Výsledková listina šestnáctého ročníku KSP po první sérii

		škola	ročník	1611	1612	1613	1614	1615	suma	celkem
1. – 2.	Jan Bulánek	G Klatovy	3	10	10	11	9	10	50	50
	Peter Perešíni	GJGTajov	2	10	10	11	10	9	50	50
3.	Marek Jančuška	G Nitra	4	9	10	11	9	10	49	49
4.	Miroslav Cicko	GJGTajov	3	10	10	11	8	9	48	48
5.	Petr Škoda	GÚstavní	4	10	10	11	6	9	46	46
6.	Pavel Motloch	GPBezruče	1	9	10	11	5	10	45	45
7.	Tomáš Gavenčiak	G Bílovec	4	10	10	11	2	10	43	43
8.	Zbyněk Falt	GNeumannov	3	10		11	10	9	40	40
9.	Jana Kravalová	G VKlobou	4	5	10	11	4	9	39	39
10.	Martin Křivánek	GKptJaroš	2	4	10	11	3	9	37	37
11.	Michal Repovský		4	10	9	11	5		35	35
12.	Peter Černo	GĽŠtúra	3	9	10	11	4		34	34
13.	Jindřich Flídr	G Lanškr	4	4	9	11		9	33	33
14.	Kryštof Hoder	GKptJaroš	4	10	10	11			31	31
15. – 17.	Jan Hrnčíř	GFXŠaldy	2	4	3	10	4	9	30	30
	Miroslav Klimoš	G Lanškr	0	4	9	5	3	9	30	30
	Marek Ludha	GJGTajov	4	4	9	11	4	2	30	30
18. – 21.	Ondřej Bílka	G Zlín	2	7	9	4		9	29	29
	Petr Kortánek	G Sedlča	2	4	4	10	3	8	29	29
	David Matoušek	GZborov	4		9	11		9	29	29
	Martin Podloucký	G Strážnic	3	7	10		3	9	29	29
22.	Martin Čech	G UBrod	3	3	10	2	4	9	28	28
23.	Michal Bečka	G MTřebová	4		10		8	9	27	27
24. – 26.	Ondřej Garncarz	G Příbor	3	4	5	11	4	2	26	26
	Jaroslav Havlín	G Sedlča	4	5		11	10		26	26
	Filip Šauer	G Klatovy	3	4	10		3	9	26	26
27. – 28.	Stanislav Basovník	G Kroměříž	3	5	9			10	24	24
	Jan Křetínský	GMLercha	4		9		5	10	24	24
29. – 30.	Stanislav Haviar	G Klatovy	3			11	10	2	23	23
	Peter Šufliarsky	G NZámky	4	5	4	10	4		23	23
31. – 33.	Martin Koníček	G UBrod	3	5		11	4	2	22	22
	Daniel Marek	GZborov	1	7	9	6			22	22
	Ján Záhornadský	GZborov	3	4	10		3	5	22	22
34.	Martina Tomisová	GZborov	4	7	2		3	9	21	21
35. – 36.	Petr Švec	G Beroun	4	5	4	5	4	2	20	20
	Benjamin Vejnar	G Nymburk	4		10			10	20	20
37.	Jana Fabriková	GKptJaroš	4	9		0	1	9	19	19
38.	Petr Soběslavský	GJHeyrovs	3	4	4	3	5	2	18	18
39.	Pavel Klavík	G Chrudim	1	4	3	5	3	2	17	17
40.	Milan Dvořák	G NMnMor	1	3	4			9	16	16
41.	Adam Přenosil	GSladkNám	2	4	9			2	15	15
42.	Cyril Hrubiš	G Bílovec	2	3	9			2	14	14
43.	Jiří Bělohradský		2	4	9				13	13
44. – 45.	Martin Kupec	GMendel	2	1	3	4	3		11	11
	Jan Richter	G Příbor	3	3	3		3	2	11	11
46. – 47.	Marek Blahuš	G UHradi	3					10	10	10
	Michal Potfaj	G NMnVáh	4	2	4		2	2	10	10
48.	Martin Schmid	G ČTřebová	0	2	4		3		9	9
49. – 52.	David Irschik	G Ledeč	3	4	3				7	7
	Petr Paščenko	GDašická	4	4			3		7	7
	Eva Schlosáriková	G Piešťany	3		4		3		7	7
	Jaromír Vojíř	G Ledeč	3	3	3		1		7	7
53.	Petr Musil	G MBuděj	2		4			2	6	6
54. – 55.	Tomáš Herceg		1	4					4	4
	Radoslav Sopoliga	G Svidník	4		3		1		4	4
56. – 57.	Petr Kratochvíl		1	2					2	2
	Aleš Razým		3				2		2	2

Milí řešitelé!

zima a s ní i Vánoce se pomalu blíží, a tak se svou nadílkou pod stromeček přichází i KSP. Doufáme, že se vám naše dárky budou líbit a že s nimi strávíte příjemné chvíle.

Aktuální informace o KSP můžete nalézt na Internetu na *http://ksp.mff.cuni.cz/,* dotazy organizárotům (nikoliv vyřešené úlohy!) je možno posílat E-mailem na adresu *ksp@mff.cuni.cz.*

Zadání druhé série šestnáctého ročníku KSP

16-2-1 Král Eeek — 10 bodů

Byl-nebyl jednou jeden král, který se jmenoval Eeek a měl tuze rád knížky. Jeho palác se už dávno stal jednou velikánskou knihovnou a král Eeek trávil celé dny vysedáváním u krbu ve své soukromé čítárně. Až jednoho dne za ním přišel lord nejvyšší knihovník a svěřil se králi, že objevil knihu, které ani za mák nerozumí.

Celý text onoho masivního fasciklu byl tvořen jen a pouze číslicemi, uspořádanými zdánlivě bez jakéhokoliv řádu, a spolehlivě činil zmatek jak v pomazané hlavě královské, tak v poněkud méně vznešené, leč vzdělanější hlavě knihovníkově. Dlouho o tom při dobrém víně přemýšleli, až dospěli k následující teorii:

Kdysi dávno žila civilizace, jejíž filosofové rádi zapisovali své myšlenky jako dlouhatánská čísla v roztodivných číselných soustavách. Ovšem základy těchto soustav byly natolik velké, že jim brzy došly symboly pro číslice, a tak jednotlivé číslice začali zapisovat v desítkové soustavě. Navíc ještě za číslo připojovali základ soustavy, rovněž zapsaný desítkově. Čísla tedy vypadala například takto:

$$(1)(6)[10] = 1 \cdot 10^1 + 6 \cdot 10^0 = 16$$
$$(10)(10)[11] = 10 \cdot 11^1 + 10 \cdot 11^0 = 120$$
$$(14)(10)(5)[16] = 14 \cdot 16^2 + 10 \cdot 16^1 + 5 \cdot 16^0 = 3749$$
$$(1)(4)(1)(0)(5)(1)[6] = 1 \cdot 6^5 + \ldots + 1 \cdot 6^0 = 13207$$

Přitom zápisy číslic nikdy nezačínaly nulou, pokud nešlo o nulu samotnou, a byla to vždy celá nezáporná čísla menší než základ. Základ byl celé číslo větší než 1 a také nikdy nezačínal nulou. Formálně řečeno, hodnota čísla se stanovovala podle následujícího pravidla:

$$(a_n)(a_{n-1})\ldots(a_1)(a_0)[z] = \sum_{i=0}^{n} a_i \cdot z^i.$$

„Léty ovšem závorky v zápisu vybledly a zbyly jen číslice, takže původní číslo už stěží kdo rozpozná," povzdechl si král Eeek. Knihovník na to ale opáčil, že zatímco se Jeho veličenstvo ráčilo věnovati hodnotné literatuře, poddaní mezitím objevili počítače a dokonce si už založili i programátorský korespondenční seminář, takže jistě dokáží napsat program, který o daném řetězci čísel rozhodne, kolik existuje způsobů, jak doplnit závorky tak, aby vznikl korektní zápis nějakého čísla.

Co říkáte, dokážete to?

