P-III-1 Agenti

Úlohu si můžeme reprezentovat pomocí orientovaného grafu. Agenti představují vrcholy grafu. Skutečnost, že agent a může vydat rozkaz agentovi b, vyjádříme orientovanou hranou (a, b). Naším úkolem je nalézt v tomto grafu vrchol, z něhož se můžeme dostat do všech ostatních vrcholů.

Začneme prohledáváním grafu do hloubky z libovolného zvoleného vrcholu. Jestliže při tomto prohledávání navštívíme všechny vrcholy, našli jsme šéfa – je jím vrchol, kterým jsme prohledávání začali. V opačném případě pokračujeme tak, že zahájíme nové prohledávání do hloubky v jednom z vrcholů, které jsme dosud nenavštívili (dříve navštívené vrcholy grafu přitom necháme označené jako navštívené). To opakujeme tak dlouho, dokud nenavštívíme všechny vrcholy našeho grafu. Nechť r je vrchol, v němž jsme zahájili poslední prohledávání.

Tvrzení: Má-li náš graf aspoň jednoho šéfa, potom vrchol r je šéfem.

Důkaz: Předpokládejme, že náš graf má šéfa a že vrchol r není šéfem. Nechť je šéfem vrchol s. Musíme uvažovat dvě možnosti:

- *Vrchol s byl navštíven při posledním prohledávání.* To by ale znamenalo, že se do tohoto vrcholu můžeme dostat z vrcholu r (neboť vrchol r je počátkem posledního prohledávání), tudíž se můžeme dostat z vrcholu r do libovolného jiného vrcholu grafu přes s, což je však v rozporu s naším předpokladem, že r není šéfem.
- *Vrchol s byl navštíven dříve než při posledním prohledávání.* Jelikož se však z vrcholu s dá dojít do libovolného vrcholu, museli bychom také vrchol r navštívit ve stejném prohledávání jako s, takže vrchol r nemůže být začátkem posledního prohledávání.

Zbývá tedy už jen ověřit (opět prohledáváním do hloubky), zda r je skutečně šéfem grafu; v opačném případě graf nemá žádného šéfa. Časová složitost celého algoritmu je $O(M + N)$, kde N je počet vrcholů a M je počet hran grafu.

```
program Agenti;

const MAXM = 10000;
      MAXN = 100;

var rozkazuje: array [1..MAXM] of integer;
    ind_od, ind_do: array [1..MAXN] of integer;
    {agent i+6 rozkazuje agentům rozkazuje[ind_od[i]]...rozkazuje[ind_do[i]]}
    N: integer;
    navstiven: array [1..MAXN] of boolean;

procedure Nacti;
var i,M,agent: integer;
begin
   write('Počet agentů:'); readln(N);

   M:=0;
   for i:=1 to N do begin
      write('Agent ',i+6,' rozkazuje: (ukonči -1)');
      ind_od[i]:=M+1;
      read(agent);
      while (agent>0) do begin
           M:=M+1;
           rozkazuje[M]:=agent-6;
           read(agent);
      end;
      ind_do[i]:=M;
   end;
end; {procedure Nacti}

procedure Prohledej(i: integer);
var j: integer;
begin
   if not navstiven[i] then begin
      navstiven[i]:=true;
      for j:=ind_od[i] to ind_do[i] do begin
         Prohledej(rozkazuje[j]);
```

```
      end;
   end;
end; {procedure Prohledej}

var i,posledni: integer;
    je_sef: boolean;

begin
   Nacti;
   for i:=1 to N do navstiven[i]:=false;
   for i:=1 to N do begin
      if not navstiven[i] then begin
         posledni:=i;
         Prohledej(i);
      end;
   end;

   for i:=1 to N do navstiven[i]:=false;
   Prohledej(posledni);

   je_sef:=true;
   for i:=1 to N do je_sef:=je_sef and navstiven[i];

   if je_sef then
      writeln('Šéfem je agent ', posledni+6)
   else
      writeln('Žádný agent není šéfem');
end. {program Agenti}
```

P-III-2 Teploty

Snadno sestrojíme řešení, které potřebuje čas $O(K)$ na zpracování jedné hodnoty ze vstupu. Stačí si v cyklicky přepisovaném poli pamatovat posledních K vstupních hodnot. Pokaždé, když přečteme další číslo ze vstupu, pole jednoduše projdeme a vypíšeme nejmenší z hodnot uložených v poli.

