Výsledková listina devatenáctého ročníku KSP po druhé sérii

		škola	*ročník*	*sérií*	*1921*	*1922*	*1923*	*1924*	*1925*	*1926*	*suma*	*celkem*
1.	Jakub Kaplan	GJKTyla	3	12		12	10		13	8	42,7	81,7
2.	Marek Nečada	G Jihlava	3	2			6	8	10	6	39,8	80,6
3.	Zbyněk Konečný	GKptJaroš	4	14	6	6	10	10	8	6	32,4	75,4
4.	Matěj Korvas	GJSeiferPH	4	2	6	5	10		6	2	31,3	74,7
5.	Vojtěch Kolář	G Neratov	3	2	0	5,5	4	9	6	6	36,2	70,4
6.	Petr Kratochvíl	G SvětláNS	4	16	5		9		8	12	30,6	69,6
7.	David Brazdil	G Zlín	2	2	6	5,5	6		6	6	33,4	69,4
8.	Jan Michelfeit	G HBrod	3	2	6	5,5	5			10	30,9	67,2
9.	Trung Ha duc	GMasaryk	1	2	2		10		6	10	34,7	67,1
10.	Vlastimil Dort	GŠpitálsPH	1	2	6	3	6		7		29,4	66,6
11.	Jan Žák	G HBrod	2	2	1	5			6	11	29,2	64,5
12.	Libor Plucnar	GPBezruče	2	2	6	2	6		7		28,5	59,0
13.	Pavel Klavík	G Chrudim	4	16	6		10		7	4	23,3	56,1
14.	Pavel Veselý	G Strakon	2	4			5		6	4	22,7	56,0
15.	Kristýna Krejčová	G Tišnov	4	5		2	5		6		18,6	52,6
16.	Josef Pihera	G Strakon	4	12						11	10,9	50,9
17.	Zbyněk Jiráček	GArabská	3	2			6		6		18,0	50,6
18.	Viktor Lucza	G Rožňava	3	2	6	3	9				20,3	50,4
19.	Miroslav Klimoš	G Bílovec	2	16						11	10,3	46,4
20.	Jakub Balhar	GJNerudy	4	4	6	1,5	4		3	2	22,2	42,9
21.	Karel Pajskr		2	2	6	2	6		6	4	31,3	42,5
22.	Petr Onderka	G VKlobou	4	7	6		10			6	23,6	41,4
23.	Tomáš Herceg	G Třebíč	4	13		3			7	8	16,1	40,3
24.	Rudolf Rosa	G Kladno	4	3							0,0	39,6
25.	Radim Cajzl	G NMnMor	0	7					7		8,7	38,4
26. – 27.	Jiří Maršík	GJKTyla	3	6							0,0	37,2
	Tomáš Sýkora	G VKlobou	3	5		3	1	4	6		20,8	37,2
28.	Ondřej Bouda	GKptJaroš	4	8	6	3			7		17,9	36,9
29.	Martin Kahoun	GJNerudy	4	7	6				7		14,7	36,5
30.	Lukáš Kripner	G Litvínov	1	2					6		9,6	36,0
31.	Jan Kohout	G Roudnice	4	7	1	1,5	1		6		13,0	35,8
32.	Lucia Simanová	GGrösslin	3	1							0,0	34,9
33.	Karel Tesař	SPŠEPlzeň	1	1							0,0	34,8
34.	Dušan Rychnovský	G Hranice	3	2							0,0	32,1
35.	Vojtěch Tůma	G Jihlava	3	1	6	2,5	7		7		30,1	30,1
36.	Jan Matějka	GJírovco	2	1							0,0	29,0
37.	Jakub Slavík	GJKTyla	3	1							0,0	28,8
38.	Martin Němec	G Ledeč	3	1	6	2	5		6		27,6	27,6
39.	Václav Strnad	GArabská	3	1							0,0	27,5
40.	Radim Pechal	SPŠ Rožnov	4	3							0,0	26,3
41.	Mirek Jarolím	GMikuláš	1	2		2					3,7	24,7
42.	David Škorvaga	G Kralupy	4	2							0,0	24,0
43.	Stanislav Fořt	G Tábor	0	2	1	0,5	0		1		5,8	21,0
44.	Jakub Červenka	GŠpitálsPH	1	1					6	6	19,4	19,4
45.	Jan Kučera	G Polička	4	1							0,0	19,0
46.	Jan Sixta	G Brandýs	4	1							0,0	18,2
47.	Martin Majer	SPŠÚžlabin	2	4					7		9,8	16,4
48.	Milan Klášterka	SPŠKlatovy	3	2					5		8,7	16,1
49.	Jakub Pavlík jn.	G Kladno	4	5						2	3,6	15,6
50.	Karolína Burešová	G ČLípa	0	1	6		5				13,9	13,9
51.	Tomáš Volf	G Tábor	0	1							0,0	12,4
52.	Richard Jedlička	G Vlašim	3	6							0,0	8,9
53.	Miroslav Jančařík	G UBrod	3	3							0,0	8,5
54.	Jan Papoušek	GKptJaroš	3	1							0,0	8,0
55.	Jan Krajdl	SPŠÚžlabin	2	3							0,0	6,6
56.	Martin Pástor	SPŠAlejová	2	1							0,0	6,0

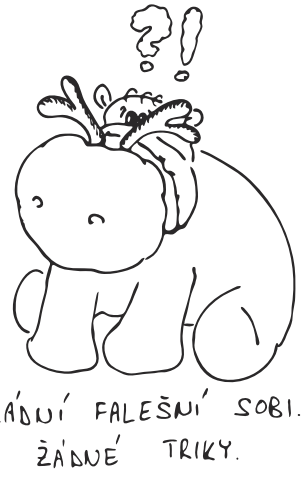
Milí řešitelé!

Mohli jsme pro vás připravit normální sérii. A krásnou. Ale my jsme řekli: „NE!“ Chceme být fér, a tak novým i stávajícím řešitelům nabízíme dalších šest úložek. A bez písemné smlouvy.

Ale pozor – tato nabídka platí pouze do 13. března 2007. 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 můžete nalézt na stránkách *http://ksp.mff.cuni.cz/*, diskutovat můžete na fóru *http://ksp.mff.cuni.cz/forum/* a záludné dotazy organizátorům lze zasílat e-mailem na adresu *ksp@mff.cuni.cz*.



Ztráty a nálezy

Prosíme stydlivého řešitele, který poslal papírovou poštou nepodepsaná řešení úloh 19-2-1 a 19-2-2, aby se nám ozval na *ksp@mff.cuni.cz*. Budou mu přiděleny body. Zvláštní znamení je červený inkoust použitý pro tisk řešení (bylo to jediné řešení druhé série v této barvě :–).

Zadání čtvrté série devatenáctého ročníku KSP

Milí čtenáři, vzpomínáte si ještě na mne? Jsem detektiv Přesprst a rozhodl jsem se zvěčnit svůj životní příběh. Bohužel se nenašlo nakladatelství, které by o něj mělo zájem, a tak jsem přijal nabídku organizátorů KSP, kteří se rozhodli můj příběh vydat a ukázat na něm, že i běžná detektivní práce vyžaduje nemalé informatické znalosti. Takže, kde jsem to jen skončil...
Seděl jsem v Zamřížově kanceláři a poslouchal jeho výklad o vztazích mafiánů. Nad mým životem se stahovala mračna. Možná by to bylo ještě smutnější, kdyby nad ním slunce už dávno nezapadlo. Zamříž dokončil svou řeč. Chvíli jsme na sebe mlčky hleděli a nastalé ticho rušilo jen tikání nástěnných hodin.
„Takže tu budeme jen tak sedět a čekat, až si pro nás přijdou?“ ozvala se jízlivě Isabela.
Zamříž se zamyslel. Znal jsem tenhle výraz v jeho tváři. Tohle rozhodně nebude procházka růžovým sadem.
„Tohle rozhodně nebude procházka růžovým sadem,“ promluvil nakonec. „Pokud by se nám podařilo prokázat styky se zahraničím, mohli bychom do celé věci zatáhnout Interpol a požádat o posily, ale jinak nevím.“
„Není nic snazšího,“ prohlásila Isabela. „V těch materiálech z jeho trezoru by měl být i soupis zahraničních plateb.“

19-4-1 Finanční toky — 8 bodů

Máme k dispozici kompletní přehled všech plateb mezi jistými podezřelými organizacemi. Tento přehled tvoří orientovaný graf (viz kuchařka 19-3), ve kterém jsou jednotlivé organizace vrcholy a z i do j vede hrana, právě když organizace i převedla peníze na účet organizace j.

Tento graf máme již uložený jako matici sousednosti. Matice sousednosti M má velikost $N \times N$ (kde N je počet vrcholů grafu) a na pozici M_{ij} je 1, pokud z i do j vede hrana, a 0 v opačném případě. (Na M_{ii} je vždy nula.)

Navrhněte algoritmus, který v takto zadaném grafu nalezne stok. Stok je vrchol, do kterého vedou hrany ze všech ostatních vrcholů a z něho samotného už žádná hrana nevede.

Uvědomte si, že Přesprstovi a Isabele jde o život, a tak by váš algoritmus měl pracovat opravdu rychle. Navíc můžete předpokládat, že matici sousednosti již máte v paměti, a tak nemusíte připočítávat čas potřebný k jejímu načtení.

Příklad: Graf zadaný maticí

0	1	0	1
0	0	0	0
1	1	0	0
1	1	1	0

má právě jeden stok – vrchol 2.

„Ano to je ono!“ zajásal Zamříž. „Podívejte, vy dva. Už teď u mě máte metál, ale aby to klaplo, musím vyřídit pár telefonů. Co kdybyste si zatím dali kafe nebo tak něco, a já se o to postarám.“ A se sluchátkem u ucha nám jemně naznačil, abychom prozatím vypadli z jeho kanceláře.
Víčka jsem měl pěkně těžká a kafe bodlo. Chutnalo příšerně, i když v tuhle chvíli mi to bylo úplně jedno. Isabela stála u okna a pozorovala dění na ulici. Z venku se ozval pištivý zvuk pneumatik rychle projíždějícího auta. Přiskočil jsem k ní a trhnutím ji odtáhl od okna.
„Co blázníš,“ stačila ze sebe vypravit, než její slova přehlušila střelba a zvuk tříštícího se skla. Chvíli jsme mlčky hleděli na roztříštěné sklo a rozdýchávali tenhle incident.
„Asi bych ti měla poděkovat,“ prolomila ticho Isabela.
„Řekl bych, že tím jsme vyrovnáni,“ usmál jsem se.
Ze své kanceláře vykoukl Zamříž: „Obvolal jsem několik lidí a dal věci do pohybu. Bohužel náš byrokratický aparát funguje někdy velmi pomalu...“

19-4-2 Byrokratický aparát — 10 bodů

Při schvalování určité záležitosti postupuje byrokratický aparát následujícím způsobem. Každý úředník má na počátku na svém stole nějaký dokument. Úředník si dokument přečte, orazítkuje a pošle ho dalšímu úředníkovi. Práce všech úředníků končí v okamžiku, kdy mají všichni úředníci na stole dokument, který schvalovali jako první (tzn. celý aparát se vrátí do výchozího stavu).

Aby to nebylo tak jednoduché, jsou zavedena speciální pravidla, komu má úředník předat dál orazítkovaný dokument. Každý úředník i má určeného právě jednoho úředníka $f(i)$, kterému své orazítkované dokumenty předává. Aby se dokumenty nehromadily u některých úředníků, zatímco jiní budou bez práce, dostává každý úředník dokumenty od právě jednoho úředníka. Tedy každý úředník jeden dokument pošle dál a jeden od někoho dostane, a tak má stále stej-

ně práce. A protože úřad je úřad, někteří úředníci klidně mohou orazítkovat tentýž dokument vícekrát.