Příklad: Posloupnost `1410516` odpovídá zápisům:

(1)(4)(1)(0)(5)(1)[6]
(1)(4)(1)(0)(5)[16]
(14)(1)(0)(5)[16]
(1)(4)(10)(5)[16]
(14)(10)(5)[16]
(1)(4)[10516]
(14)[10516]
(1)[410516]
(1)(4)(1)(0)[516]
(14)(1)(0)[516]
(1)(41)(0)[516]
(141)(0)[516]
(1)(4)(10)[516]
(14)(10)[516]
(1)(410)[516]

Naproti tomu posloupnost `100` žádnému zápisu neodpovídá.

16-2-2 Král Ovopole — 10 bodů

Nebyl-byl jednou jiný král, kterému říkali Ovopole, neboť měl zvláštní zálibu ve vajíčkách – mimo to, že si na nich rád pochutnával, je také vědecky zkoumal. Jednoho dne ho napadlo, že zjistí, ze kterého nejnižšího patra jeho vysokánského (takto N-patrového) paláce se vajíčko puštěné z okna na nádvoří rozbije.

To je samozřejmě snadné zjistit pokusy: V každém pokusu král vajíčko pustí z nějakého patra a když se vajíčko nerozbije, nechá si ho přinést a může s ním podniknout další pokus; pokud se rozbije, musí král sáhnout po dalším vajíčku. To je snadné, ale ouha, Ovopole právě s ú(div|děs|zas)em zjistil, že v celém paláci se nachází jen k vajíček, která doposud nepodlehla předchozím experimentům. Navíc král je poněkud netrpělivý, takže by s těmito dvěma vajíčky chtěl získat správnou odpověď na co nejméně pokusů.

Napište proto našemu králi program, který jeho problém vyřeší (jistě se vám za to dostane královské odměny). Takový program dostane na vstupu počet pater N a počet vajíček k a postupně bude navrhovat jednotlivé pokusy a přijímat odpovědi, jak právě vypsaný pokus dopadl. Nakonec program odpoví číslem hledaného nejnižšího patra.

16-2-3 Král Potvorník — 10 bodů

Král Potvorník (matematiky většinou zvaný „Ten, který zadává ε" nebo prostě Nepřítel) si u svých zeměměřičů objednal přeměření své královské potvorologické zahrady za účelem stavby nového plotu. Zahrada má, jak je známo, tvar nepravidelného konvexního n-úhelníku (to, že je konvexní, znamená, že všechny vnitřní úhly jsou menší než 180°) a Potvorník potřebuje zjistit její obvod, aby věděl, kolik pletiva a ostnatého drátu musí objednat.

Zeměměřiči vyměřili souřadnice všech sloupků budoucího plotu ležících přesně ve vrcholech n-úhelníka a když se užuž chystali dát se do počítání, přiběhla jedna z obyvatelek zahrady a jako na potvoru do hromádky lístků s napsanými souřadnicemi strčila a beznadějně je pomíchala. Nedosti na tom, zamíchala mezi ně i jiné lístky, na nichž byly souřadnice různých objektů ležících uvnitř zahrady.

Na vás je, abyste zeměměřiče zachránili před královskou odměnou (která by je jistě neminula, kdyby nespočítali včas) tím, že napíšete program, který dostane na vstupu všechny nasbírané souřadnice (tedy souřadnice vrcholů a nějakých bodů uvnitř, to vše v libovolně zpotvořeném pořadí) a odpoví co možná nejrychleji, jaký je obvod zahrady (jak již asi tušíte, ani tento král není zrovna vzorem trpělivosti).

16-2-4 Křížový král — 10 bodů

Při putování křížem kráž světem ve službách Křížového (nebo Krážového?) krále jste se dostali až do jeskyně obývané ohnivým drakem. Drak vás vřele přivítal a rovnou vás pozval k obědu. Brzy jste bohužel zjistili, že se také můžete stát jeho podstatnou součástí. Nicméně draci jsou čestní

a navíc se tento už dlouho nudil, takže jste dostali šanci zachránit si život. Stačí porazit draka ve hře.

Hra je velmi jednoduchá: drak si zvolí nějaké přirozené číslo mezi 1 a N (kteréžto N je předem známo) a na vás je, abyste ho uhodli. Vy si v každém tahu zvolíte množinu čísel a drak vám řekne, zda se v ní jeho číslo nachází či nikoliv. Když jste si jistí, řeknete číslo, o kterém si myslíte, že si ho drak myslí, a drak zřejmým postupem rozhodne, zda si pochutnáte na pečínce nebo skončíte na pekáči.

Nicméně drak je velmi starý a buď se definice čestnosti od dob jeho mládí trochu změnila, nebo začíná být poněkud sklerotický. Drak považuje za zcela čestné podvádět, pokud to ovšem neudělá častěji než jednou za hru. Čili ve vaší hře jedna z jeho odpovědí může (ale také nemusí) být chybná.

Pokud ovšem s odpovědí příliš otálíte a ptáte se příliš dlouho, drak začne být z nutnosti provádět komplikované operace s množinami hladový a samozřejmě v takové situaci netoužíte po tom, aby mu došla trpělivost.

Je tu ještě jeden drobný háček – v dračí řeči si nejste zrovna nejjistější a drak sice češtinu ovládá bez problémů, nicméně nedávno si nechal nabrousit zuby a jeho výslovnost od té doby není příliš dobrá. Abyste předešli nedorozumněním, dohodli jste se, že budete komunikovat v jazycích zvaných Pascal nebo C a místo toho, abyste hráli přímo, popíšete svou strategii jako program.

P.S.: Zaručená informace pocházející od předkrmu říká, že při $N = 1\,000\,000$ drakovi trpělivost vydrží alespoň 26 kol.

16-2-5 Král Popleta — 10 bodů

Panovník Popleta XI. (nebo že by už XII., kdo ví?) jistého nejmenovaného království se již chystá na odpočinek (moderně bychom řekli do důchodu) a je potřeba, aby své království přenechal svým potomkům. A to je právě ten problém. Král má dva syny, Petra a Pavla, a shodou okolností jsou to dvojčata. Jak tak léta běžela, dvořané už dávno zapomněli, který ze synů je vlastně ten prvorozený, a bude proto potřeba království mezi ně spravedlivě rozdělit.

Království je tvořeno n provinciemi, které jsou očíslovány čísly od 1 do n. Každá připadne právě jednomu ze synů. Synové sepsali své požadavky na to, jak by takové rozdělení mělo vypadat. Petrovy podmínky jsou následujícího tvaru:

- Dostanu provincii s číslem i.
- Dostane-li provincii s číslem i druhý syn, pak já dostanu provincii s číslem j.
- Dostanu provincii s číslem i nebo s číslem j (nebo ještě lépe obě).

Stejné tři typy podmínek (až na to, že provincie chce pro sebe) má i Pavel.

Popletovi XI. jde z toho hlava kolem, a tak povolal své moudré rádce, aby království rozdělili. Rádcové podmínky synů podrobně zkoumali a brzy zjistili, že neexistuje žádné rozdělení provincií, které by všechny sepsané podmínky splnilo naráz. Na druhou stranu také zjistili, že libovolné tři podmínky naráz splnit lze, tj., např. se mezi podmínkami nevyskytuje následující trojice:

- Petr požaduje pro sebe provincii s číslem 2.
- Petr požaduje pro sebe provincii s číslem 4.
- Pavel požaduje pro sebe provincii s číslem 2 nebo provincii s číslem 4.

I přes tato objevná pozorování, nakonec rádci králi Popletovi XI. poradili, ať si u každé provincie hodí korunou a podle výsledku hodu pak provincii přenechá buď Petrovi nebo Pavlovi. Že prý takto (v průměrném případě), splní alespoň polovinu všech podmínek, které si oba synové dohromady vymysleli. Všimněte si, že podmínky synů by byly splněny s pravděpodobností 1/2 a 3/4 (dle jejich typu) a tedy v průměrném případě je splněna alespoň polovina všech podmínek. Popletovi XI. se to však nějak nezdálo (přece nebude házet svou zlatou královskou korunou, co nosí na hlavě, to dá rozum) a obrátil se na vás.