Vzorové řešení vystačí s časem $O(\log K)$ na zpracování jednoho čísla. Představme si, že bychom si aktuálních K hodnot udržovali v haldě. Novou hodnotu do této haldy lehce přidáme v čase $O(\log K)$. Předtím, než vypíšeme minimum (které je uloženo v kořeni haldy), potřebujeme však ještě z haldy odstranit nejstarší hodnotu. Jak ale máme vědět, která z nich to je?

Pomůžeme si tím, že hodnoty, které nám budou přicházet, vložíme nejen do haldy, ale také do fronty. Mezi těmito dvěma datovými strukturami si budeme udržovat vzájemné odkazy, abychom v každém okamžiku dokázali o každém prvku fronty říci, kde je v haldě, a naopak.

Když tedy přijde nová hodnota, vložíme ji do haldy a na konec fronty. Následně ze začátku fronty odstraníme nejstarší hodnotu, pomocí odkazu ji najdeme v haldě a odstraníme ji také odtamtud. Nyní už jen vypíšeme hodnotu uloženou v kořeni haldy.

Obě operace s haldou mají časovou složitost $O(\log K)$, zbývající operace dokážeme provést v konstantním čase. Paměť spotřebovaná haldou i frontou je $O(K)$.

```
#include <stdio.h>
#define MAXK 100047
#define INFTY 1e10
#define SWAP(x,y) pom=(x); (x)=(y); (y)=pom

typedef struct { double val; int ptr; } tZaznam;
int K;                    // ze zadání
tZaznam H[MAXK],Q[MAXK]; // halda a fronta
int qs;                   // začátek fronty
tZaznam pom;

void init(void) { // naplníme haldu i frontu "nekonečně" velkými hodnotami
  int i;
  qs=0; H[0].val=-10000;
  for (i=0;i<K;i++) { Q[i].val=H[i+1].val=INFTY; Q[i].ptr=i+1; H[i+1].ptr=i; }
}
```

```
void bubbleup(int idx) { // bublej prvkem nahoru v haldě
  int next=idx,p1,p2;
  if (H[idx/2].val > H[idx].val) next=idx/2;
  if (next!=idx) {
    p1=H[idx].ptr; p2=H[next].ptr;
    SWAP(H[idx],H[next]);
    Q[p1].ptr=next; Q[p2].ptr=idx;
    bubbleup(next);
  }
}

void bubbledown(int idx) { // bublej prvkem dolů v haldě
  int next=idx,p1,p2;
  if (2*idx<=K)   if (H[2*idx  ].val < H[next].val) next=2*idx;
  if (2*idx+1<=K) if (H[2*idx+1].val < H[next].val) next=2*idx+1;
  if (next!=idx) {
    p1=H[idx].ptr; p2=H[next].ptr;
    SWAP(H[idx],H[next]);
    Q[p1].ptr=next; Q[p2].ptr=idx;
    bubbledown(next);
  }
}

int main(void) {
  int delptr;
  double x;
  scanf("%d ",&K); init();
  while (1) {
    scanf("%lf ",&x); if (x==-1000) break;
    delptr=Q[qs].ptr; // najdeme hodnotu, kterou je třeba vymazat z haldy
    H[delptr].val=H[K].val; H[delptr].ptr=H[K].ptr; Q[H[K].ptr].ptr=delptr;
    K--; bubbledown(delptr); bubbleup(delptr); K++; // smažeme a upravíme haldu
    Q[qs].val=H[K].val=x; Q[qs].ptr=K; H[K].ptr=qs; bubbleup(K); // vložíme
    qs=(qs+1)%K;
    printf("%g\n",H[1].val);
  }
  return 0;
}
```

P-III-3 Registrový počítač

a) Na úvod si připomeňme, že v řešení krajského kola jste se mohli kromě jiného dočíst, jak lze pomocí dvou počítadel simulovat zásobník. Pro jistotu si zopakujeme, jak na to:

Zásobník si můžeme simulovat v jednom registru (s pomocí druhého). Písmena `a`, `b`, `c` budou odpovídat číslům 1, 2, 3. Číslo uložené v registru R_1 bude představovat náš zásobník – když ho zapíšeme v poziční soustavě o základu 4, jednotlivé cifry budou představovat vložené hodnoty (cifra na místě jednotek bude naposledy vložená hodnota). Například když do prázdného zásobníku vložíme postupně písmena `a`, `c`, `b`, `a`, bude v R_1 hodnota $\mathtt{a} \times 4^3 + \mathtt{c} \times 4^2 + \mathtt{b} \times 4 + \mathtt{a} = 1 \times 4^3 + 3 \times 4^2 + 2 \times 4 + 1 = 64 + 48 + 8 + 1 = 121$.