Navrhněte algoritmus, který pro dané přiřazení úředníků f zjistí, kolik minimálně schvalovacích kroků (více než nula) bude potřeba, aby schvalovací proces skončil, tj. aby každý úředník měl na stole dokument, který schvaloval jako první.

Úředníci jsou očíslování od 1 do N a pravidla předávání dokumentů jsou zadána jako seznam ($1 \to 3, 2 \to 1, \ldots$). Nezapomeňte, že mohou existovat izolovaní úředníci, kteří dokumenty posílají sami sobě (tzn. pravidla $i \to i$).

Příklad: Pro 5 úředníků a pravidla ($1 \to 5, 2 \to 4, 3 \to 1, 4 \to 2, 5 \to 3$) trvá schvalovací proces 6 kroků.

„Úředního šimla ke cvalu nepřinutíš," pokýval jsem smutně hlavou.
Počkali jsme, než se setmělo a pak jsme se společně se Zamřížem a jeho kolegou vydali opatrně dolů na parkoviště. Vypadalo to, že nás nikdo nesleduje. Nasedli jsme do auta a vydali se na cestu. Isabela se schoulila na zadní sedačce a začala podřimovat. Byl to dlouhý den. Víčka mi těžkla a chtělo se mi spát. Projeli jsme kolem známého motorestu, kde nás dnes dopoledne Zamříž vyzvedl. Ale tahle silnice přece vede ...
„Kam to jedeme?" zeptal jsem se.
„Uvidíš," usmál se Zamříž. „Už tam skoro jsme."
Teď to do sebe začínalo zapadat. Přednáška o „pevných vztazích", střelba na Isabelu i směr naší cesty. Ten prašivý skunk Zamříž mě podrazil. Po tolika letech přátelství!
„Proč?! Pro prachy? Nebo je v tom snad něco jiného?!"
„Neber si to osobně," odpověděl Zamříž s klidnou tváří. „Není to jen pro prachy. Je to pro spoustu prachů ... "
Dál jsem nečekal. Udeřil jsem ho vší silou a strhl volant. Do rvačky se vložil Zamřížův kolega a nejspíš bych i prohrál, kdyby automobil nesjel z cesty a nenarazil do stromu. Probral jsem se. Hlava bolela jako střep a ruka také nevypadala dobře. Poplácal jsem Isabelu po tváři, aby se probrala. Zamříž i jeho kolega byli v bezvědomí. Nebyl čas na hrdinství. Buď se probudí a pokusí se nás zabít, nebo přijede policie a přišije nám dvojnásobnou vraždu. Navíc nevím, komu věřit. Sebral jsem Zamřížovi služební zbraň a vytáhl Isabelu z auta.
„Musíme pryč, a honem! Nedaleko odsud je železnice. S trochou štěstí chytíme nákladní vlak."
Běželi jsme, co nám síly stačily a modřiny dovolily. Ani nevím, jak dlouho nám to trvalo, ale nakonec jsme doběhli k trati. A dokonce jsme měli i štěstí. Ozvalo se houkání a za zatáčkou se objevil nákladní vlak...

19-4-3 Naskakování na vlak — 11 bodů

Naskakování na vlak není věc jednoduchá. Přesprst a Isabela jsou navíc celí potlučení, a tak si musí zatraceně dobře rozmyslet, na který vagón naskočí a na který ne. Navíc musí počítat s tím, že se jim nemusí podařit na nějaký vagón naskočit, takže by rádi věděli, jestli se podobný vagón (resp. posloupnost vagónů) vyskytuje ve vlaku víckrát. A tady je příležitost pro vás, abyste se zkoumáním vlaku pomohli.

Vlak si představte jako řetězec délky N, kde každé písmeno představuje jeden vagón (např. `U` je uhelný vagón, `P` je poštovní vůz atp.). Dále máte dáno číslo k ($k \leq N$) a máte zjistit, kolik navzájem různých podřetězců délky k se v řetězci (tedy ve vlaku) vyskytuje. Zároveň tyto podřetězce a počty jejich výskytů vypište.

Pozor, vlak už se blíží, takže byste to měli spočítat pekelně rychle. Nebojte se k tomu využít znalostí, které načerpáte z aktuální kuchařky, avšak pokud vymyslíte ještě efektivnější a podlejší postup, bodová odměna vás nemine.

Příklad: Pro řetězec (vlak) `UPDUPDUDUP` a $k = 3$ jsou nalezené podřetězce

`UPD`	2×
`PDU`	2×
`DUP`	2×
`DUD`	1×
`UDU`	1×

Podařilo se nám naskočit na poloprázdný vagón se dřevem. Nebyl příliš pohodlný, ale hned sousední vagón převážel poštovní zásilky. Uvelebili jsme se mezi balíky a pytlem s dopisy a drncání vlaku nás pomalu ukolébalo.
„Hej! Ty ... vstávat!"
Probudil jsem se a zamžoural do světla před sebou. Očividně bylo ráno a vlak už nedrncal. Přede mnou stála postava oblečená v pošťácké uniformě a mířila na nás revolverem.
„Tak pohyb, vy dva!" zarámusil pošťák a naznačil revolverem, abychom se zvedli. Odvedl nás do malého skladiště poštovních zásilek, které se krčilo hned vedle kolejí.
„Tady počkáte, než vyložím zásilky. Pak uvidíme, co s vámi uděláme."
Strčil nás dovnitř, zamkl dveře a odešel.
„To je prostě skvělé. Co teď budeme dělat, hm?" pronesla skoro vyčítavě Isabela a posadila se na poštovní balík.
„Já osobně bych si dal snídani."
„Cože? Ty bys sis dal ... " rozkřikla se, ale pak se zarazila. Podívala se na mě a rozesmála se na celé kolo. Je zajímavé, jak některé věci přijdou člověku veselé, když je až po uši v průšvihu.
Vykoukl jsem z okénka. Pošťák právě skládal veliký balík na váhu. Chvíli jsem pozoroval, jak si hraje se závažími, když v tom mě napadla spásná myšlenka.
„Hej, pane pošťáku, nechcete s tím pomoct?"

19-4-4 Váhy — 6+4 bodů

Pošťák zápasí s váhami, protože nemají vhodnou sadu závaží. Navrhněte optimální sadu závaží, která bude postačovat na zvážení libovolného předmětu o celočíselné hmotnosti 1 až m kilogramů s přesností na jeden kilogram. Předmět považujeme za odvážený, když se misky vah ustálí v rovnovážné poloze, a pozor – závaží můžete pokládat na obě misky vah.

Aby vám pošťák věřil (a ocenil vás šesti body), musíte také dokázat, že vámi navržená sada závaží je funkční (tedy že s ní umíte zvážit libovolný přípustný předmět). Pokud uvedete i důkaz, že daná sada je optimální (tzn. neexistuje menší sada, která by také byla funkční), přidá vám pošťák 4 body navrch.

Pokud existuje optimálních sad více, stačí najít jednu libovolnou.

Poznámka: Při vážení 1 kg předmětu potřebujeme skutečně jedno kilogramové závaží. Nestačí vzít např. 2 kg závaží a předměty, které jsou lehčí, prostě prohlásit za jednokilové.

Příklad: Pro zadané $m = 3$ je jedna z možných optimálních sad závaží $\{1, 2\}$. Věci o hmotnosti 1 a 2 kilogramy zvážíme přímo, 3 kg odvážíme tak, že dáme obě závaží na opačnou misku než vážený předmět.

Tato sada je optimální, protože menší sada by měla pouze jedno závaží a snadno nahlédneme, že s jedním závažím umíme určit hmotnost pouze u předmětů, které váží stejně, jako závaží samo.


```
% KSP 19-2-6 2.Permutující slepice

perm([],[[]]).
perm(S,P) :- perm([],S,[],P).

% Procyklíme přes všechny prvky v zadaném seznamu.
% Každý prvek utrhneme se seznamu,
% rekurzivně vytvoříme všechny permutace se seznamu
% bez tohoto prvku a tento prvek předřadíme
% před všechny tyto částečné permutace
perm(_,[],P,P).
perm(Sused,[X|S],P,R) :-
        spoj(Sused,S,SbezX),            % vytvoříme seznam bez daného prvku
        perm(SbezX,Y),                  % z tohoto seznamu uděláme všechny permutace
        pripoj(X,Y,P,Q),                % před tyto permutace předřadíme daný prvek
        perm([X|Sused],S,Q,R).          % zavoláme se na další prvek
                                        % (cyklíme seznamem dál)

% pripoj(Prvek,Sezn,MeziVysl,Vysl)

% Předřadí Prvek jako hlavu před všechny seznamy v seznamu
% seznamů Sezn (permutace bez prvku Prvek)
% Takto vytvořené permutace přidá do seznamu již
% vytvořených permutací MeziVysl a výsledek uloží do Vysl

pripoj(_,[],P,P).
pripoj(X,[Y|Ys],P,[[X|Y]|R]) :- pripoj(X,Ys,P,R).

% spoj(Sezn1,Sezn2,VyslSezn) spojí seznamy Sezn1 a Sezn2
% za sebe do VyslSezn
spoj([], Sezn2, Sezn2).
spoj([Hlava|Telo], Sezn2, [Hlava|Sezn3]) :-     % předřadíme Hlavu pred Sezn3,
        spoj(Telo,Sezn2,Sezn3).                 % který se nám vrátí z rekurze



% KSP 19-2-6 3.Palindromické slepice

% Sezn je palindromem prave tehdy,
% shoduje-li se přesně se svým obráceným seznamem
palindrom(Sezn) :- obrat(Sezn,Sezn).

% obracení uděláme známou technikou akumulátoru,
% aby bylo lineární
obrat(Sezn,Vysl) :- akumulator(Sezn,[],Vysl).

% ze zadaného seznamu trháme prvky
% a skládáme je před sebe do druhého seznamu,
% až nám dojdou prvky v prvním seznamu,
% v druhém seznamu jsou prvky v opačném pořadí
akumulator([],Sezn,Sezn).
akumulator([A|S1],S2,S3) :- akumulator(S1,[A|S2],S3).
```

```
{
        int i=0, j=mid, k=0;
        while (k < n)
                if (j >= n || i < mid && (by_x ? (a[i].x <= a[j].x) : (a[i].y <= a[j].y)))
                        temp[k++] = a[i++];
                else
                        temp[k++] = a[j++];
        for (i=0; i<n; i++)
                a[i] = temp[i];
}

// Třídění pole stromů podle Y (nerekurzivní varianta MergeSortu, když už máme slévání)
void sort_by_y(void)
{
        for (int i=1; i<N; i *= 2)
                for (int j=0; j+i < N; j += 2*i)
                        merge(trees+j, i, min(2*i, N-j), 0);
}

// Jádro pudla -- rekurzivní funkce na hledání vzdálenosti.
// Na vstupu stromy setříděny podle Y, na výstupu už podle X.
int find_dist(tree *a, int n)
{
        if (n < 2)                              // Jo tyhle triviální případy...
                return INFTY;
        int mid = n/2;                          // Rozdělíme přesně v polovině
        int mid_y = a[mid].y;                   // Pozice dělící přímky
        int d1 = find_dist(a, mid);             // Rekurzivně zpracujeme poloviny
        int d2 = find_dist(a+mid, n-mid);
        int d = min(d1, d2);                    // Dosavadní minimum
        merge(a, mid, n, 1);                    // Dotřídíme podle X

        int p = 0;                              // Najdeme body ležící na švu
        for (int i=0; i<n; i++)
                if (sqr(a[i].y - mid_y) < d)
                        temp[p++] = a[i];
        for (int i=0; i<p; i++)                 // Porovnáváme je se sedmi následníky
                for (int j=i+1; j<p && j<=i+7; j++)
                        d = min(d, distq(temp[i], temp[j]));
        return d;                               // Výsledná vzdálenost (kvadratická)
}

int main(void)
{
        scanf("%d", &N);
        for (int i=0; i<N; i++)
                scanf("%d%d", &trees[i].x, &trees[i].y);
        sort_by_y();
        printf("minimální vzdálenost: %f\n", sqrt(find_dist(trees, N)));
        return 0;
}
```