Vaším úkolem je vymyslet pravděpodobnostní algoritmus, který v průměrném případě splní co největší množství podmínek na rozdělení království. Tedy, nalezněte co největší číslo α, $0 \le \alpha \le 1$, a k němu příslušný pravděpodobnostní algoritmus takový, že střední hodnota počtu splněných podmínek na rozdělení království je αm, tj. rozdělení provincií nalezené algoritmem průměrně splní alespoň αm podmínek ze všech m podmínek. Váš algoritmus by měl pracovat v polynomiálním čase (král chce předat království svým synům ještě za svého života).

Recepty z programátorské kuchařky

Jestli programátor ve své praxi bude dělat něco velmi často, tak to bude dozajista třídění nejrůznějších dat. Co to znamená? Pojem *třídění* je možná maličko nepřesný, nehodláme data (čísla, záznamy, řetězce a jiné) rozdělovat do nějakých tříd, ale chceme stanovit jejich správné pořadí. Se setříděnými údaji se mnohem lépe pracuje, usnadníme si zejména pozdější vyhledávání uložených dat. Třídící algoritmy jsou jedny z nejstudovanějších algoritmů, my však nebudeme do nějakých velkých detailů a specialit příliš zabíhat. Zkrátka a dobře – budeme chtít třídit údaje rychle, úsporně a radostně.

V programech obvykle třídíme jednotlivé exempláře nějaké datové struktury typu pascalského záznamu. V takové struktuře bývá obsažena jedna význačná položka označovaná většinou jako *klíč*, podle které se záznamy řadí. Abychom si zjednodušili výklad, budeme nadále předpokládat, že třídíme záznamy obsahující pouze klíč, a to dejme tomu celočíselný. Čísla budeme chtít seřadit od nejmenšího k největšímu. Pomocí počtu tříděných čísel N budeme vyjadřovat časovou složitost jednotlivých algoritmů. Metody třídění můžeme rozdělit do dvou hlavních skupin, a to na tzv. *vnitřní třídění,* kdy si můžeme dovolit všechna data načíst do paměti počítače, a *vnější třídění,* kdy již třídění musíme realizovat opakovaným čtením a vytvářením diskových souborů. My se omezíme pouze na algoritmy vnitřního třídění a tříděné pole si nadeklarujeme takto:

```
const N = 20;
type Pole = array[1..N] of integer;
```

Nejjednodušší třídící algoritmy patří do skupiny *přímých metod.* Všechny mají několik společných rysů: jsou krátké a jednoduché a třídí se na místě v poli. Tyto algoritmy mají časovou složitost $\mathcal{O}(N^2)$. Z toho vyplývá, že jsou použitelné jen když tříděných dat není příliš mnoho. Stručně si přiblížíme tři nejznámější přímé metody:

Třídění přímým výběrem (SelectSort) spočívá v opakovaném vybírání nejmenšího čísla m z dosud nesetříděných čísel. Nalezené číslo m si šikovně prohodíme se začátkem pole a postup opakujeme, tentokrát na indexech $2, \ldots, N$. Nalezený nejmenší prvek se tak dostane těsně za číslo m. Je snadné si uvědomit, že když takto postupně vybíráme minimum z menších a menších intervalů, setřídíme celé pole.


```
            begin
              if (nosnosti[w][i]<ohodnoceni[w]) then
                ohodnoceni[i]:=nosnosti[w][i]
              else
                ohodnoceni[i]:=ohodnoceni[w];
              odkud[i]:=w;
            end
        end
  until trvaly[N] or (w=0);
  if trvaly[N] then vypis(N) else write('Cesta z dolu do továrny neexistuje.');
  writeln;
end.
```

Úloha 16-1-4 – Neruš mé trojúhelníky! – Program

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

#define MAX 1024

float p[MAX];
int n;

int cmp (const void *a, const void *b)
{
        return *(float *)a - *(float *)b;
}

int test (int i, int j)
{
        int h = (j+1)/n;
        if (p[i] + 180 > p[(j+1) % n] + 360 * h)
                return 1;
        return 0;
}

int main (void)
{
        int i, k, sum;

        scanf ("%d", &n);
        for (i=0; i<n; i++){
                scanf ("%f", &p[i]);
        }

        qsort (p, n, sizeof (float), cmp);

        k = 0;
        while (test (0, k)) k++;

        sum = (k - 1) * k / 2;
        for (i=1; i<n; i++){
                while (test (i, k)) k++;
                sum += (k - i) * (k - i - 1) / 2;
        }

        printf ("%d\n", n*(n-1)*(n-2)/6-sum);

        return 0;
}
```

```
      r := m;
    end
  else
    l := m+1;
end;
WriteLn('Minimum je na pozici ', (l+r) div 2, '.');
end.
```

A ještě lahůdka pro milovníky jazyka C:
for (l=0, r=N+1, m= (l+r)/2; l!=r; m= (l+r)/2, $A[m]$<$A[m+1]$? ($A[m-1]$<$A[m]$?r=m−1:l=r=m): (l=m+1));

Úloha 16-1-3 – Důl – Program

```
program dul;
const MAX=100;
var N: word; { počet měst }
    nosnosti: array[1..MAX,1..MAX] of integer;     { nosnosti silnic, -1 = silnice neexistuje }
    ohodnoceni: array[1..MAX] of integer;          { nosnosti zatím nalezených cest, -1 = nekonečno }
    odkud: array[1..MAX] of integer;               { předchozí město na nejlepší cestě }
    trvaly: array[1..MAX] of boolean;              { trvale ohodnocen? }
    i,j,w:word;
    max_nosnost:integer;
procedure vypis(konec: word);
begin
  if odkud[konec]<>-1 then
    vypis(odkud[konec])
  else
    write('Nalezená cesta:');
  write(' ',konec);
end;
begin
  { Nejdříve načteme popis silniční situace Kaputánii }
  readln(N);
  max_nosnost:=0;
  for i:=1 to N do
    for j:=i+1 to N do
      begin
        readln(nosnosti[i][j]);
        nosnosti[j][i]:=nosnosti[i][j];
        if max_nosnost<nosnosti[i][j] then max_nosnost:=nosnosti[i][j];
      end;
  for i:=1 to N do
    begin
      trvaly[i]:=false;
      ohodnoceni[i]:=-1;
    end;
  odkud[1]:=-1;
  ohodnoceni[1]:=max_nosnost;
  repeat
    w:=0;
    for i:=1 to N do
      if not(trvaly[i]) then
        if w=0 then
          w:=i
        else
          if ohodnoceni[i]>ohodnoceni[w] then
            w:=i;
    if w<>0 then
      begin
        trvaly[w]:=true;
        for i:=1 to N do
          if (nosnosti[w][i]<>-1) and not(trvaly[i]) and
               { podmínku "not(trvaly[i])" lze vypustit }
             (ohodnoceni[i]<ohodnoceni[w]) and
               { podmínka ohodnoceni[i]<ohodnoceni[w] je vždy splněna !}
             (ohodnoceni[i]<nosnosti[w][i]) then
```



```
procedure SelectSort(var A: Pole);
var i,j,k,x: integer;
begin
  for i:=1 to N-1 do
  begin
    k:=i;
    for j:=i+1 to N do
      if A[j] < A[k] then k:=j;
    if k > i then
      begin x:=A[k]; A[k]:=A[i]; A[i]:=x end
  end
end;
```

Třídění přímým vkládáním (InsertSort) funguje na podobném principu, vlevo na začátku pole si střádáme již správně utříděnou posloupnost. Čísla z pravého úseku se berou jedno po druhém a do levého úseku se vkládáním zatřiďují podle velikosti, kam patří.

```
procedure InsertSort(var A: Pole);
var i,j,x: integer;
begin
  for i:=2 to N do
  begin
    x:=A[i];
    j:=i-1;
    while (j>0) and (x < A[j]) do
    begin
      A[j+1]:=A[j];
      j:=j-1;
    end;
    A[j+1]:=x;
  end
end;
```

(Upozornění: ve našich příkladech předpokládáme, že máme v překladači zapnuto zkrácené vyhodnocování logických výrazu, třeba v předchozím while cyklu se při `j=0` již s prvkem `A[0]` neporovnává. Zdrojáky pro úplné vyhodnocování si jistě každý dokáže sám upravit.)

Bublinkové třídění (BubbleSort) pracuje maličko jinak. Postupně se systematicky porovnávají dvojice sousedních prvků a vyměňují se spolu vždy, když menší číslo následuje po větším. Na konci výpočtu jsou všechny dvojice sousedních prvků uspořádané, pole je tedy setříděné. Algoritmu se říká „bublinkový“ proto, že podobně jako bublinky v limonádě stoupají vysoká čísla v poli vzhůru.