Jak ale s takovýmto registrem-zásobníkem pracovat? Vložit novou hodnotu x je jednoduché – pomocí registru R_2 vynásobíme obsah R_1 čtyřmi a potom ho x-krát zvětšíme o 1. Rovněž odebrání naposledy vložené hodnoty není těžké – je to přesně opačná operace. Vydělíme obsah registru R_1 čtyřmi. Zbytek po dělení je naposledy vložená hodnota, podíl (který dostaneme v R_2) je obsah zásobníku bez této hodnoty.

Nyní můžeme již přikročit k řešení zadané úlohy. Jednou možností je simulovat (pomocí dvou zásobníků) frontu, v níž si udržujeme ta písmena, jejichž pár jsme ještě neviděli. Toto řešení je poměrně komplikované a jeho základní myšlenka spočívá v tom, že přicházející písmena vkládáme do prvního zásobníku, písmena na kontrolu vybíráme z druhého zásobníku a vždy, když se nám druhý zásobník vyprázdni, do něj přesypeme obsah prvního zásobníku.

Ukážeme si raději jednodušší řešení. Budeme opět používat dva zásobníky. Do prvního budeme vkládat všechna přicházející malá písmena (jako hodnoty 1,2,3), do druhého velká (také jako hodnoty 1,2,3). Po dočtení vstupního slova jednoduše porovnáme obsahy obou zásobníků. Vstupní slovo bylo správné právě tehdy, je-li jejich obsah stejný. To již snadno ověříme.

```
var vstup: char;
    i,co: byte;
begin
   Read(vstup);
```

```
   while vstup<>'$' do begin
      if vstup>='a' then begin
         if vstup='a' then co:=1;
         if vstup='b' then co:=2;
         if vstup='c' then co:=3;
         while not Zero(R1) do begin Dec(R1); for i:=1 to 4 do Inc(R0); end;
         while not Zero(R0) do begin Dec(R0); Inc(R1); end;
         while co>0 do begin Inc(R1); co:=co-1; end;
      end else begin
         if vstup='A' then co:=1;
         if vstup='B' then co:=2;
         if vstup='C' then co:=3;
         while not Zero(R2) do begin Dec(R2); for i:=1 to 4 do Inc(R0); end;
         while not Zero(R0) do begin Dec(R0); Inc(R2); end;
         while co>0 do begin Inc(R2); co:=co-1; end;
      end;
      Read(vstup);
   end;
   while not Zero(R1) and not Zero(R2) do begin Dec(R1); Dec(R2); end;
   if Zero(R1) and Zero(R2) then Accept;
end.
```

b) Pro zvýšení přehlednosti označíme původní registry R_1, R_2, R_3 a nové registry Q_1, Q_2.

Použijeme myšlenku, kterou znáte již z řešení domácího kola. Zakódujeme obsah všech tří registrů do jediného, druhý registr budeme používat jako pomocný při práci s prvním. Místo tří registrů s obsahem a, b, c budeme mít tedy jeden registr Q_1 s obsahem $2^a 3^b 5^c$. Při simulování každé operace použijeme Q_2 jako pomocný registr. Na začátku i po provedení každé operace v něm bude uložena nula.

Operaci *Inc* (R_x) v původním programu nahradíme tím, že obsah nového registru Q_1 vynásobíme 2, 3, resp. 5. Podobně příkaz *Dec* (R_x) nahradíme příslušným dělením.

Nahradit podmínku *Zero* (R_x) bude trochu komplikovanější. Během vyhodnocování nějaké složené podmínky totiž nemůžeme provádět operace s registry – zjistit, zda je v R_x nula, tedy musíme *před* vyhodnocením příslušné podmínky. Navíc drobné problémy způsobí skutečnost, že tato podmínka se může vyskytovat i v podmínce pro příkaz `while`, kde bude vyhodnocována při každé iteraci (nejen před prvním voláním `while`), a že v jedné podmínce můžeme testovat více proměnných.