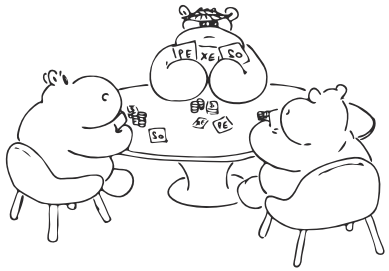
Úloha 19-2-6 – Prolog – program

```
% KSP 19-1-6 1.Příliš těžké slepice

% prostredni(Sezn, Prost)       Prost je prostřední prvek Sezn

% Vyšleme seznamem dva signály, jeden dvakrát rychlejší nez druhý.
% Až dojede rychlejší signál na konec, pomalejší ukazuje na
% prostředek seznamu.

prostredni([A], [C|T2], C).                     % seznam liché délky
                                                % rychlý signál A dojel na konec,
                                                % pomalý C je prostředek

prostredni([A,B], [C,D|T2], D).                 % seznam sudé délky
                                                % rychlý signál A dojel na konec,
                                                % pomalý D je prostředek
                                                % (dle zadání je vic vpravo)

% z prvního seznamu reprezentující rychlý seznam utrhneme dva prvky
% (posuneme se o dva prvky), z druhého seznamu utrhneme jeden prvek
% (posuneme se o jeden prvek), zavoláme se rekurzivně na zbytky
% seznamů T1 a T2 a necháme si z rekurze vrátit výsledek Prost
prostredni([A,B|T1],[C|T2],Prost) :- prostredni(T1,T2,Prost).
```


„To je dobré," poplácal mě pošťák po zádech. „Nechtěl bys pracovat u nás?"
„No, víš...," začal jsem nesměle s podíval se na Isabelu, která seděla na poštovním balíku.
„Nic mi neříkej. Úplně tě chápu," mrknul na mě šibalsky. „A teď odsud zmizte, než přijde šéf."
Vydali jsme se z nádraží do města. K čertu, vždyť jsem ani nevěděl, co je to za město. Ale zůstat tady nemůžeme. Musíme zmizet za hranice. Zběžně jsem si prošacoval kapsy. Jen pár drobáků, navlhlý doutník a zbraň, kterou jsem sebral Zamřížovi.
„Musíme sehnat nějaké peníze a vypadnout ze země," nahodil jsem, aby řeč nestála.
„A co chceš dělat?" podívala se na mě Isabela. „Vykrást banku?"
Rozhlédl jsem se po ulici, ale nikde žádná banka v dohledu. Zato na protější straně ulice jsem zahlédl bar. Nápis nade dveřmi „U Tří Es" hlásal, že se jedná o nefalšované hráčské doupě.
„A co takhle zkusit štěstí..."

19-4-5 Hazardní hra — 10 bodů

Hazardní hráči jsou samozřejmě všemi mastmi mazaní. Nechali Přesprsta vyhrát několik her Pokeru a teď ho chtějí oškubat na jiné, nepříliš známé hře.



Pravidla jsou následující. Bankéř vyhlásí číslo m, což je shodou okolností částka, o kterou se hraje. Hráči se poté snaží vymyslet, jak co nejrychleji tuto částku vysázet na stůl. Přitom ovšem smí používat pouze předem definované tahy:

- přidat jednu minci (všechny mince mají stejnou hodnotu, takže tento tah je vlastně zvýšení sumy o 1)
- odebrat jednu minci (snížení sumy o 1)
- dorovnat na desetinásobek aktuální sumy (tedy vynásobit deseti)
- vydělit aktuální sumu deseti (zaokrouhluj dolů)

Kdo vysází danou částku nejrychleji, vyhrává všechny peníze, které jsou momentálně na stole. Přesprst opravdu potřebuje vyhrát, a tak se neobejde bez vaší pomoci.

Příklad: Pro číslo $m = 192$ jsou nejrychlejší tahy: $+1, +1, *10, -1, *10, +1, +1$.

Klidným krokem jsem vyšel ven a přepočítával vyhrané peníze. Isabela na ně užasle zírala. „Jak jsi to dokázal?"
„Stačí mít trochu štěstí a umět počítat," usmál jsem se. Schoval jsem peníze do kapsy a vykročil směrem k nádraží. Ušli jsme sotva pár kroků, když v tom hned vedle nás zastavilo u chodníku policejní auto...

To be continued...

19-4-6 Prolog — 14 bodů

Milí znalci a přátelé Prologu,

tentokrát vás čeká velmi zajímavá, ale také náročná kapitola. Dozvíte se něco o negaci v Prologu, ale také o myších, logice, filozofii a jiných nebezpečných věcech.

Jemný začátek – vstup a výstup

Při startu programu je aktuální vstup nastaven z klávesnice, aktuální výstup na obrazovku.

Základní jednotkou, kterou můžete načíst nebo vypsat, je term. Predikát `read(T)` načte celý aktuální term (číslo, znak, řetězec, struktura) a unifikuje ho do proměnné `T`. Predikát `write(T)` vypíše term `T`. Pokud je v termu `T` zunifikovaná proměnná, vypíše se místo ní její hodnota.

Pokud chcete číst a vypisovat znaky, použijte predikáty:

`get0(Z)`	přečte znak
`get(Z)`	přečte znak a ignoruje přitom řídící znaky
`put(Z)`	vypíše znak (nevypisuje řídící znaky)
`tab(N)`	vypíše N mezer
`nl`	nový řádek

Příklad:

```
?-write('Zadej zvire: '), read(Zvire), write(Zvire).
Zadej zvire: kachna.
kachna
Zvire = kachna
Yes
```

Příklad: Tento predikát vypíše seznam, každá položka bude na novém řádku:

```
vypis([]).
vypis([H|T]) :- write(H), nl, vypis(T).
```

Pozor: Vstup a výstup je tak trochu neprologovský – jistě víte, že když Prolog splňuje nějaký predikát `p`, postupně splňuje jednotlivé predikáty v jeho těle. Pokud nějaký z nich neuspěje, Prolog „zapomene" dosud provedené predikáty z těla `p` a hledá na dalších řádcích jiná těla splňovaného predikátu `p`. U vstupů a výstupů ale nejde „zapomenout" už provedenou hodnotu. Pokud tedy nějaký predikát vypíše něco na obrazovku, zůstane to vypsané i tehdy, když ten predikát nakonec neuspěje.

Filozofické zastavení

Narozdíl od klasických programovacích jazyků, které přesně popisují algoritmus na řešení dané úlohy, Prolog se snaží vyřešit problém pomocí *matematické logiky*. Každá klauzule (řádek programu) odpovídá nějaké *výrokové formuli*. Příkladem výrokové formule je formule „a & b → c". Prologovský program je tedy *množinou klauzulí*, každá klauzule má svůj *význam*, který vyjadřuje nějaká logická formule. Pokud vezmeme všechny logické formule, které odpovídají všem klauzulím programu, dostaneme *význam programu*. Když pochopíme, že prologovký program je vlastně souborem logických formulí, mezi kterými je „a zároveň", všimneme si, že v Prologu nezáleží na pořadí jednotlivých klauzulí, ani na pořadí jednotlivých predikátů v jejich tělech.

Bylo by hezké, kdyby fungovala představa čistě logického jazyka, ve kterém nezáleží na pořadí vyhodnocování predikátů, ale bohužel to tak nejde. S takovým jazykem bychom toho mnoho nenaprogramovali, takže se musíme uskromnit.

Opouštíme ideály – predikát řezu

Jistě už jste přemýšleli nad tím, jak se v Prologu udělá negace a proč jsme si ji ještě neukázali. Abychom prolo-

govskou negaci byli schopni vytvořit, vysvětlíme si nejprve *predikát řezu*.

Predikát řezu slouží k vyjádření negace a zrychlení prologovských programů. Značí se vykřičníkem „!“. Názornou představu o predikátu řezu si můžeme udělat už z jeho názvu – pokud použijeme řez v nějaké větvi výpočtu, řez zakáže použít další možné větve tohoto výpočtu, tedy zakáže další backtrackování. Nejlepší bude příklad:

```
a(X) :- b(X), !, c(X).
a(X) :- d(X).
```

Prolog zde zkouší splnit první řádek. Pokud se splní predikát `b(X)`, dojdeme k predikátu řezu. Ten okamžitě uspěje, ale přitom zakáže nový vstup do predikátu, ve kterém se nachází, tedy nesmíme už znovou zkoušet splnit predikát `a(X)`. Pokud tedy neuspěje následující predikát `c(X)`, Prolog už díky řezu v prvním řádku nesmí zkoušet další možnost v druhém řádku. Odřezali jsme tedy druhou větev výpočtu. Kdyby v prvním řádku nebyl operátor řezu, Prolog by zkoušel splnit nejprve první řádek, a kdyby se mu to nepovedlo, skočil by hned na druhý tak, jak to známe.

Operátor řezu lze použít dvěma způsoby:

1. Operátor řezu jako tzv. *zelený řez*, ten nemění význam programu, pouze ho urychluje tím, že uřezává neperspektivní větve zbytečné pro výpočet, ale program by fungoval i bez něj.

2. Operátor řezu jako tzv. *červený řez*, kterým změníme průběh vyhodnocování programu.

Příklad zeleného řezu:

V tomto příkladu chceme slít dohromady dvě setříděné posloupnosti tak, aby výsledná posloupnost byla setříděná. Například z posloupností `[1,3,5]` a `[2,4,6]` dostaneme `[1,2,3,4,5,6]`.

```
slij([X|A],[Y|B],[X|C]) :- X=<Y, !, slij(A,[Y|B],C).
slij([X|A],[Y|B],[X,Y|C]) :- X=Y, !, slij(A,B,C).
slij([X|A],[Y|B],[Y|C]) :- Y=<X,slij([X|A],B,C).
slij(A,[],A).
slij([],B,B).
```

Operátor řezu v prvních dvou řádcích se dá chápat jako rada – pokud je například `X=<Y`, nemá cenu zkoušet, jestli je `X=Y` nebo `Y=<X`, protože víme, že to vždy bude nepravda. Operátor řezu zde tedy interpreteru říká, že pokud uspělo `X=<Y`, ostatní větve už by neuspěly.

Příklad červeného řezu:

Červený řez se často používá v predikátech, které při prvním spuštění dají dobrý výsledek, ale při odmítnutí středníkem nebo při znovuzavolání jiným predikátem by mohly začít dávat nesmysly. Příkladem je třeba tento predikát, který má odstranit všechny výskyty prvku `X` v predikátu `Sezn`:

```
odstran(_,[],[]).
odstran(X,[X|Y],Z) :- odstran(X,Y,Z).
odstran(X,[Q|Y],[Q|Z]) :- odstran(X,Y,Z).
```

Prohlédněte si tuto konverzaci s interpreterem Prologu:

```
?-odstran(b,[a,b,c],Vysl).
Vysl = [a, c] ;
Vysl = [a, b, c] ;
No
```

To ale není správné chování. Na odmítnutí středníkem měl Prolog reagovat okamžitým `No.`, protože `[a,b,c]` není přece původní seznam s vymazaným prvkem `b`. Problém je v tom, že po odmítnutí uživatelem zkouší Prolog jiné možnosti, jak splnit predikát `odstran`, a použije třetí variantu `odstran` i v případě, že `X=Q`. Tuto chybu odstraníme tím, že přidáme operátor řezu do druhého řádku, čímž zakážeme případné použití třetího řádku v případě, že `X=Q`:

```
odstran(_,[],[]).
odstran(X,[X|Y],Z) :- !, odstran(X,Y,Z).
odstran(X,[Q|Y],[Q|Z]) :- odstran(X,Y,Z).
```

Jiný příklad: Chceme vložit do seznamu prvek, pokud tam ještě není:

```
vloz(A,Sezn,Sezn) :- prvek(A, Sezn), !.
vloz(A,Sezn,[A|Sezn]).
```

Predikát fail

Predikát `fail` má jedinou funkci – okamžitě si vynutí selhání. Tedy napíšeme-li `predikat :- fail`, tento predikát nikdy neuspěje. Na první pohled se může zdát trošku divné, proč bychom mohli potřebovat takový predikát, ale ve skutečnosti je to jeden z nejužitečnějších predikátů v Prolog, protože ho potřebujeme pro negaci.