```
procedure BubbleSort(var A: Pole);
var i,j,x: integer;
begin
  for i:=2 to N do
    for j:=N downto i do
      if A[j-1] > A[j] then
        begin
          x:=A[j-1]; A[j-1]:=A[j]; A[j]:=x
        end
end;
```

Lepší třídící algoritmy běží v čase $\mathcal{O}(N \log N)$. V minulé Kuchařce jsme si ukázali datovou strukturu zvanou halda. Do haldy umíme vkládat prvek v čase $\mathcal{O}(\log N)$ a odebírat z haldy nejmenší prvek taktéž v čase $\mathcal{O}(\log N)$. Což tedy si nejdříve všechna tříděná čísla postupně vložit do haldy a následně z haldy N-krát odebrat minimum? A ejhle, vymysleli jsme algoritmus, který na první, vkládací fázi potřebuje $N \log N$ kroků, na fázi vybírací taktéž $N \log N$ kroků. *Třídění haldou (HeapSort)* tedy běží celé v čase $\mathcal{O}(N \log N)$ a lze ho výhodně naprogramovat tak, aby potřeboval pouze jediné pole.

```
procedure HeapSort(var A: Pole);
var i,x: integer;
  { "zabublání" prvku v haldě }
  procedure bubbledown(n,i: integer);
  var j,x: integer;
  begin
    while 2*i <= n do begin
      j := 2*i;
      if (j<n) and (A[j+1] > A[j]) then j:=j+1;
      if A[i] >= A[j] then break;
      x := A[i];
      A[i] := A[j];
      A[j] := x;
      i := j;
   end;
  { postav haldu }
  for i:=N div 2 downto 1 do bubbledown(N,i);
  { vybírej nejmenší prvek }
  for i:=N downto 2 do begin
    x := A[1];
    A[1] := A[i];
    A[i] := x;
    bubbledown(i-1,1);
  end;
end;
```

Třídění sléváním (MergeSort) je založené na principu slévání již setříděných posloupností dohromady v jedinou setříděnou. Dvě posloupnosti jednoduše v lineárním čase slijeme tak, že se díváme na nejmenší prvky obou posloupností a na výstup vydáme menší z nich, který z jeho posloupnosti poté odmažeme. Algoritmus, který umí slévat dvě posloupnosti uvnitř jednoho pole, je dosti složitý, my budeme proto výslednou slitou posloupnost ukládat do pomocného pole.

Na počátku bude každý prvek jednoprvkovou setříděnou posloupností. V další fázi se ze všech sousedních jednoprvkových slitím vytvoří dvouprvkové posloupnosti, poté ze všech sousedních dvouprvkových posloupností čtyřprvková, a tak dále, obecně v i-té fázi ze dvou 2^{i-1}-prvkových posloupností vytvoříme 2^i-prvkovou. Takto pokračujeme, dokud nezbude jediná setříděná posloupnost – celé pole. Zjevně v každé fázi vykonáme $\mathcal{O}(N)$ kroků, dvě posloupnosti totiž umíme slévat v lineárním čase vzhledem k jejich délce, a naše posloupnosti pokrývají celé tříděné pole. Počet fází bude $\mathcal{O}(\log N)$, neboť v každé fázi pracujeme s dvojnásobně velkými posloupnostmi a tedy nejpozději v ($\log_2 N$)-té fázi již celé pole bude jedinou setříděnou posloupností. Dohromady proto MergeSort spotřebuje čas $\mathcal{O}(N \log N)$.

V samotném programu ovšem s výhodou použijeme metodu Rozděl & Panuj: necháme si zvlášť rekurzivně setřídit levou i pravou polovinu pole a výsledky slijeme. Nevýhodou algoritmu MergeSort je, že na slévání potřebuje ještě jedno pomocné pole.

```
procedure MergeSort(var A,P:Pole; l,r:integer);
var i,j,k,s: integer;
begin
  s:=(l+r) div 2;
  if l < s then MergeSort(A, P, l, s);
  if s+1 < r then MergeSort(A, P, s+1, r);
  i:=l;
```

```
  j:=s+1;
  k:=l;
  while (i <= s) and (j <= r) do
  begin
    if A[i] <= A[j] then
      begin P[k]:=A[i]; i:=i+1 end
    else
      begin P[k]:=A[j]; j:=j+1 end;
    k:=k+1
  end;
  while i <= s do
    begin P[k]:=A[i]; i:=i+1; k:=k+1 end;
  while j <= r do
    begin P[k]:=A[j]; j:=j+1; k:=k+1 end;
  for k:=l to r do
    A[k]:=P[k]
end;
```

Jako poslední rychlý algoritmus si předvedeme *QuickSort*. Podobně jako MergeSort, i on je postaven na jednoduché myšlence a metodě Rozděl & Panuj. Nejprve si zvolíme nějaké číslo, kterému budeme říkat pivot. Více o jeho volbě později. Poté prvky v poli takto přeuspořádáme: v levé části pole budou pouze prvky menší než pivot, v prostřední části prvky rovné pivotu a v pravé části prvky větší než pivot. A poté si prvky levé i pravé části rekurzivním voláním setřídíme. Jistě tak správně setřídíme celé pole.

Velká zrada ovšem spočívá ve volbě pivota. Pro naše účely by se hodilo, aby po přeházení prvků levá i pravá část pole byly přibližně stejně velké. Nejlepší volbou pivota by tedy byl tzv. *medián* tříděného úseku, tj. prvek takový, jenž by se po setřídění vyskytoval uprostřed úseku. Přeuspořádání jistě zvládneme v lineárním čase a stejným argumentem jako při analýze MergeSortu (v i-té úrovni rekurze třídíme úseky dlouhé $N/2^i$) dostáváme časovou složitost $\mathcal{O}(N \log N)$. Ačkoli existuje algoritmus, který medián pole nalezne v čase $\mathcal{O}(N)$, v QuickSortu se obvykle nepoužívá, jelikož konstanta u členu N je dost velká. Namísto toho se většinou pivot volí libovolně z našeho dosud nesetříděného úseku – zkrátka se sáhne někam do pole a nalezený prvek se prohlásí za pivot. Dá se ukázat, že algoritmus s velmi vysokou pravděpodobností poběží v čase $\mathcal{O}(N \log N)$. Důkaz je netriviální a my ho zde nebudeme předvádět.

Je však potřeba si uvědomit, že QuickSort může v ojedinělých případech nepříjemně zpomalit. Představme si, že pivot v každém rekurzivním volání neustále nešťastně volíme jako největší prvek z tříděného úseku. V takovém případě bude pravá část pole prázdná a levá bude mít velikost $N-1$. Rekurze dosáhne hloubky N a čas tak vyjde $\mathcal{O}(N^2)$.

```
procedure QuickSort(var A:Pole; l,r:integer);
var i,j,k,x: integer;
begin
  i:=l; j:=r;
  k:= A[(i+j) div 2];
  repeat
    while A[i] < k do i:=i+1;
    while A[j] > k do j:=j-1;
    if i<=j then
    begin
      x:=A[i]; A[i]:=A[j]; A[j]:=x;
      i:=i+1;
      j:=j-1;
    end
  until i >= j;
  if j>l then QuickSort(A, l, j);
  if i<r then QuickSort(A, i, r);
end;
```

Trošičku stranou všech zmíněných algoritmů stojí *přihrádkové třídění (RadixSort)*. RadixSort se vyznačuje tím, že tříděná čísla sice musí ležet v jistém nevelkém intervalu, nicméně s touto podmínkou je zvládne setřídit v lineárním čase. Předpokládejme tedy, že všechna tříděná čísla jsou z intervalu $[D, H]$. Připravíme si $H - D + 1$ přihrádek indexovaných čísly $D, D+1, \ldots, H-1, H$. Postupně čteme tříděné záznamy a „sypeme“ je do té přihrádky, jejíž index se shoduje s klíčem záznamu. Po přečtení vstupu projdeme přihrádky od nejmenší k největší a vypíšeme obsažené záznamy. Zjevně jsme data setřídili v čase $\mathcal{O}(N + (H - D))$, tedy $\mathcal{O}(N)$ pokud jsou D a H konstantní.

Pokud by bylo třeba přihrádek příliš mnoho, můžeme použít tzv. *víceprůchodový* RadixSort. V první fázi čísla rozdělíme do přihrádek podle nejméně významné cifry. Je důležité, aby se uvnitř přihrádky zachovalo pořadí, v jakém byla čísla vložena. Poté postupně vybereme prvky od nejnižší po nejvyšší přihrádku. V druhé fázi čísla opětovně rozdělíme do přihrádek, tentokrát podle druhé cifry, a opět je z nich vybereme. A tak dále, dokud nezpracujeme všechny cifry (jejichž počet je podle zadání úlohy nějaký malý). Zbývá si jen rozmyslet, že po i-té fázi jsou čísla správně utříděná podle i-té a nižších cifer.