Definujme si „makro“ (kus výpočtu) *SpocitejZ*, které bude fungovat následovně: Pro každý registr R_x nastaví proměnnou z_x tak, aby v ní byla kladná hodnota právě tehdy, je-li v R_x nula, jinak bude $z_x = 0$. Výpočet makra začne tím, že obsah Q_1 vydělíme příslušným prvočíslem, přičemž si (v proměnné z_x) zapamatujeme zbytek, který jsme dostali při tomto dělení. Vrátíme obsah Q_1 do původního stavu. Jestliže obsah Q_1 byl dělitelný příslušným prvočíslem (tedy neplatí *Zero* (R_x)), bude v z_x nula, jinak tam bude kladný zbytek. Výraz *Zero*(R_x) má tedy v tomto okamžiku stejnou pravdivostní hodnotu jako výraz $(z_x > 0)$.

Každý příkaz „`if` P `then` *příkazy*`;`“ nahradíme makrem *SpocitejZ* a příkazem „`if` P' `then` *příkazy*`;`“, kde podmínka P' vznikla z P tak, že jsme v ní místo všech výskytů výrazu *Zero* (R_x) dali výraz $(z_x > 0)$.

Každý příkaz „`while` P `do` *příkazy*`;`“ nahradíme voláním makra *SpocitejZ* před cyklem a na konci každé iterace, tedy následujícím kusem výpočtu: „*SpocitejZ*`; while` P' `do begin` *příkazy*`;` *SpocitejZ*`; end;`“

Nová „makra“ *Inc*, *Dec* (jimiž nahradíme každý výskyt těchto příkazů v původním programu) a *SpocitejZ* (na simulaci *Zero*) budou tedy vypadat následovně:

```
var x,y,z1,z2,z3,i: byte; {nové proměnné, které nebyly v původním programu}
```

{ *Inc* (R_x) – x je v proměnné `x`, předpokládáme, že $Q_2 = 0$ }

```
  if x=1 then y:=2 else if x=2 then y:=3 else y:=5;
  {vynásobíme obsah Q1 čislem y}
  while not Zero(Q1) do begin
     Dec(Q1);
     for i:=1 to y do Inc(Q2);
  end;
  while not Zero(Q2) do begin
     Dec(Q2); Inc(Q1);
  end;
```

{ *Dec* (R_x) – x je v proměnné `x`, předpokládáme, že $Q_2 = 0$ }

```
  if x=1 then y:=2 else if x=2 then y:=3 else y:=5;
```

```
  z:=0;
  {vydělíme obsah Q1 číslem y}
  while not Zero(Q1) do begin
     Dec(Q1);
     if Zero(Q1) then begin z:=1; break; end; {v Rx byla nula}
     for i:=1 to y-1 do Dec(Q1);
     Inc(Q2);
  end;
  if z=1 then begin
     {obnovíme původní stav Q1 -- nic se nemění}
     while not Zero(Q2) do begin
        Dec(Q2); for i:=1 to y do Inc(Q1);
     end;
     Inc(Q1);
  end else begin
     {přesuneme do Q1 podíl}
     while not Zero(Q2) do begin
        Dec(Q2); Inc(Q1);
     end;
  end;
```

{ *SpocitejZ* – předpokládáme, že $Q_2 = 0$ }

```
  {nejdříve chceme spočítat z1, čili budeme dělit dvěma}
  y:=2;
  {vydělíme obsah Q1 číslem y}
  z1:=0;
  while not Zero(Q1) do begin
     Dec(Q1);
     z1:=z1+1;
     if z1=y then begin z1:=0; Inc(Q2); end;
  end;
  {vrátíme zpět původní hodnotu do Q1}
  while not Zero(Q2) do begin
     Dec(Q2); for i:=1 to y do Inc(Q1);
  end;
  for i:=1 to z1 do Inc(Q1);
  {a v proměnné z1 máme hledaný zbytek}
  {opakujeme totéž pro z2, y:=3 a z3, y:=5}
```

Dva důležité detaily, kterých jste si mohli povšimnout:

1. Nesmíme zapomenout ošetřit situaci, že registr R_i obsahuje nulu. V tom případě i po provedení *Dec* (R_i) musí v R_i (podle definice) zůstat nula.
2. Když jsme při simulování příkazů *Inc*, *Dec* a *Zero* potřebovali použít proměnné, muselo se jednat o *nové* proměnné, které se dosud v programu nevyskytovaly. (Co kdyby například původní program obsahoval část „`for i:=1 to 3 do` *Inc* (R_1)“ a my bychom při simulaci *Inc* (R_1) použili proměnnou `i`?) Volných jmen pro nové proměnné máme k dispozici nekonečně mnoho, lehce tedy najdeme nějaké nepoužité.

Snadno nahlédneme, že když tímto způsobem upravíme libovolný program, bude upravený program ekvivalentní s původním – tj. bude dávat pro každý vstup stejný výstup jako původní program. Přitom pokud původní program používal tři registry, upravený program už používá jen dva.

Uvědomte si, že aplikováním tohoto postupu na program používající $k > 3$ registrů dostaneme program používající $k - 1$ registrů. Proto platí poměrně překvapivý výsledek: K libovolné úloze, kterou dokážeme řešit na registrovém počítači, existuje program, jemuž na její řešení stačí dva registry.

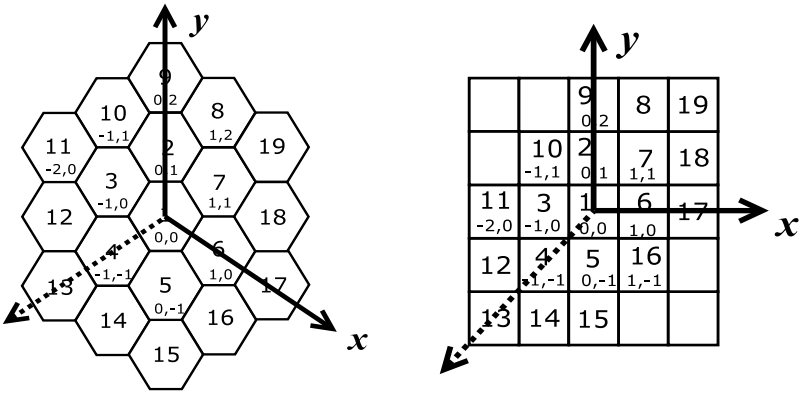
P-III-4 Psíci

Na poskakování psíků se můžeme dívat jako na hru. Stav hry lze jednoznačně popsat pozicí obou psíků. Skok psíků představuje tah. Když oba psíci skočí, změní stav hry. Povolené stavy hry budou ty, které odpovídají povoleným pozicím psíků. Pro každý stav hry budeme zkoumat, kolika tahy se do něho dá dostat z počátečního stavu (když jsou oba psíci na výchozích místech). Tento počet tahů budeme označovat jako vzdálenost daného stavu.

Vzdálenost počátečního stavu je 0. Všechny stavy, do nichž se lze z něho dostat jedním tahem, budou ve vzdálenosti 1. Nyní projdeme všechny stavy ve vzdálenosti 1 a hledáme, do kterých nových stavů se z nich dostaneme - ty budou zjevně ve vzdálenosti 2. Takto můžeme analogicky pokračovat pro stavy ve vzdálenosti 3, 4, ... Skončíme, když najdeme koncový stav (oba psíci jsou na svých cílových místech), nebo když už nenajdeme žádný nový stav. Tato technika prohledávání stavů se nazývá prohledávání do šířky.

Otázkou zůstává, jak pro každý stav určit, do kterých dalších (sousedních) stavů se z něho lze dostat jedním tahem. Pomohlo by nám, kdybychom uměli pro každé políčko na louce určit čísla jeho sousedů. Sousední stavy bychom potom určili snadno. Ze všech možných pohybů oběma psíky 6 směry (36 možností) vyškrtáme skákání stejným směrem, skákání na bodláky, skok některého psíka mimo louku a současný skok obou psíků na totéž políčko.