Negace

Jak už jsme prozradili, negaci v Prologu tvoříme pomocí řezu. Ukážeme si tedy definici vestavěného predikátu `not(A)`, který uspěje, pokud neuspěje `A`.

```
not(A) :- A, !, fail.
not(A).
```

Predikát `not(A)` se nejprve pokusí splnit cíl `A`. Pokud se `A` splní, zakážeme zkoušet další větve výpočtu a přikážeme predikátu selhat. Pokud se cíl `A` nesplní, Prolog zastaví vyhodnocování tohoto řádku, tudíž se nedostaneme ani k operátoru řezu, takže nezakážeme další větev, která se automaticky splní.

Vidíte, že bez operátoru řezu a bez `fail` bychom toto chování neuměli přikázat.

Příklad: Opět chceme vložit do seznamu prvek, pokud tam ještě není:

```
vloz(A,Sezn,[A|Sezn]) :- not(prvek(A, Sezn)).
vloz(A,Sezn,Sezn) :- prvek(A,Sezn).
```

Kvíz

⋆ Máme následující konstrukci:

```
p(X,Y) :- X > Y, !, fail.
p(X,Y) :- write(X), tab(1).
p(X,Y) :- X1 is X + 1, p(X1,Y).
```

Co se stane, zavoláme-li:

```
?-p(3,6),fail.
```

1. Vypíše se `6 5 4 3`
2. Nekonečná smyčka
3. Vypíše se `3 4 5 6`
4. `Yes.`
5. Nastane běhová chyba.

⋆ Máme konstrukci:

```
p :- !, p.
```

Co udělá dotaz:

```
?-p.
```

1. Nastane běhová chyba
2. `No.`
3. Nekonečná smyčka
4. `Yes.`


```
   Pozice:=0;
   repeat
      Syn:=Pozice * 2 + 1;
      if Syn >= VelikostHaldy then break;   {už jsme úplně dole}
      if Syn+1 < VelikostHaldy then
         {uvažovaný vrchol má 2 syny, tak vybereme úkol s vyšší odměnou}
         if (Halda[Syn+1].Odmena > Halda[Syn].Odmena) then inc(Syn);

      if Halda[Syn].Odmena < Halda[Pozice].Odmena then break;
         {pokud všechny zakázky níž už jsou hůř placené, tak jsme hovovi ...}

      Swap:=Halda[Syn];
      Halda[Syn]:=Halda[Pozice];
      Halda[Pozice]:=Swap;
      Pozice:=Syn; {prohodíme, aby to na dané hladině bylo v pořádku a klesneme níž}
   until false;   {ven budeme skákat pomocí breaku}
end;


begin
   NactiVstup();
   if (N < 0) then begin
      writeln('Není co dělat.');
   end else begin
      SeradDleTerminu(1,N);

      VelikostHaldy:=0;      {Inicializace haldy}
      Cas:=Ukoly[1].Termin;  {Máme setřízeno -> Tohle je maximální uvažovatelný čas}
      VykonanychUkolu:=0;    {Zatím jsme nic neudělali}
      DalsiUkol:=1;          {a taky jsme se ještě na nic nepodívali}

      while (Cas > 0) do begin  {Projdeme všechny časy odzadu}
         while ((DalsiUkol <= N) and (Ukoly[DalsiUkol].Termin = Cas)) do begin
            PridejDoHaldy(Ukoly[DalsiUkol]); {přidáme do haldy všechny úkoly, které končí v tento čas}
            inc(DalsiUkol);
         end;

         inc(VykonanychUkolu);
         Vykonane[VykonanychUkolu]:=Halda[0].CisloUkolu; {nejvýhodnější je v kořeni haldy}
         VyradKorenZHaldy();                  {a vyřadíme ho, je už hotový}

         if VelikostHaldy = 0 then begin
            if DalsiUkol > N then
               break      {už jsme udělali všechno, co se dalo a ještě nám zbyl čas}
            else
               Cas:=Ukoly[DalsiUkol].Termin;
               {nějakou dobu nemáme co dělat a tak skočíme rovnou na další zajímavou položku}
         end else
            dec(Cas);  {Jinak se jen posuneme zase v čase o trochu zpět}
      end;

      VypisVysledek();
   end;
end.
```

Úloha 19-2-5 – Hluboký les – program

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

#define MAX 1000                              // Maximální velikost vstupu
#define INFTY 1000000000                      // Nekonečno :-)

typedef struct {                              // Pozice jednoho stromu
        int x, y;
} tree;

int N;                                        // Počet stromů
tree trees[MAX];                              // Stromy
tree temp[MAX];                               // Pole pomocné víceúčelové

// Minimum a druhá mocnina
int min(int x, int y) { return (x < y) ? x : y; }
int sqr(int x) { return x*x; }

// Místo vzdáleností počítáme vždy jejich druhé mocniny, předpokládáme, že jsou <INFTY.
int distq(tree a, tree b) { return sqr(a.x - b.x) + sqr(a.y - b.y); }

// Slití dvou setříděných úseků (a[0...mid-1], a[mid...n-1]) do jednoho (a[0...n-1]).
// Pokud by_x>0, třídíme podle X, jinak podle Y.
void merge(tree *a, int mid, int n, int by_x)
```

```
                    tj. zatím jsme uvažovali jen vyšší časy.}

    VelikostHaldy:integer;
    Halda:array[0..MaxN-1] of TUkol;  {Halda, ze které vybíráme nejvhodnější úkol pro daný čas.}

    VykonanychUkolu:integer;  {U kolika úkolů už jistě víme, že je uděláme a kdy.}
    Vykonane:array[1..MaxN] of integer; {Čísla zakázek, co doopravdy uděláme.}

    Cas:integer;     {Čas, pro který se rozhodujeme, co uděláme.}

procedure NactiVstup();
var i:integer;
begin
   readln(N);
   for i:=1 to N do with Ukoly[i] do begin
      readln(Termin,Odmena);
      CisloUkolu:=i;
   end;
end;

procedure VypisVysledek();
var i:integer;
begin
   write('Nejvýhodnější pořadí je: ');
   for i:=VykonanychUkolu downto 1 do begin {máme je uloženy v opačném pořadí, než je je třeba vykonat}
      write(Vykonane[i],' ');
   end;
   writeln;
end;

procedure SeradDleTerminu(Min,Max:integer);
{Seřadí dle termínu dokončení sestupně, tj. nejméně spěchající úkoly na konec.}
{Je to obyčejný QuickSort.}
var L,R:integer;
    Pivot:integer;
    Swap:TUkol;
begin
  L:=Min; R:=Max;
  Pivot:=Ukoly[(Min + Max) div 2].Termin;
  repeat
    while Ukoly[L].Termin>Pivot do inc(L);
    while Ukoly[R].Termin<Pivot do dec(R);
    if L<=R then
      begin
        Swap:=Ukoly[L];
        Ukoly[L]:=Ukoly[R];
        Ukoly[R]:=Swap;
        inc(L); dec(R);
      end;
  until L >= R;
  if R>Min then SeradDleTerminu(Min,R);
  if L<Max then SeradDleTerminu(L,Max);
end;

procedure PridejDoHaldy(Co:TUkol);
{Přidá úkol do haldy}
var Pozice:integer; {Na jakém místě je (možná) nekonzistence haldy}
    Rodic:integer;  {Úkol, který je v haldě o hladinu výš}
    Swap:TUkol;
begin
   Pozice:=VelikostHaldy;
   inc(VelikostHaldy);
   Halda[Pozice]:=Co;
   while (Pozice > 0) do begin  {Bubláme ke kořeni}
      Rodic:=(Pozice - 1) div 2;
      if (Halda[Rodic].Odmena > Halda[Pozice].Odmena) then break; {Dál se to už nezmění ...}
      Swap:=Halda[Pozice];
      Halda[Pozice]:=Halda[Rodic];
      Halda[Rodic]:=Swap;    {tak jsme vybublali o hladinu výš a všechno můžeme zopakovat}
      Pozice:=Rodic;
   end;
end;

procedure VyradKorenZHaldy();
{Odstraní kořen z haldy ...}
var Pozice:integer; {Kde je (možná) nekonzistence v haldě...}
    Syn:integer; {Syn vrcholu, který právě uvažujeme}
    Swap:TUkol;
begin
   dec(VelikostHaldy);    {Halda se zmenší}
   Halda[0]:=Halda[VelikostHaldy];
```


⋆ Máme predikát:

```
nacti_jmeno(Jmeno) :- write('Zadej jmeno: '),
      read(Jmeno).
```

Co se stane po následující konverzaci:

```
?-nacti_jmeno(Jmeno).
Zadej jmeno: Kleofac
```

1. Nekonečná smyčka
2. Interpret vypíše `Kleofac`
3. Nastane běhová chyba
4. Interpret vypíše nějakou volnou proměnnou
5. Interpret napíše `No`.

Soutěžní úložky

Důležité upozornění: Všechna řešení musí vracet smysluplná řešení i při opětovném volání, např. odmítání středníkem.

1. Lednice (3 body) Myši opět chystají útok na vaši lednici. Myší útok je samozřejmě potřeba dobře naplánovat, a proto myši vyslaly zvěda, který má za úkol zjistit zásoby v lednici. Lednice je zadána jako seznam potravin, které se mohou opakovat. Myši by potřebovaly program, který dostane na vstupu ledničku jako seznam potravin a jednu konkrétní potravinu, a vypíše, kolikrát se daná potravina v lednici vyskytuje.

Příklad:

Pro lednici `[syr, maslo, syr, cibule, syr, syr]` a potravinu `syr` by měl program odpovědět `4`.

2. Myší spartakiáda (3 body) Vážení a vážené, myši, myšáci a myšáčata! Vítejte na myší spartakiádě! Jako první číslo vystupují bílé a černé myši v čísle „Myší obrazec“! Jak vidíte, na cvičišti se nachází `N` soudků. Na každém soudku smí stát právě jedna myš, buď černá nebo bílá. Myši se nyní vystřídají tak, že na `N` soudcích vytvoří všechny možné barevné kombinace černé a bílé.

Dokážete napsat program, který vypíše na obrazovku všechny možné kombinace myší?

Příklad:

Pro 2 soudky jsou možné obrazce: `bb`, `bč`, `čb`, `čč`.

Pro 3 soudky jsou možné obrazce: `bbb`, `bbč`, `bčb`, `bčč`, `čbb`, `čbč`, `ččb`, `ččč`.

3. Myš v bludišti (5 bodů) Myši se rozhodly skoncovat s nejstrašnější noční myší můrou, kterou je dostat se do myšího bludiště. Poprosily vás o program, který by dokázal najít východ z bludiště. Myší bludiště vypadá tak, že máte pokoje očíslované celými čísly `1,2,3,4,...,N` a mezi některými pokoji vede chodba a mezi některými nevede. Technické detaily nechaly myši na vás. Vymyslete vlastní rozumnou reprezentaci bludiště. Myši vám poskytnou plán bludiště ve vaší reprezentaci, startovní pokoj myši a cílový pokoj myši. Váš program by měl najít libovolnou (ne nutně nejkratší) cestu z bludiště nebo odpovědět, že cesta neexistuje.

Hint: Přečtěte si KSP kuchařku o grafech, kterou můžete najít na adrese *http://ksp.mff.cuni.cz/tasks/19/cook3.html*.

4. Oprava (3 body) Najděte chybu v tomto predikátu, popište, proč nefunguje, a opravte jej. Nejdůležitější je podrobný popis a příklad vstupu, na kterém predikát nefunguje.