```
const
  D = 1;
  H = 100;
procedure RadixSort(var A: Pole);
var C: array[D..H] of integer;
    i,j,k: integer;
begin
  for i:=D to H do C[i]:=0;
  for i:=1 to N do C[A[i]] := C[A[i]] + 1;
  k:=1;
  for i:=D to H do
    for j:=1 to C[i] do
    begin
      A[k]:=i;
      k:=k+1;
    end
end;
```

Na závěr našeho povídání o třídících algoritmech si ukážeme, že třídit obecné údaje, se kterými neumíme provádět nic jiného, než je navzájem porovnávat, rychleji než $\mathcal{O}(N \log N)$ nejen nikdo neumí, ale také ani umět nemůže. Libovolný třídící algoritmus založený na porovnávání a prohazování prvků totiž musí na některé vstupy vynaložit řádově alespoň $N \log N$ kroků. (RadixSort na první pohled tento výsledek porušuje, na druhý však už ne, když si uvědomíme, o jak speciální druh tříděných dat se jedná.)

Třídící algoritmus v průběhu své činnosti nějak porovnává prvky a nějak je přehazuje. Provedeme myšlenkový experiment. Pozměníme algoritmus tak, že nejdříve bude pouze porovnávat, podle toho zjistí, jak jsou prvky v poli uspořádány, a když už si je jistý správným pořadím, prvky najednou popřehází. Předpokládejme pro jednoduchost, že všechny tříděné údaje jsou navzájem různé. Porovnávací činnost algoritmu si můžeme popsat tzv. *rozhodovacím stromem*. Zde je příklad rozhodovacího stromu pro tříprvkové pole:


```
}

static int cmps (const void *a, const void *b)
{
  const char *sa = *(const char **) a;
  const char *sb = *(const char **) b;
  if (!sa)
    return 1;
  if (!sb)
    return -1;
  return strcmp (sa, sb);
}

int main (void)
{
  int n, k, i;
  scanf ("%d%d", &n, &k);
  read_words (&n, k, words + k);
  while (1)
    {
      read_words (&n, k, words);
      qsort (words, 2 * k, sizeof (char *), cmps);
      if (!n)
        break;
      for (i = 0; i < k; i++)
        {
          puts (words[i]);
          free (words[i]);
        }
    }
  for (i = 0; words[i]; i++)
    {
      puts (words[i]);
      free (words[i]);
    }
  return 0;
}
```

Úloha 16-1-2 – Lokální minimum – Program

```
program minimum;
const
  MAXP = 100;
var
  l, r, m : Integer; {Levy konec, pravy konec a stred zkoumaneho intervalu}
  n : Integer;       {Delka posloupnosti}
  A : Array[0..MAXP+1] of Integer; {Pole s prvky}
begin
  {Inicializace - vlastne jeste neni soucasti hledani minima}
  Read(n);
  for l := 1 to n do
    Read(A[l]);
  A[0] := MAXINT;
  A[n+1] := MAXINT;

  {Zaciname hledat lokalni minimum}
  l := 0; r := n+1;
  while l <> r do begin
    m := (l+r) div 2;
    if A[m] < A[m+1] then
      if A[m-1] < A[m] then
        r := m-1
      else begin
        l := m;
```

$p(\omega)$ vyhovujících prvků ω. Pokud je jasné, co je náhodná proměnná, místo $\Pr_\omega$ často píšeme jenom Pr. Stejně tak závorky okolo $(\mathbf{E}F)$ budeme vynechávat.

Zpět k příkladu 2: Funkci S můžeme vyjádřit jako součet dvou jednodušších funkcí $S_1((x,y)) = x$ a $S_2((x,y)) = y$, jejichž střední hodnota je $\mathbf{E}S_1 = \mathbf{E}S_2 = \frac{7}{2}$ (stačí si všimnout, že každou hodnotu x dosáhneme pro 6 různých y, takže jsme ve stejné situaci jako v předchozím příkladu). Proto

$$\mathbf{E}S = \mathbf{E}(S_1 + S_2) = (\mathbf{E}S_1) + (\mathbf{E}S_2) = \frac{7}{2} + \frac{7}{2} = 7.$$

Zpět k analýze našeho algoritmu: Zkusme spočítat průměrný počet volání funkce *random2* během výpočtu, čili střední hodnotu funkce R, která každému možnému průběhu výpočtu (tj. tomu, jaké jsme dostali náhodné bity – zbytek je již jednoznačně určen) přiřadí, kolikrát byla v tomto případě *random2* zavolána. To můžeme elegantně spočítat třeba tak, že R vyjádříme jako součet funkcí $R_1, R_2, \ldots$, kde $R_i = 1$, pokud i-té volání nastalo, jinak $R_i = 0$. Pak dostaneme:

$$\mathbf{E}R = \mathbf{E}\left(\sum_{i=1}^{\infty} R_i\right) = \sum_{i=1}^{\infty} \mathbf{E}R_i.$$

Pravděpodobnost toho, že $R_i = 1$, jsme již spočítali a je to $p_{i-1} < 2^{1-i}$, takže podle $(*)$ dostaneme:

$$\mathbf{E}R_i = 0 \cdot \Pr[R_i = 0] + 1 \cdot \Pr[R_i = 1] = 0 + p_{i-1} < 2^{1-i}.$$

Z toho:

$$\mathbf{E}R < \sum_{i=1}^{\infty} 2^{1-i} = 2 \cdot \sum_{i=1}^{\infty} 2^{-i} = 2 \cdot 1 = 2$$

[to není nic jiného než součet nekonečné geometrické řady].

Spočítat z toho průměrnou časovou složitost našeho algoritmu je už hračka: nechť $T(\omega)$ značí časovou složitost výpočtu ω, čili $\mathcal{O}(R(\omega) \cdot \log N)$. Pak

$$\mathbf{E}T = \mathbf{E}\mathcal{O}(R \cdot \log N) = \mathcal{O}((\mathbf{E}R) \cdot \log N) = \mathcal{O}(\log N).$$

[Cvičeníčko: dokažte, že střední hodnotu lze „přetahovat přes O“.]

Takže i přesto, že náš algoritmus na první pohled vypadal velmi neefektivně, v průměrném případě běží jen logaritmicky dlouho (a tím pádem také funkci *randbit* volá jen logaritmicky-krát).

Jiný, možná elegantnější, způsob odvození hodnoty $\mathbf{E}R$ můžeme získat takto: Algoritmus v každém případě zavolá funkci *random2* jednou a pokud získá moc velké číslo (to nastává s pravděpodobností $p < 1/2$), dostane se opět na začátek a nic si z předchozího nepamatuje. Proto platí:

$$\mathbf{E}R = 1 + p \cdot \mathbf{E}R,$$

kterážto rovnice má řešení

$$\mathbf{E}R = 1/(1-p) < 2.$$

[Cvičeníčko: chvíli přemítejte o tom, proč je tato úvaha opravdu korektní.]

Poznámka: Někteří řešitelé vymýšleli různé důmyslné (a někdy i korektní) způsoby, jak část náhodných bitů z neúspěšných pokusů „recyklovat“ v pokusech dalších. To samozřejmě lze, ale jediné, co tím dokážeme, je pro některá N zmenšit multiplikativní konstantu schovanou v O-čku, což nestojí za tu námahu. Kdybychom ovšem řešili obecnější úlohu a bylo naším cílem generovat n náhodných čísel namísto jednoho, začalo by se recyklování náhodnosti vyplácet a došli bychom (po úctyhodném množství počítání) k tomu, že čím větší je n, tím více se průměrný počet náhodných bitů blíží k $n \cdot \log N$ (přesně, tedy bez O-čka!). To si ale třeba nechme na jindy, pro zvídavé napovím jen, že jeden ze způsobů, jak toho dosáhnout, je vzít populární kompresní algoritmus řečený aritmetické kódování, inicializovat mu pravděpodobnosti všech znaků na $1/N$ a poté nechat dekomprimovat posloupnost náhodných bitů.

Ještě jedna poznámka: V mnohých řešeních se vyskytlo počítání dvojkových logaritmů pomocí vzorců typu

$$\lfloor \log_2 N \rfloor = \mathrm{trunc}(\log(N)/\log(2)).$$

To by bylo správně, nebýt jednoho zádrhele: počítač nepočítá s opravdovými reálnými čísly, nýbrž pouze s jejich aproximacemi. Proto všechny výpočty v typech jako je *real*, *float* apod. jsou zatíženy chybami, které i po zaokrouhlení mohou být podstatné. Často se samozřejmě dá dokázat, že v daném případě je výsledek správný, ale nebývá to jednoduché, takže pokud se chcete takovým problémům vyhnout, je praktické při počítání s celými čísly používat jen celočíselné operace.

Tak, to už je pravděpodobně všechno.

Martin Mareš

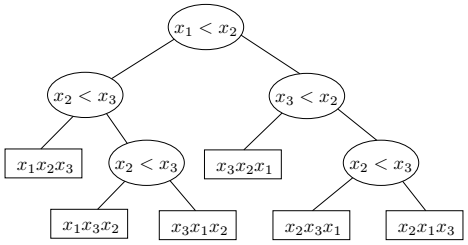
Úloha 16-1-1 – Telefonní seznam – Program

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

#define MAX_K 100
#define L 100

static char *words[2 * MAX_K + 1];

static void read_words (int *n, int k, char **tgt)
{
  char buffer[L + 1];
  while (*n > 0 && k-- > 0)
    {
      scanf ("%s", buffer);
      tgt[k] = strdup (buffer);
      (*n)--;
    }
  while (k-- > 0)
    tgt[k] = NULL;
```