Abychom našli sousední políčka snadněji, ukážeme si, jak se dá šestiúhelníkový plán louky reprezentovat v obyčejném dvojrozměrném poli. Na políčku 1 si zvolíme dva směry. Jeden určuje rostoucí směr první souřadnice, druhý směr druhé souřadnice. Takto jsme přiřadili každému políčku souřadnice x, y, kterým odpovídají indexy v obyčejném dvojrozměrném poli. V dvojrozměrném poli je už nalezení sousedů lehké. Konkrétně při naší volbě souřadnicových os budou mít sousedi políčka (x, y) souřadnice $(x, y+1)$, $(x-1, y)$, $(x-1, y-1)$, $(x, y-1)$, $(x+1, y)$ a $(x+1, y+1)$.



Jak zjistíme pro políčko s číslem k jeho souřadnice? Všimněte si, že spirálu můžeme rozložit na vrstvy šestiúhelníkového tvaru. Nejprve určíme, na kolikáté vrstvě spirály se k nachází, potom stranu na této vrstvě, pozici políčka na straně a je to.

Nultá vrstva spirály obsahuje 1 políčko, i-tá vrstva pak $6i$, jelikož každá vrstva má 6 stran a na každé straně je i políček. Celkový počet políček ve spirálách $0 \ldots v$ je tedy $1 + \sum_{i=1}^{v} 6i = 3v^2 + 3v + 1$.

A jak zjistíme, kde se nachází políčko k? Nejdříve spočteme vrstvu – najdeme kladné řešení rovnice $k = 3v^2 + 3v + 1$ a zaokrouhlíme ho nahoru. Po vyřešení dostaneme $v = \left\lceil -1/2 + \sqrt{k/3 - 1/12} \right\rceil$. (Jednodušší a trochu pomalejší postup: zvyšujeme v, dokud počet políček nedosáhne k.)

Když už víme vrstvu v, kde se políčko nachází, pořadové číslo políčka ve vrstvě (číslováno od 0) dopočítáme snadno: odečteme od k celkový počet políček na dřívějších vrstvách a ještě 1. Na vrstvě v má každá strana v políček, rozdíl tedy stačí vydělit v. Číslo strany s bude podíl, pozice na straně p bude zbytek po tomto dělení.

Již umíme pro dané políčko k spočítat jeho vrstvu v, stranu s a pozici na straně p. Z těchto hodnot dostaneme souřadnice podle následující tabulky (platí pro $v \geq 1$):

s	x	y
0	$+v - 1 - p$	$+v$
1	$-1 - p$	$+v - 1 - p$
2	$-v$	$-1 - p$
3	$-v + 1 + p$	$-v$
4	$+1 + p$	$-v + 1 + p$
5	$+v$	$+1 + p$

Implementace

Stav hry je jednoznačně reprezentován souřadnicemi x_1, y_1, x_2, y_2 obou psíků. Při prohledávání do šířky si pamatujeme seznam stavů, které jsme již dosáhli, ale ještě jsme z nich nezkoušeli prohledávat nové vrcholy. K tomu nám poslouží fronta.

Dále potřebujeme umět pro každý stav rychle zjistit, zda jsme ho ještě neviděli, už viděli, nebo zda se do něj nedá jít (bodláky). K tomu používáme čtyřrozměrné pole, kde je to přímo zapsáno. (Přesně totéž se dalo reprezentovat dvojrozměrným polem, které bychom indexovali původními souřadnicemi. Navíc bychom ale potřebovali umět ze souřadnic určit původní číslo políčka.)

Abychom nemuseli při prohledávání do šířky stále kontrolovat, zda se nedostaneme ven z louky, postavíme okolo celé louky bodláky. Zakážeme také stavy, v nichž by byli oba psíci na stejném místě.

