 0 to N * K do
  O[I, S]:= Null;

  O[0, 0]:= 0;
  for I:= 0 to N - 1 do
  for S:= 0 to K * N do
  if O[I, S] <> Null then
  begin
    if (O[I + 1, S + H[I + 1]] = Null) or (O[I, S] < O[I + 1, S + H[I + 1]]) then
    begin
      O[I + 1, S + H[I + 1]]:= O[I, S];
      T[I + 1, S + H[I + 1]]:= False;
    end;
    if (O[I + 1, S + D[I + 1]] = Null) or (O[I, S] + 1 < O[I + 1, S + D[I + 1]]) then
    begin
      O[I + 1, S + D[I + 1]]:= O[I, S] + 1;
      T[I + 1, S + D[I + 1]]:= True;
    end;
  end;

  R:= 0;
  for I:= 1 to N do
  R:= R + H[I] + D[I];

  for I:= R div 2 downto 0 do
  if (O[N, I] <> Null) or (O[N, R - I] <> Null) then
  begin
    if (O[N, R - I] = Null) or (O[N, I] <= O[N, R - I]) then
      R:= I else R:= R - I;
  end;

  for I:= N downto 1 do
  if T[N, R] then
  begin
    Writeln(I);
    R:= R - D[I];
  end else
    R:= R - H[I];
end;
```



Milí řešitelé, v minulých dílech jsme vám postupně představili Kabinet software a výuky informatiky (KSVI), Středisko informatické sítě a laboratoří (SISAL), Ústav formální a aplikované lingvistiky (ÚFAL) a Centrum komputační lingvistiky (CKL). Dneska se posuneme v budově MFF UK na Malostranském náměstí do druhého a třetího patra do západního křídla budovy, kde společně sídlí Katedra aplikované matematiky (KAM) a Institut teoretické informatiky (ITI).

Katedra aplikované matematiky patří k předním světovým pracovištím v diskrétní matematice, jež v současné podobě vzniklo během posledních 15 let pod vedením profesora Nešetřila. Členové této katedry se intenzivně věnují takovým oborům matematiky jako jsou teorie grafů, algoritmy, výpočetní geometrie, ale třeba i teorie čísel, kombinatorická optimalizace nebo operační výzkum. Připomeňme si, že operační výzkum je oblast na pomezí informatiky a matematiky, která zahrnuje studium úloh jako je např. rozvrhování výrobních procesů v průmyslových podnicích.

Katedra aplikované matematiky zajišťuje výuku řady kurzů povinných v prvních ročnících studia na MFF, kromě přednášek z teorie grafů, kombinatoriky, a algoritmů, též výuku některých čistě matematických předmětů pro studenty informatiky (např. matematické analýzy nebo lineární algebry). Kromě povinných přednášek, tato katedra nabízí velmi široké spektrum odborných přednášek a seminářů, na jejichž výuce se podílejí přední odborníci z Akademie věd České republiky a jiných českých vysokých škol. Tradičním seminářem pro studenty bakalářských a magisterských programů je kombinatorický seminář, kde studenti mají možnost seznámit se s novými výsledky z oblasti diskrétní matematiky. Katedra aplikované matematiky též každoročně pořádá pro své studenty jedno- či dvoutýdenní Jarní školu kombinatoriky. Na této katedře mohou na bakalářské nebo diplomové práci na MFF UK pracovat nejen studenti informatiky ale i studenti matematiky.

Katedra aplikované matematiky má široké mezinárodní styky, zaštiťované centrem DIMATIA (Center for Discrete Mathematics, Theoretical Computer Science and Applications), v jejichž rámci pořádá velké množství mezinárodních konferencí a workshopů, účastní se evropské sítě COMBSTRU, atd. Rozsáhlá mezinárodní spolupráce například umožňuje, aby každý rok skupina pregraduální studentů této katedry strávila několik týdnů v partnerském centru DIMACS na Rutgers university v New Jersey, USA.