Každý vrchol obsahuje porovnání dvou prvků x a y, v levém podstromu daného vrcholu je činnost algoritmu pokud $x < y$, v pravém podstromu činnost při $x \geq y$. V listech je už jisté správné pořadí prvků.

Každému algoritmu odpovídá nějaký rozhodovací strom a každý průběh činnosti algoritmu odpovídá průchodu rozhodovacím stromem od kořene do nějakého listu. Naším cílem bude ukázat, že v libovolném rozhodovacím stromu (a tedy i libovolném odpovídajícím algoritmu) bude existovat cesta z kořene do nějakého listu (neboli výpočet algoritmu) délky $N \log N$.

Kolik maximálně hladin h a tedy i jaká nejdelší cesta se v takovém stromu může vyskytnout? Náš strom má tolik listů, kolik je možných pořadí tříděných prvků, tedy právě $N!$. Různým pořadím odpovídají různé listy. V i-té hladině se počet vrcholů oproti $(i-1)$-ní hladině nejvýše zdvojnásobí, platí tedy

$$2^h \geq N!,$$

neboli

$$h \geq \log_2(N!).$$

Existuje několik způsobů, jak si h zdola odhadnout, my použijeme skutečnost, že

$$n^n \geq n! \geq n^{n/2}.$$

Rozepsáno

$$h \geq \log_2(N!) \geq \log_2(N^{N/2}) \geq \frac{N}{2} \log N.$$

Vidíme tedy, že pro každý třídící algoritmus existuje vstup, na kterém se bude muset provést alespoň $N \log N$ kroků. Zkuste si též rozmyslet (drobnou modifikací předchozího důkazu), že ani *průměrný* čas třídění nemůže být rychlejší než $N \log N$.

Dnešní menu vám servíroval
Tomáš Valla

Svá řešení nám zasílejte do 31. prosince 2003 na známou adresu:

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

Vzorová řešení první série šestnáctého ročníku KSP

16-1-1 Telefonní seznam

Jak většina řešitelů správně poznala, tato úložka se dá řešit použitím haldy. Snadno si rozmyslíme, že když „přefiltrujeme“ zadanou posloupnost slov haldou velikosti k – tj. do haldy vložíme prvních k prvků a pak střídavě odebíráme minimum a přidáváme další prvky – je výsledná posloupnost setříděná. Přidání do haldy i odebrání z ní vyžaduje $\mathcal{O}(\log k)$ porovnání a těchto operací provedeme $\mathcal{O}(n)$. Je ovšem nutné vzít v úvahu také to, že porovnání řetězců nelze provést v konstantním čase. V nejhorším případě se může stát, že se dva řetězce liší až na některé z posledních pozic, pak jejich porovnání zabere čas $\Theta(L)$, kde L je jejich délka. Výsledná časová složitost tohoto řešení je tedy $\mathcal{O}(n \cdot L \cdot \log k)$. V paměti si stačí udržovat pouze haldu, prvky se načítají postupně a lze je rovnou vypisovat, tedy paměťová složitost je $\mathcal{O}(k \cdot L)$.

Existují ovšem nejméně dvě řešení dosahující stejné nebo lepší časové složitosti:

Je možné prostě zapomenout na k a setřídit slova pomocí přihrádkového třídění. Toto řešení pracuje s časovou složitostí $\mathcal{O}((n+p) \cdot L)$, kde p je počet písmen v abecedě, a paměťovou složitostí $\mathcal{O}(n \cdot L)$.

Další řešení pracuje na podobném principu jako první řešení, nicméně vyhýbá se použití haldy. Funguje takto: z posloupnosti slov si vezmeme prvních $2k$ a setřídíme je, to vyžaduje $\mathcal{O}(k \cdot \log k)$ porovnání. k nejmenších z nich vypíšeme, zbytek přidáme na začátek posloupnosti a opakujeme. Pro důkaz správnosti je potřeba uvědomit si dvě věci:

1) Vzhledem k tomu, že pozice každého prvku v posloupnosti se od své správné pozice v setříděné posloupnosti liší nejvýše o k, k nejmenších musí být mezi prvními $2k$ prvky posloupnosti, a tedy po setřídění tohoto úseku budou na správných pozicích.

2) Přerovnáním zbývajících k prvků nepokazíme to, že prvky jsou ve vzdálenosti nejvýše k od správné pozice. Ukážeme to sporem. Nechť tedy takový „pokažený“ prvek existuje. Buď x největší takový. Pokud je jeho správná pozice (v původní nezkrácené posloupnosti) nejvýše $2k$, nemohli jsme nic pokazit. Tedy tato pozice je nějaké q, $q > 2k$. Označme t novou pozici slova x, víme že $q - t > k$. Jestliže $t = 2k$, je x vůbec největší z prvních $2k$, a mohli jsme ho posunout pouze směrem ke q. Tedy $t < 2k$. Slovo y na pozici $t+1$ je větší než x, jeho správná pozice je tedy alespoň $q+1$. Nicméně $(q+1)-(t+1) > k$ a to je spor s tím, že x je největší s touto vlastností.

Časová složitost tohoto řešení je opět $\mathcal{O}(n \cdot L \cdot \log k)$, protože tříděných úseků je $\mathcal{O}(n/k)$. Paměťová je $\mathcal{O}(k \cdot L)$, protože si stačí pamatovat aktuálně tříděný úsek. Program implementuje toto řešení, protože se nejsnáze naprogramuje a já jsem líný.