Prohledáváme prostor velikosti řádově $O(N^2)$, kde N je velikost louky. Časová i paměťová složitost prohledávání je lineární vzhledem k velikosti tohoto prostoru, tedy $O(N^2)$.

```
program Psici;
const
   EPS = 1.0E-6;   {max. chyba vzniklá v reálných číslech}
   MAX_V = 16;     {největší možná vrstva (i se zarážkou)}
   MAX_STAVU = MAX_V*MAX_V*MAX_V*MAX_V;
   INF = 299999;   {největší možný počet skoků}
   MAX_SMER = 6;   {počet směrů, jimiž se mohou psíci hýbat}

type
   TSour =  record x,y : integer; end; {souřadnice jednoho psíka}
   TStav = record p1, p2 : TSour; end; {souřadnice dvou psíků}
   {prostor pro 2 psíky: [x1,y1,x2,y2] říká, zda tam mohou být}
   TMrizka = array [-MAX_V..MAX_V,-MAX_V..MAX_V, -MAX_V..MAX_V,-MAX_V..MAX_V] of integer;

function dekVrstvu(k: integer): integer; {číslo vrstvy, kde se k nachází}
begin
   dekVrstvu:= trunc(0.5 + sqrt(k/3.0 - 1.0/12.0 - EPS) );
end;

function zakVrstvu(v: integer): integer; {poslední prvek na dané vrstvě}
begin
   zakVrstvu:= 3*v*v - 3*v +1;
end;

function dekoduj(k: integer): TSour; {Dekóduj číslo políčka na souřadnice}
var v, pv, s, ps: integer;
    sour: TSour;
begin
   if k=1 then begin  {pro k=1 (vrstva 0) naše vzorce nefungují}
      sour.x:= 0;
      sour.y:= 0;
   end else begin
      v:= dekVrstvu(k);
      pv:= k - (3*v*v - 3*v + 1); {pozice ve vrstvě}
      s:= pv div v; {strana šestiúhelníka, kde se pv nachází}
      ps:= pv mod v; {pozice na straně šestiúhelníka}

      case s of
      0: begin sour.x:= +v-1-ps; sour.y:= +v     ; end;
      1: begin sour.x:=   -1-ps; sour.y:= +v-1-ps; end;
      2: begin sour.x:= -v     ; sour.y:=   -1-ps; end;
      3: begin sour.x:= -v+1+ps; sour.y:= -v     ; end;
      4: begin sour.x:=   +1+ps; sour.y:= -v+1+ps; end;
      5: begin sour.x:= +v     ; sour.y:=   +1+ps; end;
      else writeln('Velká chyba v dekoduj');
      end;
   end;
   dekoduj:=sour;
end;

var
   n, m, s1, t1, s2, t2: integer; {vstup}
   v: integer; {max. použitá vrstva pro dané n}
   A: TMrizka; {prohledávaný prostor}
   F: array [0..MAX_STAVU] of TStav; {fronta}

{zakáže všechny situace, kde se nachází bodlák na daném místě}
procedure pridejBodlak(b: TSour);
var x, y: integer;
begin
   for x:= -v to v do for y:= -v to v do begin
      A[x, y, b.x, b.y]:= INF;        {2. psík stojí na bodláku}
      A[b.x, b.y, x, y]:= INF;        {1. psík stojí na bodláku}
   end;
end;

{vyčistí celý prohledávaný prostor}
procedure inicializace;
var x1, y1, x2, y2, i, last: integer;
begin
   {vyčištění prostoru}
   for x1:= -v to v do for y1:= -v to v do
      for x2:= -v to v do for y2:= -v to v do
         A[x1, y1, x2, y2]:= -1;
   {zarážky při okrajích - přidáme umělé bodláky}
   last:= zakVrstvu(v+1);
   for i:=n+1 to last do pridejBodlak(dekoduj(i));
   {zakážeme být oběma psíkům na stejném místě}
```

```
   for x1:= -v to v do for y1:= -v to v do A[x1, y1, x1, y1]:= INF;
end;

{posunutí souřadnice daným směrem}
function pohyb(a: TSour; s: integer): TSour;
const smer: array [1..MAX_SMER] of TSour = (
   (x: 0; y: 1), (x:-1; y: 0), (x:-1; y:-1),
   (x: 0; y:-1), (x: 1; y: 0), (x: 1; y: 1) );
begin
   a.x:= a.x+smer[s].x; a.y:= a.y+smer[s].y; pohyb:=a;
end;

function pracuj: integer;
{prohledávání prostoru, kde se mohou psíci nacházet}
var
   zac: integer;    {pozice prvního prvku ve frontě}
   kon: integer;    {pozice za posledním prvkem ve frontě = volné místo}
   i, j, vzd: integer;
   p1, p2, q1, q2: TSour;
   pt1, pt2: TSour; {dekódované pozice skrýší pro psíky}
begin
   zac:=0; kon:=1; {přidáme počátek do fronty}
   p1:= dekoduj(s1);
   p2:= dekoduj(s2);
   F[zac].p1:= p1; F[zac].p2:= p2;
   A[p1.x, p1.y, p2.x, p2.y]:= 0; {začínáme ve vzdálenosti 0}

   pt1:=dekoduj(t1); pt2:=dekoduj(t2);

   while zac<>kon do begin
      {vybereme stav z fronty a najdeme k němu vzdálenost}
      p1:=F[zac].p1; p2:=F[zac].p2;
      vzd:= A[p1.x, p1.y, p2.x, p2.y];
      inc(zac);

      {zkoušíme všechny kombinace směrů, kam mohou skákat}
      for i:=1 to MAX_SMER do for j:=1 to MAX_SMER do begin
         if i=j then continue; {nemohou skákat stejně}

         {nové pozice psíků}
         q1:= pohyb(p1, i); q2:= pohyb(p2, j);

         {již navštívená pozice resp. zarážka ?}
         if A[q1.x, q1.y, q2.x, q2.y] >= 0 then continue;

         {nově objevený stav -> přidáme do fronty}
         A[q1.x, q1.y, q2.x, q2.y]:= vzd+1;
         F[kon].p1:=q1; F[kon].p2:=q2;
         inc(kon);

         {našli jsme koncový stav?}
         if (pt1.x=q1.x) and (pt1.y=q1.y) and (pt2.x=q2.x) and (pt2.y=q2.y)
            then begin pracuj:=vzd+1; exit; end;
      end;
   end;
   pracuj:= -1;
end;

var x, i: integer;
begin
   while true do begin
      read(n, m);
      if (n=0) and (m=0) then break;

      v:= dekVrstvu(n);
      inicializace();

      read(s1, t1, s2, t2);
      for i:=1 to m do begin read(x); pridejBodlak(dekoduj(x)); end;
      if (s1=t1) and (s2=t2) then x:=0 {speciální případ} else x:=pracuj;
      if x>=0 then writeln(x) else writeln('nelze');
   end;
end.
```