```
minimum(X,Y,X) :- X =< Y, !.
minimum(X,Y,Y).
```


Recepty z programátorské kuchařky

V tomto dílu programátorské kuchařky si povíme něco o hešování. (V literatuře se také často setkáme s jinými přepisy tohoto anglicko-českého patvaru (hashování), či více či méně zdařilými pokusy se tomuto slovu zcela vyhnout a místo „heš“ používat například termín asociativní pole.) Na heš se můžeme dívat jako na pole, které ale neindexujeme po sobě následujícími přirozenými čísly, ale hodnotami nějakého jiného typu (řetězci, velkými čísly, apod.). Hodnotě, kterou heš indexujeme, budeme říkat *klíč*. K čemu nám takové pole může být dobré?

- Aplikace typu slovník – máme zadán seznam slov a jejich významů a chceme k zadanému slovu rychle najít jeho význam. Vytvoříme si heš, kde klíče budou slova a hodnoty jim přiřazené budou překlady.
- Rozpoznávání klíčových slov (například v překladačích programovacích jazyků) – klíče budou klíčová slova, hodnoty jim přiřazené v tomto příkladě moc význam nemají, stačí nám vědět, zda dané slovo v heši je.
- V nějaké malé části programu si u objektů, se kterými pracujeme, potřebujeme pamatovat nějakou informaci navíc a nechceme kvůli tomu do objektu přidávat nové datové položky (třeba proto, aby nám zbytečně nezabíraly paměť v ostatních částech programu). Klíčem heše budou příslušné objekty.
- Potřebujeme najít v seznamu objekty, které jsou „stejné“ podle nějakého kritéria (například v seznamu osob ty, co se stejně jmenují). Klíčem heše je jméno. Postupně procházíme seznam a pro každou položku zjišťujeme, zda už je v heši uložena nějaká osoba se stejným jménem. Pokud není, aktuální položku přidáme do heše.

Potřebovali bychom tedy umět do heše přidávat nové hodnoty, najít hodnotu pro zadaný klíč a případně také umět z heše nějakou hodnotu smazat.

Samozřejmě používat jako klíč libovolný typ, o kterém nic nevíme (speciálně ani to, co znamená, že dva objekty toho typu jsou stejné), dost dobře nejde. Proto potřebujeme ještě *hešovací funkci* – funkci, která objektu přiřadí nějaké malé přirozené číslo $0 \leq x < K$, kde K je velikost heše (ta by měla odpovídat počtu objektů N, které v ní chceme uchovávat; v praxi bývá rozumné udělat si heš o velikosti zhruba $K = 2N$). Dále popsaný postup funguje pro libovolnou takovou funkci, nicméně aby také fungoval rychle, je potřeba, aby hešovací funkce byla dobře zvolena. K tomu, co to znamená, si něco řekneme níže, prozatím nám bude stačit představa, že tato funkce by měla rozdělovat klíče zhruba rovnoměrně, tedy že pravděpodobnost, že dvěma klíčům přiřadí stejnou hodnotu, by měla být zhruba $1/K$.

Ideální případ by nastal, kdyby se nám podařilo nalézt funkci, která by každým dvěma klíčům přiřazovala různou hodnotu (i to se může podařit, pokud množinu klíčů, které v heši budou, známe dopředu – viz třeba příklad s rozpoznáváním klíčových slov v překladačích). Pak nám stačí použít jednoduché pole velikosti K, jehož prvky budou obsahovat jednak hodnotu klíče, jednak jemu přiřazená data:

```
struct položka_heše
{
  int obsazeno;
  typ_klíče klíč;
  typ_hodnoty hodnota;
} heš[K];
```

A operace naprogramujeme zřejmým způsobem:

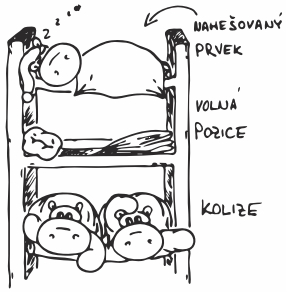
```
void přidej (typ_klíče klíč, typ_hodnoty hodnota)
{
  unsigned index = hešovací_funkce (klíč);

  // Kolize nejsou, čili heš[index].obsazeno=0.
  heš[index].obsazeno = 1;
  heš[index].klíč = klíč;
  heš[index].hodnota = hodnota;
}

int najdi (typ_klíče klíč, typ_hodnoty *hodnota)
{
  unsigned index = hešovací_funkce (klíč);

  // Nic tu není nebo je tu něco jiného.
  if (!heš[index].obsazeno ||
    !stejný(klíč, heš[index].hodnota))
      return 0;

  // Našel jsem.
  *hodnota = heš[index].hodnota;
  return 1;
}
```

Normálně samozřejmě takové štěstí mít nebudeme a vyskytnou se klíče, jimž hešovací funkce přiřadí stejnou hodnotu (říká se, že nastala *kolize*). Co potom?

Jedno z řešení je založit si pro každou hodnotu hešovací funkce seznam, do kterého si uložíme všechny prvky s touto hodnotou. Funkce pro vkládání pak bude v případě kolize přidávat do seznamu, vyhledávací funkce si vždy spočítá hodnotu hešovací funkce a projde celý seznam pro tuto hodnotu. Tomu se říká *hešování se separovanými řetězci.*

Jiná možnost je v případě kolize uložit kolidující hodnotu na první následující volné místo v poli (cyklicky, tj. dojdeme-li ke konci pole, pokračujeme na začátku). Samozřejmě pak musíme i příslušně upravit hledání – snadno si rozmyslíme, že musíme projít všechny položky od pozice, kterou nám poradí hešovací funkce, až po první nepoužitou položku. Tento přístup se obvykle nazývá *hešování se srůstajícími řetězci* (protože seznamy hodnot odpovídající různým hodnotám hešovací funkce se nám mohou spojit). Implementace pak vypadá takto:

```
void přidej (typ_klíče klíč,typ_hodnoty hodnota)
{
  unsigned index = hešovací_funkce (klíč);

  while (heš[index].obsazeno)
  {
    index++;
    if (index == K)
      index = 0;
  }

  heš[index].obsazeno = 1;
  heš[index].klíč = klíč;
  heš[index].hodnota = hodnota;
}

int najdi (typ_klíče klíč, typ_hodnoty *hodnota)
{
  unsigned index = hešovací_funkce (klíč);

  while (heš[index].obsazeno)
  {
    if (stejný (klíč, heš[index].klíč))
    {
      *hodnota = heš[index].hodnota;
      return 1;
    }

    // Něco tu je,ale ne
    // to, co hledám.
    index++;
    if (index == K)
      index = 0;
  }

  // Nic tu není.
  return 0;
}
```



Jaká je časová složitost tohoto postupu? V nejhorším případě bude mít všech N objektů stejnou hodnotu hešovací funkce. Hledání může v nejhorším přeskakovat postupně všechny, čili složitost v nejhorším případě může být až $O(NT + H)$, kde T je čas pro porovnání dvou klíčů a H je čas na spočtení hešovací funkce. Laicky řečeno, pro nalezení jednoho prvku budeme muset projít celý heš (v lineárním čase).

Nicméně tohle se nám obvykle nestane – pokud velikost pole bude dost velká (alespoň dvojnásobek prvků heše) a zvolili jsme dobrou hešovací funkci, pak v průměrném případě bude potřeba udělat pouze konstantně mnoho porovnání, tj. časová složitost hledání i přidávání bude jen $O(T + H)$. A budeme-li schopni prvky hešovat i porovnávat v konstantním čase (což například pro čísla není problém), získáme konstantní časovou složitost obou operací.

Mazání prvků může působit menší problémy (rozmyslete si, proč nelze prostě nastavit u mazaného prvku „obsazeno“ na 0). Pokud to potřebujeme dělat, buď musíme použít separované řetězce (což se může hodit i z jiných důvodů, ale je o trošku pracnější), nebo použijeme následující fígl: když budeme nějaký prvek mazat, najdeme ho a označíme jako smazaný. Nicméně při hledání nějakého jiného prvku se nemůžeme zastavit na tomto smazaném prvku, ale musíme hledat i za ním. Ovšem pokud nějaký prvek přidáváme, můžeme jím smazaný prvek přepsat.

A jakou hešovací funkci tedy použít? To je tak trochu magie a dobré hešovací funkce mají mimo jiné hlubokou souvislost s kryptografií a s generátory pseudonáhodných čísel. Obvykle se dělá to, že se hešovaný objekt rozloží na posloupnost čísel (třeba ASCII kódů písmen v řetězci), tato čísla se nějakou operací „slejí“ dohromady a výsledek se vezme modulo K. Operace na slévání se používají různé, od jednoduchého xoru až třeba po komplikované vzorce typu

```
#define mix(a,b,c) {                          \
  a-=b; a-=c; a^=(c>>13);                     \
  b-=c; b-=a; b^=(a<< 8);                     \
  c-=a; c-=b; c^=((b&0xffffffff)>>13);        \
  a-=b; a-=c; a^=((c&0xffffffff)>>12);        \
  b-=c; b-=a; b =(b ^ (a<<16)) & 0xffffffff; \
  c-=a; c-=b; c =(c ^ (b>> 5)) & 0xffffffff; \
  a-=b; a-=c; a =(a ^ (c>> 3)) & 0xffffffff; \
  b-=c; b-=a; b =(b ^ (a<<10)) & 0xffffffff; \
  c-=a; c-=b; c =(c ^ (b>>15)) & 0xffffffff; \
}
```

My se ale spokojíme s málem a ukážeme si jednoduchý způsob, jak hešovat čísla a řetězce. Pro čísla stačí zvolit za velikost tabulky vhodné prvočíslo a klíč vymodulit tímto prvočíslem. (S hledáním prvočísel si samozřejmě nemusíme dělat starosti, v praxi dobře poslouží tabulka několika prvočísel přímo uvedená v programu.)

Rozumná funkce pro hešování řetězců je třeba:

```
unsigned hash_string (unsigned char *str)
{
  unsigned r = 0;
```


hlavu, dojet na konec seznamu, uškubnout poslední prvek a takhle pokračovat, dokud bychom nezískali prostřední prvek, případně dvojici prvků, v závislosti na tom, zda byl vstupní seznam sudé nebo liché délky. Slepicím se takové řešení moc nelíbilo, možná také proto, že nerady slyší zmínku o škubání.

Zkusme tedy použít trik. Vyšleme v řadě slepic dva signály – jeden pomalý, jeden dvakrát rychlejší. Jakmile dojede rychlý signál na konec, pomalý bude právě uprostřed. V Prologu toto realizujeme tak, že si na vstup dáme tentýž seznam dvakrát a v každém kroku utrhneme v prvním seznamu jenom hlavu, v druhém seznamu první i druhý prvek zároveň.

2. Permutující slepice

Ukázalo se, že tato úloha byla poměrně obtížná. Problém byl hlavně s manipulací se seznamy, většině řešitelů se nepovedlo vytvořit požadovaný seznam permutací.

Jak na to: Ze vstupního seznamu postupně vyberu každý prvek X (tedy jakýsi cyklus for přes všechny prvky seznamu) a vytvořím seznam bez tohoto prvku. Tento seznam nechám rekurzivně zpracovat a dostanu seznam všech možných permutací, jen bez prvku X, načež prvek X předřadím jako hlavu před každou z těchto permutací. Když toto provedu se všemi prvky v zadaném seznamu, vygeneruji všechny permutace.

Jako další argument si musím předávat seznam již vytvořených, hotových permutací, abychom je nezapomněli (neb Prolog nemá globální proměnné) a kdykoliv vytvořím novou permutaci, vložit ji do tohoto seznamu.

Nápad se lehce řekne, ale hůř napíše. Prohlédněte si tedy připojený program.

Nakonec dodáme, že se opravovatelé příliš nešťourali v syntaktických detailech a okrajových případech této úlohy a body se udělovaly i za přibližné řešení.

3. Palindromické slepice

Tato úložka byla opět jednoduchá pro toho, kdo se rozhodl pro jednoduché kvadratické řešení. Snadno vidíme, že stačí vzít vždy první a poslední prvek seznamu, porovnat je a takto pokračovat, až dojedeme doprostřed seznamu.

Rychlejšího, lineárního řešení dosáhneme tak, že seznam otočíme a potom porovnáme původní vstupní seznam s otočeným seznamem. Pokud se shodují, byl vstupní seznam palindromem.

Jak ale otočíme seznam v lineárním čase? To dokážeme pomocí známého triku – použitím tzv. akumulátoru. Princip je velmi jednoduchý – vezmeme si původní seznam, z něj budeme trhat prvky a předřaďovat je jako hlavu do druhého seznamu. Až nám dojdou prvky v původním seznamu, v druhém seznamu (akumulátoru) budeme mít otočený původní seznam.

Jana Kravalová

Úloha 19-2-2 – Kvalitní hesla – program

```
program heslo;

var buf:string;
    max_start1, max_start2, max_len : integer;
    len, i, j , buf_len: integer;
begin
    readln(buf);
    buf_len:= length(buf);

    max_len:=0;
    for i:= 1 to buf_len-1 do begin   //pro každý rozestup začátků
        len:=0;                       //hledam nejdelší shodující se posloupnosti
        for j:= 1 to buf_len-i do begin
            if (buf[i+j]=buf[j]) then begin
                Inc(len);  //pokud se znaky shodují, prodloužíme posloupnost shodných znaků
                if (len>max_len) then begin
                    max_len:=len;
                    max_start1:=j-len+1;
                    max_start2:=max_start1+i;
                end;
            end
            else len:=0;  //jinak počitadla vynuluju
        end;
    end;

    writeln('Řetězce mají délku ',max_len,' a začínají na ',max_start1,' a ',max_start2);
    readln;

end.
```

Úloha 19-2-3 – Moneymaker – program

```
const MaxN = 10000;

type TUkol = record
   Odmena:integer;
   Termin:integer;
   CisloUkolu:integer;
end;

var Ukoly:array[1..MaxN] of TUkol;  {Za co jsou nám ochotni lidi zaplatit ...}
    N:integer; {počet úkolů}
    DalsiUkol:integer;   {První úkol, na který ještě nepřišla řada,
```

Kdo jste tento vzoreček ještě nepotkali, vzpomeňte si na pana Pythagora a jeho větu – chceme změřit přeponu pravoúhlého trojúhelníka S_1TS_2 s pravým úhlem u vrcholu $T = (x_2, y_1)$. Místo vzdáleností budeme ale raději porovnávat jejich druhé mocniny, což jsou pro celočíselné souřadnice bodů také celá čísla. Tak si ušetříme starosti se zaokrouhlovacími chybami a program bude nadále fungovat, jelikož $x < y$ platí právě tehdy, když $x^2 < y^2$, tedy aspoň pro nezáporná čísla, což výraz pod odmocninou bezpochyby je.

Ještě si všimněme jednoho zajímavého faktu: pokud chceme do čtverce velikosti $d \times d$ umisťovat body tak, aby vzdálenost každých dvou byla alespoň d, vejdou se tam maximálně čtyři (třeba do vrcholů čtverce). Dokázat to můžeme například tak, že čtverec rozřežeme na čtyři menší čtverce velikosti $d/2 \times d/2$, které budou mít společné hrany, a nahlédneme, že do každého z nich můžeme umístit nejvýše jeden bod. Nejvzdálenější body v malém čtverci jsou totiž jeho protilehlé vrcholy a ty mají vzdálenost $d\sqrt{2}/2 < d$.

Jak onehdy naznačili jistí programátorští kuchaři, hodit by se mohla metoda Rozděl a panuj. Ta by se pro hledání nejbližší dvojice bodů dala použít zhruba následovně:

- Rozděl všechny body vodorovnou přímkou do dvou stejně velkých množin X_1 a X_2.
- Rekurzivním zavoláním algoritmu najdi minimální vzdálenost d_1 dvojic bodů v X_1 a d_2 v X_2.
- Doplň dvojice sahající přes hraniční přímku: zajímají nás jen takové dvojice, které mohou změnit výsledek, čili jejichž vzdálenost je menší než $d = \min(d_1, d_2)$. Proto stačí uvážit body vzdálené od hraniční přímky méně než d (ostatní body mají moc daleko k hraniční přímce, natož k bodům v druhé množině). Projdeme všechny dvojice takových bodů a označíme d_3 minimum z jejich vzdáleností.
- Vrať jako výsledek $\min(d_1, d_2, d_3)$.

Pokud by první a třetí krok algoritmu běžely v lineárním čase, choval by se celý algoritmus podobně jako QuickSort s rovnoměrným dělením, který jsme ukazovali v kuchařce, a tedy by jeho časová složitost byla $\mathcal{O}(N \log N)$ a paměťová $\mathcal{O}(N)$. Stručně: Na vstup délky N spotřebujeme čas $\mathcal{O}(N)$ plus ho rozložíme na dva vstupy délky $N/2$. Pro ty potřebujeme dohromady také čas $\mathcal{O}(N)$ plus je rozdělíme na čtyři vstupy délky $N/4$, a tak dále, až se po $\log_2 N$ krocích dostaneme ke vstupům délky 1 a celkem tedy spotřebujeme čas $\mathcal{O}(N \log N)$. To je velmi lákavá představa, jen zatím poněkud efemérní, jelikož není vůbec jasné, jak první a třetí krok provést.

Rozdělování bodů: Nabízí se vybrat souřadnici rozdělovací přímky náhodně (podobně jako u QuickSortu bychom se tak dostali na průměrně rovnoměrné rozdělení) nebo si vzpomenout na lineární algoritmus pro výpočet mediánu uvedený v kuchařce. Oba přístupy ale mají společný háček: pokud většina stromů leží na jedné vodorovné přímce, vybereme nejspíš tuto přímku a body rozdělíme nerovnoměrně. Tomu by se dalo odpomoci dělením na tři části – body ležící na dělící přímce bychom zpracovali úplně zvlášť, beztak padnou do pásu, ve kterém dvojice kontrolujeme explicitně.

Mnohem jednodušší je na začátku algoritmu setřídit všechny body podle svislé souřadnice a rozdělit je prostě na prvních $\lfloor N/2 \rfloor$ a zbylých $\lceil N/2 \rceil$. Různé body na dělící přímce sice mohou padnout do různých polovin, ale to není nikterak na škodu, stejně je následně všechny probereme. Třídění nám časovou složitost nepokazí a rozdělování pak dokonce zvládneme v konstantním čase.

Porovnávání hraničních dvojic: Dvojic může být až kvadraticky mnoho (představte si všechny body ležící na dvou vodorovných přímkách), takže je musíme probírat šikovně. Kdybychom je měli setříděné zleva doprava, stačilo by pro každý bod B prozkoumat jen několik bodů od něj doprava – jakmile x-ová vzdálenost překročí d, nemá smysl dál hledat. Zajímají nás tedy body z X_1 ležící ve čtverečku $d \times d$ bezprostředně nad přímkou a body z X_2 ve stejně velkém čtverečku pod přímkou. A my už víme, že v každém z těchto dvou čtverečků mohou ležet nejvýše 4 zajímavé body (každé dva body ležící v téže množině jsou přeci vzdálené aspoň d a použijeme pozorování o umisťování do čtverečků). To je celkem 8 bodů, navíc jedním z nich je náš bod B, čili pro každý bod B zbývá prozkoumat jen 7 následníků. To snadno stihneme v lineárním čase.

Předpokládali jsme ale, že prvky máme setříděné. To skutečně máme, jenže podle druhé souřadnice, než potřebujeme. Jak z toho ven? Jistě můžeme body na počátku setřídit podle každé souřadnice zvlášť a při rozdělování udržovat obě poloviny také setříděné oběma způsoby, ale opět bychom se dostali do potíží s mnoha body na jedné přímce. Proto se uchýlíme k drobnému úskoku: zabudujeme do naší funkce třídění sléváním: funkce na vstupu dostane body setříděné podle y a vrátí je setříděné podle x. To půjde snadno, jelikož z rekurzivních volání dostane každou polovinu správně setříděnou, a tak je jen v lineárním čase slije.

Tím jsme doplnili bílá místa v algoritmu a zbývá ukázat program. Je napsaný v C99 a drží se téměř doslovně našeho algoritmu. Vypisuje pouze nalezenou vzdálenost, ale sami jistě vymyslíte, jak do něj dodělat, aby vypisoval i souřadnice místa, kde nejhlubší je les.

Pár poznámek na závěr:

- Sedmička je trochu přemrštěný odhad: zajímají nás pouze ty dvojice, jejichž vzdálenost je *ostře* menší než d, takže čtverce, ve kterých body mohou ležet, jsou o maličko menší než $d \times d$ a do takových se už vejdou jen tři body (zkuste si dokázat). Správná konstanta je tedy 5.
- Také bychom mohli zkoumat na švu body z X_1 a hledat k nim do páru body z X_2. Pro každý bod z X_1 leží kandidáti z X_2 v obdélníku $2d \times d$ a do něj se vejde nejvýše 6 bodů, což Marek Nečada pěkně dokázal rozřezáním na 6 kousků velikosti $2d/3 \times d/2$ s úhlopříčkou délky $5d/6$.
- Algoritmus, který jsme použili pro zkoumání dvojic ležících na švu, by bylo možné použít i na celou úlohu: body setřídíme podle jedné ze souřadnic a pro každý bod zkoušíme do dvojice jen ty, které jsou v této souřadnici vzdálené maximálně tolik, kolik činí zatím nejmenší nalezená vzdálenost. To může být v nejhorším případě také kvadratické, ale v průměru se dostaneme na $\mathcal{O}(N \cdot \sqrt{N})$. Idea důkazu (podle Zbyňka Konečného): leží-li všechny body v obdélníku $a \times b$ a minimální vzdálenost činí d, nesmí se kruhy o poloměru $d/2$ se středy v zadaných bodech protnout, takže součet jejich obsahů $N\pi d^2/4$ smí být maximálně $(a+2d)(b+2d)$ (kruhy mohou na krajích z obdélníků přečuhovat až o d). Dostaneme kvadratickou nerovnici pro d a z ní po pár úpravách $d = \mathcal{O}(\min(a, b)/\sqrt{N})$.

Martin Mareš

19-2-6 Prolog

1. Příliš těžké slepice Navzdory názvu nebyla úložka příliš těžká, pokud by se ovšem slepice spokojily s kvadratickým řešením. Snad každého napadlo vzít seznam, uškubnout mu