Zdeněk Dvořák

16-1-2 Lokální minimum

Je podivu hodné, kolik lidí se upokojilo s triviálním algoritmem o lineární časové složitosti – i přes explicitní upozornění, že to jde lépe. Pravda, nechalo se najít pár vylepšení typu číst každý druhý prvek, používat pouze konstatní paměťovou složitost, leč k nalezení logaritmické složitosti (co asi tak může být mezi $\mathcal{O}(n)$ a nesmyslem $\mathcal{O}(1)$?) je tato cesta neperspektivní. Jest lyže má něco logaritmickou složitost, obvykle se tam nějakým způsobem rovnoměrně rozdělují vstupní data – v tomto případě se bude půlit celý interval a po výběru správné poloviny se úloha bude rekurzivně řešit pouze uvnitř této.

V intervalu $[x_0 \ldots x_{n/2} \ldots x_{n+1}]$ existuje Lokální Minimum (sporem). Vezmeme prostřední prvek. Pokud je LM, úloha je vyřešena. Jinak se LM nutně nalézá v polovině (či

v obou), jejíž krajní prvek $x_{n/2\pm1}$ je menší než $x_{n/2}$.

Proč? Nechť bez újmy na obecnosti $x_{n/2} > x_{n/2+1}$. Pro spor předpokládejme, že v pravé polovině není LM. Proto nutně $x_{n/2+1} > x_{n/2+2}$ a stejným argumentem pokračuji až dostávám $x_n > x_{n+1}$, což je hledaný spor, neboť $x_{n+1} = \infty$.

Úlohu proto stačí rekurzivně spustit na vybranou polovinu.

I Plovací sval nahlédne, že to lze provést nejvýše logaritmuskrát, z čehož plyne i celková složitost algoritmu $\mathcal{O}(\log n)$. Všechna čísla jsou uložena v poli, a tedy paměťová složitost je $\mathcal{O}(n)$.

Pavel Šanda

16-1-3 Důl

Většina z vás správně rozpoznala, že tento příklad je jedním z oněch dvou, na které se má použít recept z naší programátorské kuchařky. Úlohu lze vcelku jednoduše vyřešit modifikovaným Dijkstrovým algoritmem, kde místo délky cesty uvažujeme nosnost nejhorší silnice, kterou cesta obsahuje.

Zmiňme tedy jen stručně rozdíly oproti algoritmu z naší kuchařky. Ohodnocení měst není tvořeno délkami nejkratších cest, ale maximálními nosnostmi dosud nalezených cest, kde nosnost cesty je minimum nosností všech silnic, které obsahuje. V každém kroku vybereme dočasně ohodnocené město M s největším ohodnocením, a jeho ohodnocení prohlásíme za trvalé. Poté zkusíme nalézt výhodnější cesty přes město M do zbylých (dočasně ohodnocených) měst. Poznamenejme, že nosnost cesty přes město M je minimum z nosnosti cesty do města M a silnice z města M do sousedního města. Důkaz správnosti se provede podobně jako důkaz správnosti původního Dijkstrova algoritmu.

Přímočará implementace právě popsaného algoritmu bez použití haldy má časovou složitost $\mathcal{O}(n^2)$ a paměťovou složitost $\mathcal{O}(n + m)$, kde n je počet měst a m je počet silnic. Řešení bez použití haldy, lze nalézt ve vzorovém programu. Za pomoci haldy, lze dosáhnout časové složitosti $\mathcal{O}(m \log n)$.

Existuje i alternativní řešení pomocí datové struktury zvané *disjoint find union* (DFU), kterou si podrobně popíšeme v některém z následujících dílů naší kuchařky. Datová struktura DFU nám umožňuje udržovat si rozklad množiny na disjunktní podmnožiny. Na začátku je množina rozložena na jednobodové podmnožiny a v průběhu práce s ní můžeme podmnožiny sjednocovat do větších. Tato datová struktura nám pak umožňuje odpovídat na dotazy, zda dva prvky jsou ve stejné podmnožině či nikoliv. Provedeme-li celkem K operací na n-prvkové množině, pak čas spotřebovaný touto datovou strukturou je nejvýše $\mathcal{O}((K+n)\cdot\alpha(n))$, kde $\alpha(n)$ je inverzní funkce k tzv. Ackermannově funkci. Poznamenejme, že funkce $\alpha(n)$ roste s n velmi velmi pomalu a pro počet atomů ve vesmíru je tato tato funkce rovna 4.

Jak tedy takovéto řešení bude fungovat? Nejdříve si silnice setřídíme od té s největší nosností po tu s nejmenší. Následně budeme postupně povolovat silnice, které můžeme použít, tak dlouho dokud nebude existovat cesta z dolu do továrny jen po povolených silnicích. Silnice povolujeme od té s největší nosností. Na začátku není žádná silnice povolena. Množiny měst, mezi kterými existuje cesta jen po povolených silnicích, tvoří právě rozklad množiny všech měst, který udržujeme pomocí DFU. Právě popsané řešení má časovou složitost $\mathcal{O}(m \log m)+\mathcal{O}((m+n)\cdot\alpha(n)) = \mathcal{O}(m \log n)$ a paměťovou složitost $\mathcal{O}(n + m)$.

Dan Král'

16-1-4 Neruš mé trojúhelníky!

Představitele Bosstánu by jistě potěšilo, že se našlo tolik řešení jejich šifrovacího problému. Avšak ne všechna řešení byla dost rychlá, aby se dala používat. Spousta z vás řešila problém naprosto přímočarým způsobem, který ovšem běží v čase $\mathcal{O}(n^3)$, a tedy velice pomalu už pro malé n. S malým pozorováním lze tento algoritmus vylepšit až na složitost $\mathcal{O}(n^2)$. Algoritmus vlastně funguje stejně jako přímočaré řešení jen s tím rozdílem, že si pro každou dvojici bodů pamatujeme interval, ve kterém může ležet třetí bod (a výsledný trojúhelník přitom bude obsahovat střed) a ten pouze updatujeme. Avšak ani toto řešení, jak mnozí poznali, není nejlepší. Vzorové řešení pracuje v čase $\mathcal{O}(n \log n)$ následujícím způsobem:

Nejprve si vstupní data, body udané úhly, setřídíme. Pak si problém převedeme na opačnou úlohu, tedy kolik trojúhelníku neobsahuje střed. Tato úloha se řeší velice jednoduše, přesněji v lineárním čase. Nejprve si určíme, kdy trojúhelník neobsahuje střed. To nastává právě tehdy, když jsou body trojúhelníka v intervalu dlouhém < 180 stupňů, tedy jednodušeji, jsou pouze na jedné půlce kružnice. A tyto body se již určí velice jednoduše. Postupně volíme body na kružnici a pro ně určujeme nejvzdálenější bod po směru kružnice takový, že jejich úhlová vzdálenost je < 180 stupňů. Protože konec intervalu pro bod y následující na kružnici po x stačí hledat za koncem intervalu pro bod x, celou kružnici objedeme maximálně 2-krát. Tedy algoritmus má lineární složitost. Zbývá snad už jen dodat, že je nutné si dát pozor na to, abychom započítali každý trojúhelník právě jednou.

Složitost třídění je $\mathcal{O}(n \log n)$ a algoritmus na hledání počtu trojúhelníků neobsahujících střed má složitost $\mathcal{O}(n)$. Tedy celková časová složitost algoritmu je $\mathcal{O}(n \log n)$ a paměťová složitost je $\mathcal{O}(n)$.

Tomáš Vyskočil

16-1-5 Pravděpodobnostní algoritmy

Jak se ukázalo (a jak ostatně napoví i pohled na výsledkovou listinu), generování náhodných čísel s rovnoměrným rozdělením pravděpodobností je problém trochu potvornější, než jak na první pohled vypadá. Nejdříve si ho proto vyřešme pro případ, kdy je N mocninou dvojky, tedy rovné nějakému 2^k. Tehdy nám stačí k-krát si „hodit korunou“ (tj. zavolat funkci *randbit*) a výsledek si vyložit jako dvojkový zápis nějakého čísla mezi 0 a $N-1$:

```
int random2 (int N)
{
  int z = 0;
  N--;
  while (N)
    {
      z = 2*z + randbit ();
      N /= 2;
    }
  return z;
}
```

(všimněte si, že k ani nemusíme počítat, že stačí $N-1$ postupně dělit dvojkou tak dlouho, než vyjde 0).

Všechny výsledky jsou určitě stejně pravděpodobné, protože každé číslo je určeno právě jedním dvojkovým zápisem a pravděpodobnost každého dvojkového zápisu je $(1/2)\cdot(1/2)\cdot\ldots\cdot(1/2) = 2^{-k} = 1/N$.