Před pěti lety vznikl na MFF UK *Institut teoretické informatiky* jako výzkumné centrum podporované Ministerstvem školství, mládeže a tělovýchovy České republiky. Tento společný projekt Univerzity Karlovy, Akademie věd České republiky a Západočeské univerzity, je předním českým výzkumným pracovištěm v kombinatorice a teoretické informatice, které za pět let své existence získalo vynikající postavení ve světě. Personálně se na řešení projektu na MFF UK podílí pracovníci Katedry aplikované matematiky a Katedry teoretické informatiky a matematické logiky. Během plánovaného pokračování projektu by se na jeho řešení v příštích pěti letech měla nově podílet Masarykova Univerzita v Brně, a aktivity Institutu nově zahrnou úzkou spolupráci s některými českými a zahraničními firmami na poli výzkumu a vývoje.

Výsledková listina sedmnáctého ročníku KSP po druhé sérii

		škola	ročník	1721	1722	1723	1724	1725	suma	celkem
1.	Peter Černo	GĽŠtúra	4	10	9	11	10	11	51	101
2.	Miroslav Cicko	GJGTajov	4	10	9	11	10	10	50	94
3.	Miroslav Klimoš	G Lanškr	0	10	9	9	10	9	47	93
4.	Ondřej Bílka	G Zlín	3	10	10	10	8		38	86
5.	Peter Perešíni	GJGTajov	3	6	9	11	10	5	41	81
6. – 7.	Jan Pelc	G UBrod	3	10	10	11	3	7	41	70
	Josef Pihera	G Strakon	2	8	10	9	9	5	41	70
8.	Martin Koníček	G UBrod	4		9	11	6	7	33	64
9.	Zbyněk Konečný	GKptJaroš	2		10	10	4	2	26	62
10.	Adam Zivner	G UBrod	3	8	9	8	8	3	36	56
11.	Martin Čech	G UBrod	4	4	9	11	6	7	37	52
12.	Pavel Klavík	G Chrudim	2	0	10	6	4	6	26	46
13.	Stanislav Basovník	G Kroměříž	4		6	9		8	23	43
14.	Jan Bulánek	G Klatovy	4	6	9	6			21	40
15. – 16.	Jakub Kaplan	GJKTyla	1		5	8	5	8	26	37
	Lukáš Lánský	GJKTyla	1			9	3	5	17	37
17. – 18.	Jan Hrnčíř	GFXŠaldy	3	2	7	7	4		20	34
	Petr Kratochvíl	G SvětláNS	2		6	1	2	5	14	34
19.	Martin Kupec	GMendel	3		6	5		4	15	28
20.	Tomáš Herceg	G Třebíč	2	0	3	7	1		11	27
21.	Roman Smrž	GOhradní	1		10	4			14	26
22. – 23.	Eva Schlosáriková	G Piešťany	4		7	9	3		19	25
	Josef Špak	GJírovco	2			7		2	9	25
24.	Lukáš Špalek	G Čadca	4		5	5			10	24
25. – 26.	Cyril Hrubiš	G Bílovec	3	0	4	8	3		15	20
	Michal Pavelčík	G UBrod	2			8	6		14	20
27. – 29.	Ondřej Bouda	GKptJaroš	2	2		6			8	18
	Zbyněk Falt	GNeumannov	4						0	18
	Adam Ráž	GBudějo	2		5	8		3	16	18
30. – 32.	Jiří Cabal	SPŠ DvKrál	2						0	15
	Ondřej Garncarz	G Příbor	4	0		0	2	0	2	15
	Martin Kahoun	GJNerudy	2	0	5				5	15
33.	Jan Palenčar	G Martin	2						0	14
34.	Martin Podloucký	G Strážnic	4						0	12
35. – 38.	Jakub Jenis	GsvCyrMet	1						0	11
	Hana Klempová	GUBalvanJN	4		8	1	1		10	11
	Jakub Porod	G Týn nV	2			6			6	11
	Ján Zahornadský	GZborov	4						0	11
39. – 41.	Lukáš Beleš	G Čadca	4						0	10
	Jakub Benda	GJNerudy	2						0	10
	Michal Vaner	G Turnov	3						0	10
42.	Marian Kaluža	GHavlíčkov	2			0			0	9
43. – 44.	Jiří Machálek	G Holešov	3						0	8
	Petr Soběslavský	GJHeyrovs	4						0	8
45. – 46.	Daniel Sedláček	SPŠE Hav	1						0	7
	Filip Šauer	G Klatovy	4						0	7
47.	Jiří Nohavec	G Domažl	4	0					0	6
48. – 50.	Dalibor Adamčík	SPŠE Preš	2						0	5
	Jan Staněk	GKptJaroš	3						0	5
	Zdeněk Vilušínský	G Turnov	4			0	2		2	5
51. – 53.	Tomáš Ehrlich	G Holešov	2						0	2
	Petr Musil	G MBuděj	3						0	2
	Martin Vařák	G Bílovec	2						0	2
54. – 56.	Florián Danko	SPŠEtech	2						0	1
	Tamara Kuštárová	GBiling	0						0	1
	Petr Zimčík	G UBrod	1						0	1
57. – 58.	Miroslav Hovorka	GJateční	4						0	0
	Adrián Lachata	G Svidník	3						0	0