```
  unsigned char c;

  while ((c = *str++) != 0)
    r = r * 67 + c - 113;

  return r;
}
```

Zde můžeme použít vcelku libovolnou velikost tabulky, která nebude dělitelná čísly 67 a 113. Šikovné je vybrat si například mocninu dvojky (což v příštím odstavci oceníme), ta bude s prvočísly 67 a 113 zaručeně nesoudělná. Jen si musíme dávat pozor, abychom nepoužili tak velkou hešovací tabulku, že by 67 umocněno na obvyklou délku řetězce bylo menší než velikost tabulky (čili by hešovací funkce časteji volila začátek heše než konec). Tehdy ale stačí místo našich čísel použít jiná, větší prvočísla.

A co když nestačí pevná velikost heše? Použijeme „nafukovací“ heš. Na začátku si zvolíme nějakou pevnou velikost, sledujeme počet vložených prvků a když se jich zaplní víc než polovina (nebo třeba třetina; menší číslo znamená větší rychlost [méně kolizí], ale větší paměťové plýtvání), vytvoříme nový heš dvojnásobné velikosti (případně zaokrouhlené na vyšší prvočíslo, pokud to naše hešovací funkce vyžaduje) a starý heš do něj prvek po prvku vložíme.

To na první pohled vypadá velice neefektivně, ale protože se po každém nafouknutí heš zvětší na dvojnásobek, musí mezi přehešováním na N prvků a na $2N$ přibýt alespoň N prvků, čili průměrně provádíme jedno přehešování na každý vložený prvek.

Pokud navíc používáme mazání prvků popsané výše (u prvku si pamatujeme, že je smazaný, ale stále zabírá místo v heši), nemůžeme při mazání takového prvku snížit počet prvků v heši, ale na druhou stranu při nafukování můžeme takové prvky opravdu smazat (a konečně je odečíst z počtu obsazených prvků).

Pár poznámek na závěr:

- S hešováním se separovanými řetězci se zachází podobně, nafukování také funguje a navíc je snadno vidět, že po vložení N náhodných prvků bude v každé příhrádce (příhrádky odpovídají hodnotám hešovací funkce) průměrně N/K prvků, čili pro K velké řádově jako N konstantně mnoho. Pro srůstající řetězce to pravda být nemusí (protože jakmile jednou vznikne dlouhý řetězec, nově vložené prvky mají sklony „nalepovat se“ za něj), ale platí, že bude-li heš naplněna nejvýše na polovinu, průměrná délka kolizního řetízku bude omezená nějakou konstantou nezávislou na počtu prvků a velikosti heše. Důkaz si ovšem raději odpustíme, není úplně snadný.
- Bystrý čtenář si jistě všiml, že v případě prvočíselných velikostí heše jsme v důkazu časové složitosti nafukování trochu podváděli – z heše velikosti N přeci přehešováváme do heše velikosti větší než $2N$. Zachrání nás ale věta z teorie čísel, obvykle zvaná Bertrandův postulát, která říká, že mezi čísly t a $2t$ se vždy nachází alespoň jedno prvočíslo. Takže nový heš bude maximálně 4-krát větší, a tedy počet přehešování na jedno vložení bude nadále omezen konstantou.

Vzorová řešení druhé série devatenáctého ročníku KSP

19-2-1 Čokoláda podruhé

Jak si mnoho z vás všimlo, existuje docela jednoduchá vyhrávající strategie pro prvního hráče. V prvním kroku náš začínající bandita rozlomí čokoládu na dvě totožné části (všimněme si, že je to možné pouze tehdy, máme-li alespoň jeden rozměr sudý) a dále už jen opakuje podle osy prvního zlomu soupeřovy tahy do té doby, než vyhraje.

Uvědomme si, že taková strategie určitě vede k vítězství. Pokud jsme hráli podle popsané stragie a prohráli jsme, tj. odlomili jsme kostičku 1×1, udělali jsme to proto, že náš soupeř udělal to samé v minulém tahu – musel tedy prohrát on.

To bylo v případě, že hra skončí. Mohlo by se teoreticky stát, že budeme lámat do nekonečna a nikdo neprohraje. V našem případě se to ale určitě nestane, protože se čokoláda skládá jen z konečného počtu čtverečků.

Cyril Hrubiš

19-2-2 Kvalitní hesla

Jako obvykle jsem chtěl začít své řešení vtipnou poznámkou či glosou, ale vzhledem k tomu, že mi zubařka o Vánocích vyvrtala 4 zuby a já mohl všechny ty cukrovinkami se cpoucí lidi jen sledovat, jistě chápete, že na vtipy nemam náladu. Takže k řešení.

První nápad je vyzkoušet všechny možné podřetězce hesla a vzájemně je porovnat. Bez ohledu na vámi navrhované heuristiky pracuje toto řešení v čase $\mathcal{O}(N^3)$ vzhledem k délce hesla v nejhorším případě s pamětí $\mathcal{O}(N)$, což jste si povětšinou správně uvědomovali a zasloužíte si pochvalu. Pokud jste mi přece jen tvrdili, že je to rychlejší, tak se ještě jednou zamyslete, zda skutečně neexistuje nějaký protipříklad. Pokud by vás přece jen nenapadl (případně já udělal chybu), ozvěte se.

Ono to ale jde o něco (a následně o dost) lépe. Představme si, že máme dobrého kamaráda a ten nám poví, jak daleko od sebe leží začátky shodných podstringů. Pak je ale snadné zjistit, kde se takové podřetězce nachází, prostě jen projdeme celé heslo a nalezneme nejdelší posloupnost shodných dvojic v této vzdálenosti. A to zvládneme jedním průchodem. No a co když nemáme dobré kamarády? Tak prostě prozkoušíme všechny možné vzdálenosti a jen si zapamatujeme, kde jsme dosáhli maxima. Že takových vzdáleností je N a časová složitost $\mathcal{O}(N^2)$ nás tudíž nemine, jistě nemusím dodávat.

Nyní si sedněte, udělejte pohodlí a připravte kyblíčky (na nervy). Začnu slovem *suffixový strom*. Už tím jsem vás jistě odradil, ale pro ty skutečně otrlé dodám ještě odkaz *http://www.dogma.net/markn/articles/suffixt/suffixt.htm* a zároveň velmi poděkuji Kubovi Kaplanovi za tento odkaz a potřebnou inspiraci. Pojednání je bohužel anglicky, takže přidávám ještě jedno české: *http://mj.ucw.cz/vyuka/ga/*, jež je obklopeno ještě několika dalšími zajímavými algoritmy. Po přečtení těchto článečků zjistíte, co to takový suffixový strom je, ke svému údivu objevíte, že se dá zkonstruovat lineárně a s lineární pamětí (no já taky zíral) a ... využijete služeb kyblíčku. Pro ty méně otrlé připojím malý popis. Nejprve vysvětlení: *suffix* je „koncovka“ slova, např. pro slovo `book` získáme suffixy `book`, `ook`, `ok`, `k` a prázdný suffix. Suffixový strom je pak *trie,* ve které jsou uloženy všechny suffixy zadaného řetězce (pokud nevíte co je trie, zapátrejte v ročenkách). Ta má zjevně paměťovou složitost

až $\mathcal{O}(N^2)$ a logicky s tím i čas na její stavbu je $\mathcal{O}(N^2)$. Tím bychom si moc nepomohli, a tak tuto trii vylepšíme. Všechny vrcholy, které mají jen jednoho následníka, sloučíme s jejich otci. V naší trii tak například budou sloučeny vrcholy `b`→`o`→`o`→`k` do jednoho vrcholu `book`. Tím se paměťová složitost zmenší na lineární (uvědomte si, že přidáním suffixu do takto komprimované trie přidáme maximálně dva vrcholy a že všechny řetězce přiřazené k hranám trie jsou podslova zadaného slova, takže si je stačí pamatovat jako polohu začátku a konce v tomto slovu). Technické detaily již ponechám článku a teď k vlastnímu řešení.

Na konec hesla připojím nějaký nikde se nevyskytující znak a nad takto vytvořeným řetězcem vytvořím suffixový strom. Jak se nám projeví společná část suffixů? To je část od kořene stromu k prvnímu místu, kde se tyto suffixy rozdělí. Je velmi důležité si uvědomit, že k tomu musí dojít, protože máme-li dva suffixy, tak se tyto musí lišit v délce a tudíž kratší se oddělí na nějakém znaku od delšího (liší se přinejmenším koncovým znakem hesla, který byl přidán a jistě se uprostřed delšího suffixu nevyskytuje). Tyto společné části suffixů ale hledáme, protože každý podstring je zjevně začátkem (libovolně dlouhým!) alespoň jednoho suffixu. Námi hledaný řetězec se tedy jistě vyskytuje na začátku alespoň dvou suffixů (rozmyslet!).

Příklad: pro heslo `ananas` a suffixy `ananas`, `nanas`, `anas`, `nas`, `as`, `a` mají nejdelší společný začátek `ananas` a `anas` a výsledkem je `ana`. Dále si všimněte, že suffix `anas` krom toho začíná na řetězce `a`, `an`, `ana`, `anas`.

Nejdelší společnou část suffixů pak snadno najdeme jedním průchodem stromu do hloubky, při kterém budeme hledat nejdelší cestu z kořene do vrcholu s alespoň dvěma následníky. To je díky lineární velikosti stromu též lineární, a tak jsou celkové složitosti paměťová i časová $\mathcal{O}(N)$. Ale ptám se vás, stálo to za to? :–)

Jan Bulánek

Poznámka na okraj: Byl bych sice ten poslední, kdo by se ošklíbal nad suffixovými stromy, jenže mi to přece jen nedá, abych neukázal ještě jedno řešení, které má sice o trochu horší časovou složitost, ale vystačí si s daleko jednodušší mašinerií, konkrétně s hešováním z kuchařky v této sérii a přihrádkovým tříděním.

Když vymyslíme algoritmus (budeme mu říkat třeba *pohrabáč,* protože se jím prohrabujeme řetězcem), který v lineárním čase pro dané k zjistí, zda se v zadaném řetězci vyskytuje nějaký podřetězec délky k vícekrát, můžeme použít půlení intervalu na nalezení největšího takového k, přičemž pohrabáč použijeme jen $\log N$-krát.

Lineární pohrabáč vám sice nenabídnu, ale ukážu, jak to udělat alespoň v *průměrně lineárním čase.* Zvolíme si šikovnou hešovací funkci, která bude hešovat k-znakové řetězce do N^2 přihrádek a bude mít navíc tu vlastnost, že pokud jsme ji spočítali pro znaky $a_1 \dots a_k$, dokážeme ji z toho v konstantním čase spočítat pro $a_2 \dots a_{k+1}$ (zkuste si rozmyslet, že funkce `hash_string` z kuchařky toto splňuje). Tak dokážeme zahešovat všechny podřetězce délky k v lineárním čase a víme, že pokud se nějaký vyskytl vícekrát, určitě oba výskyty skončí v jedné přihrádce. Stačí tedy po zahešování projít všechny kolize a zjistit, jestli existovaly dva stejné podřetězce. Navíc platí, že při tomto počtu přihrádek je průměrný počet kolizí $\mathcal{O}(1)$ (to si nedokážeme, ale s pomocí povídání o pravděpodobnosti a středních hodnotách z 16. ročníku to snadno vymyslíte), takže probrání všech kolidujících párů zvládneme v čase $\mathcal{O}(k) = \mathcal{O}(N)$.