Ale co když N nebude mocninou dvojky? Mohli bychom například zvolit nějaké $N' > N$, které mocninou dvojky bude (a takové určitě najdeme mezi N a $2N$), vygenerovat x v rozsahu 0 až $N'-1$ a výsledek nějak „upravit“, aby byl vždy menší než N. Jak to ale přesně provést?

Pokus č. 1: Spočíst x mod N: to nebude fungovat, protože číslo 0 můžeme vygenerovat dvojím způsobem (pro $x = 0$ i $x = N$), zatímco třeba číslo $N-1$ pouze jedním způsobem ($x = N-1$), a proto má 0 dvojnásobnou pravděpodobnost než $N-1$.

Pokus č. 2: Postupné generování dvojkového čísla po bitech si můžeme také vyložit jako hledání nějakého čísla v poli $(0, 1, \ldots, N-1)$ půlením intervalů, přičemž v každém kroku se nerozhodujeme podle nerovnosti s hledaným číslem, nýbrž náhodně. To je bezpochyby pro $N = 2^k$ totéž, ale v ostatních případech musíme dříve nebo později narazit na interval, který se na dva stejně dlouhé rozdělit nedá. Ať už to ošetříme jakkoliv, opět dostaneme nestejné pravděpodobnosti vygenerovaných čísel.

A není náhodou, že se nám nedaří rovnoměrného rozdělení dosáhnout – ono to totiž na žádný konečný počet volání funkce *randbit* nelze. Poslyšte důkaz: Kdyby stačilo h hodů mincí, můžeme si program upravit tak, aby *randbit* volal vždy právě h-krát (počítali bychom si, kolikrát ho zavolal, a nakonec bychom doplnili příslušný počet volání, která by nijak neovlivnila výsledek programu). Program pak může probíhat 2^h různými způsoby podle toho, jakou posloupnost náhodných bitů od funkce *randbit* dostal, přičemž všechny tyto průběhy jsou stejně pravděpodobné. Každé číslo x, které program může vygenerovat, pak odpovídá některým z těchto průběhů (může jich být více, protože lze dospět různými cestami k témuž výsledku) a označíme-li si počet takových průběhů c_x, bude pravděpodobnost vygenerování čísla x rovna přesně $c_x/2^h$. Jenže pokud není N mocnina dvojky, nemůžeme žádanou pravděpodobnost $1/N$ vyjádřit jako $c_x/2^h$ pro žádné c_x. [Pokud je $a/b = c/d$, pak $ad = bc$, čili v našem případě $2^h = N \cdot c_x$. Jenže v rozkladu levé strany na součin prvočísel vystupují jen dvojky, v rozkladu pravé i jiná prvočísla.] Takže náš program opravdu nemůže fungovat.

Zkusme tedy na konečnost na chvíli zapomenout a navrhnout naprosto přímočaré řešení: Vygenerujeme náhodné číslo mezi 0 a $N'-1$. Pokud bude menší jak N, prohlásíme ho za výsledek, jinak ho zahodíme a generujeme znovu atd.:

```
int random (int N)
{
  int x;
  do
    x = random2 (N);
  while (x >= N);
  return x;
}
```

(zde zneužíváme toho, že výpočet k ve funkci *random2* zaokrouhluje nahoru, takže místo do $N-1$ generujeme až do $N'-1$, jak potřebujeme).

Jelikož výsledky funkce *random2* jsou, jak už víme, všechny stejně pravděpodobné a my si z nich pouze vybíráme a v případě, že si číslo nevybereme, zapomeneme veškerý dosavadní průběh výpočtu, musí být i výsledky naší nové funkce všechny stejně pravděpodobné.

Háček ovšem může být v tom, že nedokážeme předpovědět, kolik iterací bude naše funkce na vygenerování jednoho čísla potřebovat. Může se to povést už napoprvé, ale také se to nemusí povést vůbec – třeba tehdy, bude-li nám funkce *randbit* vracet stále samé jedničky. Nicméně pravděpodobnost toho, že se nám první číslo nebude hodit, je $p = 1-N/N' < 1/2$, pravděpodobnost toho, že ani to druhé ne, je $p^2 < 1/4$, …, l neúspěšných pokusů nastane s pravděpodobností $p^l < 2^{-l}$ a nekonečná posloupnost neúspěchů má pravděpodobnost 0. [Což neznamená, že by nemohla nastat.]

Zkusme tedy spočítat, kolik pokusů budeme potřebovat *v průměru*. Nejprve si ale nadefinujme, co takový průměr přesně je: Mějme nějakou množinu náhodných jevů $\Omega = \{\omega_1, \omega_2, \ldots\}$, přičemž jev $\omega \in \Omega$ nastává s pravděpodobností $p(\omega)$. Pak pro každou funkci F, která přiřazuje těmto jevům nějaká reálná čísla, můžeme nadefinovat *střední hodnotu* $\mathbf{E}F$ takto:

$$\mathbf{E}F = \sum_{\omega\in\Omega} F(\omega)\cdot p(\omega).$$

Pokud mají všechny jevy stejnou pravděpodobnost, tedy $\forall\omega \in \Omega : p(\omega) = 1/|\Omega|$, splývá tato definice s obyčejným (aritmetickým) průměrem všech funkčních hodnot; pokud ale jsou některé jevy pravděpodobnější, dáváme jim větší váhu, a tak průměr ovlivňují více.

Příklad 1: Pokud budeme házet kostkou a budeme chtít spočítat, kolik nám v průměru padne, bude naše množina $\Omega = \{1, 2, 3, 4, 5, 6\}$ a všechna $p(\omega)$ budou rovna $1/6$. Pak si stačí zvolit funkci $I(x) = x$ a spočítat její střední hodnotu:

$$\mathbf{E}I = \sum_{x=1}^{6} I(x)\cdot p(x) = \sum_{x=1}^{6} x\cdot\frac{1}{6} = \frac{1}{6}\cdot\sum_{x=1}^{6} x = \frac{21}{6} = \frac{7}{2}.$$

Příklad 2: Házíme dvěma kostkami a chceme spočíst průměrný součet. Množina Ω bude tentokrát zahrnovat všechny uspořádané dvojice (x, y), kde $1 \le x, y \le 6$, každá dvojice bude mít pravděpodobnost $1/36$ a střední hodnotu budeme počítat z funkce $S((x, y)) = x+y$ [dvojích závorek se nelekejte, pouze vyjadřují, že parametrem funkce nejsou dvě čísla, nýbrž jedna uspořádaná dvojice čísel]. Mohli bychom počítat otrocky podle definice, ale raději si všimneme jedné zajímavé vlastnosti středních hodnot:

Odbočka: Střední hodnota je *lineární*, to znamená, že platí:

$$\begin{aligned}\mathbf{E}(F+G) &= \sum_{\omega\in\Omega}(F+G)(\omega)\cdot p(\omega) = \\ &= \sum_{\omega\in\Omega}(F(\omega)+G(\omega))\cdot p(\omega) = \\ &= \left(\sum_{\omega\in\Omega}F(\omega)\cdot p(\omega)\right)+\left(\sum_{\omega\in\Omega}G(\omega)\cdot p(\omega)\right) = \\ &= (\mathbf{E}F)+(\mathbf{E}G).\end{aligned}$$

a také (což se dokáže obdobně):

$$\mathbf{E}(\alpha\cdot F) = \ldots = \alpha\cdot(\mathbf{E}F) \qquad \text{pro každé } \alpha \in \mathbf{R}.$$

Intuitivně řečeno: $\mathbf{E}$ lze „přetáhnout“ jak přes $+$, tak přes násobení konstantou.

Také si všimněme, že střední hodnotu lze ekvivalentně definovat jako:

$$\mathbf{E}F = \sum_{x} x\cdot\Pr_\omega[F(\omega) = x], \qquad (*)$$

kde sčítáme přes všechna x z oboru hodnot funkce F a $\Pr_\omega[\textit{podmínka}]$ značíme pravděpodobnost, že náhodně vybrané $\omega \in \Omega$ splňuje *podmínku*, jinými slovy součet všech