Jenže ještě tu je jeden háček: přihrádek jsme zvolili N^2, a tak si nemůžeme dovolit přihrádky reprezentovat polem, protože jenom na jeho projití potřebujeme kvadratický čas. Proto si prostě ke každému začátku podřetězce poznamenáme stranou hodnotu hešovací funkce a začátky podle těchto hodnot setřídíme – to jde přihrádkovým tříděním s N přihrádkami stihnout v lineárním čase, což přesně potřebujeme. Poté porovnáme všechny podřetězce se stejnou hešovací funkcí a zjistíme, zda jsou nějaké dva shodné.

Celkově tedy umíme pohrabovat v čase $\mathcal{O}(N)$ průměrně a požadované k najít v $\mathcal{O}(N \log N)$ a lineární paměti.

Martin Mareš

19-2-3 Moneymaker

Asi nejjednodušší řešení této úlohy by se dalo popsat slovy když metoda *hrr na ně* nezabere, tak se stáhneme a zkusíme to zezadu. Až na jedno řešení využívající intervalové stromy skončili všichni řešitelé začínající od počátku kvadratickou, popř. ještě horší časovou složitostí. Nyní ale zpět k tomu, jak se úloha měla řešit.

Označme T termín nejméně spěchající zakázky. Budeme postupně, pro jednotlivé časy $t < T$, generovat pořadí plnění zakázek (označme je $A_t^t, A_{t+1}^t, \dots, A_T^t$), kterým maximalizujeme zisk v časovém úseku $< t; T >$. Pokud zjistíme jak toto pořadí vypadá pro $t = 1$, tak máme hotovo.

Pro $t = T$ je to jednoduché. Mezi všemi zakázkami s termínem T vybereme tu, která je nejlépe placená. Nyní předpokládejme, že známe optimální pořadí zakázek od času $t+1$ (tj. známe $A_{t+1}^{t+1}, A_{t+2}^{t+1}, \dots, A_T^{t+1}$). Pak tvrdím, že jedna z možných sekvencí zakázek s maximálním ziskem je:

- $A_i^t = A_i^{t+1}$ pro $i \geq t+1$
- A_t^t nalezneme jako zakázku s maximální odměnou, která má termín t, nebo pozdější, a kterou jsme ještě nepoužili (tj. není mezi $\{A_i^{t+1}\}$).

Dokáže se to snadno. Pro spor předpokládejme, že známe nějaké pořadí $B_t^t, B_{t+1}^t, \dots, B_T^t$, které nám zajistí lepší zisk.

Zároveň ale víme, že odměna za úkoly $A_{t+1}^t, A_{t+2}^t, \dots, A_T^t$ je alespoň stejně velká jako za zakázky $B_{t+1}^t, B_{t+2}^t, \dots, B_T^t$ (z toho, že jsme předpokládali, že $\{A_i^{t+1}\}$ maximalizuje zisk na časovém intervalu $< t+1; T >$). Z toho plyne, že odměna za B_t^t je větší než odměna za A_t^t. Jelikož ale A_t^t má maximální odměnu ze všech zakázek, která nebyly obsaženy v $\{A_i^{t+1}\}$, musí tedy existovat $j > t$ takové, že $A_j^{t+1} (= A_j^t) = B_t^t$, nebo jsme dostali spor. Prohodíme tedy v posloupnosti $\{B_i^t\}$ pozice úkolů B_t^t a B_j^t (to si můžeme dovolit, jelikož pak úkol B_j^t splníme dřív a zakázku B_t^t můžeme splnit až v čase j, protože se až tak pozdě vyskytovala v posloupnosti $\{A_i^{t+1}\}$, která termíny respektuje). Tím jsme zřejmě nezměníme celkovou odměnu za úkoly v $\{B_i^t\}$ a tedy celý tento odstavec můžeme použít na novou posloupnost $\{B_i^t\}$ úplně stejně.

To jsme ale ještě nic dokázali, jak si jistě čtenář všiml. Spor dostaneme až když si uvědomíme, že výše uvedené nemůžeme opakovat donekonečna. Pokud budeme uvažovat počet úkolů, které jsou v $\{A_i^t\}$ a $\{B_i^t\}$ na stejném místě (tj. počet takových k, že $A_k^t = B_k^t$), tak v každém cyklu stoupne o 1 (úkol B_t^t se dostane na stejné místo jako je v posloupnosti $\{A_i^t\}$), tedy po několika opakováních výše uvedeného musíme někdy dostat spor.

No a jak toto nejlépe implementovat? Nejdřív setřídíme úkoly dle termínu. Pak budeme odzadu generovat jednotlivé úkoly, které je třeba v daný čas t udělat. K tomu použijeme haldu. Budeme si v ní udržovat úkoly, které mají termín t či pozdější a které jsme zatím ještě nezařadili mezi zakázky, které splníme. Na začátku bude prázdná a v každém kroku do haldy přidáme všechny úkoly, které mají termín t (pozdější tam již máme z předchozích kroků) a odebereme maximum. Tím jsme skoro hotovi.

Kdybychom implementovali výše uvedené doslovně, tak čas běhu programu bude kromě velikosti vstupu záviset i na nejpozdějším termínu úkolu. Toho se ale je možno jednoduše zbavit. Pokud bude halda prázdná, tak můžeme rovnou posunout čas na nejbližší dřívější termín zakázky, čímž si ušetříme čas.

No a složitost. Setřízení pomocí rychlého třídícího algoritmu trvá $\mathcal{O}(N \cdot \log N)$. Přidání do haldy zabere $\mathcal{O}(\log N)$ a provádíme ho N-krát, tedy opět $\mathcal{O}(N \cdot \log N)$. No a ještě z haldy odebíráme kořen. To uděláme také maximálně N-krát a trvá to $\mathcal{O}(\log N)$. Dohromady tedy $\mathcal{O}(N \cdot \log N)$.

V paměti máme vstup a haldu. Jejich velikost je přímo úměrná velikosti vstupu, tedy paměťová složitost je $\mathcal{O}(N)$.

Pavel Čížek

19-2-4 Optimální formace

Tato úloha, na první pohled docela snadná – stačilo najít souřadnice vrcholů *jakéhokoli* konvexního mnohoúhelníka, byla nakonec docela těžká. První, byť jen malý zádrhel, byl v tom, uvědomit si, že každý střelec musí svůj dostřel využít naplno. Pokud bychom totiž u střelce S_i využili jen část dostřelu tak, aby všechny vnitřní úhly byly ostře menší než 180°, mohli bychom útvar malilinko „zplacatit“ a dostat tak ještě o kousek větší obvod. Takhle bychom se buď u každého dostali na maximální dostřel, nebo maximum neexistuje.

Kdy jde nakreslit konvexní mnohoúhelník o zadaných stranách? Pokud splňuje zobecnění trojúhelníkové nerovnosti – mnohoúhelníkovou nerovnost, která říká, že pro každou hranu S_i musí součet délek ostatních hran být větší než délka S_i. Tuto nerovnost stačí ověřit pro S_i nejdelší stranu. Nejdelší stranu budu dále označovat S_0, ostatní pak S_1, S_2, S_{n-1} v pořadí v jakém navazovaly na S_0. Vrcholy označím $V_0, \dots V_{n-1}$, přitom $S_i = (V_i, V_{i+1})$ a $S_{n-1} = (V_{n-1}, V_0)$.

Jak teď sestrojit konkrétní souřadnice? Mohli bychom si mnohoúhelník představit jako trojúhelník $V_0 V_1 V_l$, kde l je co nejblíže polovině vzdálenosti $V_1 - V_0$ mimo S_0. Takový trojúhelník splní trojúhelníkovou nerovnost a celkem snadno můžeme spočíst souřadnice jeho vrcholů. Teď už stačí body na přímkách $V_1 - V_l$ a $V_l - V_0$ lehce „vyboulit“ abychom dosáhli úhlů menších než 180° a přitom si nepokazili sestrojitelnost ani konvexnost. Jak na to?

Jedna z možností je umístit strany $S_1, S_2, \dots S_{n-1}$ jako tětivy po obvodu *dostatečně velké* kružnice k a tu pak ve vhodném bodě V_l „zlomit“ a přiblížit tím body V_0 a V_1 na délku úsečky S_0. Jak velkou kružnici k zvolit? Stačí, když po tom, co na ní umístíme S_0 jako tětivu, délka oblouku $V_0 V_1$ bude menší nebo rovna obvodu mnohoúhelníka bez nejdelší strany. Nyní se dostává ke slovu analytická geometrie černější než černá magie a proto nebudu všechny kroky zdůvodňovat a důkazy korektnosti jednotlivých voleb přenechám laskavému čtenáři.

Obvod mnohoúhelníka bez nejdelší strany označím S_u Poloměr kružnice k zvolím

$$r \geq \frac{S_u S_0}{\sqrt{S_u^2 S_0^2}}.$$

Střed této kružnice umístím na souřadnici $[r, 0]$, od souřadnice $[0, 0]$ začnu směrem nahoru po kružnici umisťovat body $V_1, V_2, \dots V_{n-1}, V_0$ ve správných vzdálenostech na souřadnice $[x_i, y_i]$. Jak spočítat souřadnice těchto bodů? Další bod $V_{i+1} = [x_{i+1}, y_{i+1}]$ musí být ve správné vzdálenosti od bodu $V_i = [x_i, y_i]$ i středu kružnice k $[0, r]$, musí tedy platit

$$(x_{i+1} - r)^2 + y_{i+1}^2 = r^2$$
$$(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2 = S_i^2$$

Vyřeším kvadratickou rovnici, ze které dostanu dvě možnosti pro V_{i+1}. Kružnici jsem si zvolil dvakrát tak velkou, než bych nutně potřeboval a proto mohu spoléhat na to, že pokládám vždy směrem nahoru. Označím-li si pro jednotlivá i

$$A_{i+1} = 1 + \frac{(r - x_i)^2}{y_i^2}$$
$$B_{i+1} = \frac{(r - x_i)(x_i^2 + y_i^2 + S_i^2)}{y_i^2} - 2r$$
$$C_{i+1} = \frac{(x_i^2 + y_i^2 + S_i^2)^2}{4y_i^2},$$

dostanu $x_{i+1} = (B_{i+1} + \sqrt{B_{i+1}^2 - 4A_{i+1}C_{i+1}})/2A_{i+1}$ a

$$y_{i+1} = \frac{x_i^2 + y_i^2 + S_i^2 + 2x_{i+1}(r - x_i)}{2y_i}.$$

Bod V_l zvolím co nejblíže polovině vzniklého oblouku $V_1 V_0$. Je třeba dopočíst finální souřadnici bodu V_0 tak, aby byl od V_1 vzdálen S_0 a od V_0 vzdálen F. Toto opět vede na soustavu kvadratických rovnic, z jejichž dvou řešení vybereme to s větší souřadnicí y. Výpočet souřadnic $[x_0, y_0]$ je jen rutina a přenechám ji odvážnému čtenáři, který se dočetl až sem.

Předposlední částí výpočtu je otočit body $V_{l+1}, V_{l+2}, \dots, V_{n-1}$ okolo bodu V_l o stejný úhel, o jaký byl otočen bod V_0 (ten se pohyboval na kružnici se středem v V_l). To je možné třeba spočtením tohoto úhlu goniometrickými funkcemi a sestavením transformační matice nebo řešením dalších soustav kvadratických rovnic.

Poslední částí je pak jednoduché vypsání výsledků, které je po tom všem už hračkou.

Časová i paměťová složitost tohoto algoritmu je lineární. Paměťovou složitost by bylo možno snížit na konstantní, pokud bychom některé hodnoty zapomínali a v průběhu výpočtu si je znovu dopočítali. Program neuvádíme, protože se skládá pouze z výpočtů pomocí zde navržených vzorců.

Tomáš Gavenčiak

19-2-5 Hluboký les

Je zajisté triviální nalézt les nejhlubší zkoumáním vzdáleností všech dvojic stromů, ale uznejte sami, že za to bychom sotva slibovali 13 bodů, protože je to cca desetiřádkový program s ošklivou kvadratickou složitostí. Zkrátka to, čemu se říkává dřevorubecké řešení. Pojďme se raději zakoukat do hladiny křišťálové studánky, jestli nám neporadí, jak na to jít lépe (třeba od lesa):

Stromy si představme jako body v rovině, x-ová souřadnice bude odpovídat směru zleva doprava, y-ová shora dolů. Vzdálenost stromů $S_1 = (x_1, y_1)$ a $S_2 = (x_2, y_2)$ bude činit:

$$d(S_1, S_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.$$