P-III-5 AttoSoft

Označme $S = \sum_{i=1}^{N} l_i$ počet dní, které AttoSoft potřebuje na dokončení všech programů. Poslední program tedy dokončíme po S dnech.

Pro každý program spočítáme pokutu, kterou bychom za něj zaplatili, kdybychom ho dokončili až po S dnech. Program s nejmenší takovou pokutou zařadíme do rozvrhu jako poslední. Je-li to program číslo i, zbývá nám naplánovat všechny zbývající programy na prvních $S - l_i$ dní, což provedeme stejným způsobem (tzn. opět vybereme jako poslední program s nejnižší pokutou po $S - l_i$ dnech, atd.)

Správnost uvedeného algoritmu dokážeme indukcí vzhledem k počtu programů, které potřebujeme dokončit. Pokud je třeba dokončit jeden program, existuje jen jediný možný rozvrh, a náš algoritmus tedy funguje jistě správně.

Nechť tedy počet programů, které je třeba dokončit, je N a nechť pro libovolný menší počet programů náš algoritmus funguje správně. Označme G řešení získané naším algoritmem (v tomto řešení je posledním programem program číslo i). Nechť existuje jiné, levnější řešení O, které končí programem číslo j. Jestliže $i = j$, pak rozdíl mezi G a O musí být v pořadí prvních $N - 1$ programů. Podle indukčního předpokladu však toto pořadí v řešení G je optimální, proto řešení O nemůže být levnější.

V opačném případě vytvoříme nový rozvrh O' následujícím způsobem. Nechť rozvrh O dokončuje programy v pořadí $o_1, o_2, \ldots, o_N$ a nechť $o_k = i$. Podle rozvrhu O' dokončíme programy v následujícím pořadí: $o_1, o_2, \ldots, o_{k-1}, o_{k+1}, \ldots, o_N, i$. Všimněte si, že řešení O' je nejvýše tak drahé, jako řešení O. Pokuta za programy $o_{k+1}, \ldots, o_N$ je totiž nižší než v řešení O, neboť je dokončíme dříve (pokuta roste s počtem dní po termínu). Navíc pokuta za program i dokončený po S dnech určitě nepřesahuje pokutu za program o_N dokončený po S dnech, jelikož program i jsme vybrali tak, aby tato pokuta byla nejmenší možná.

Řešení O' ale nemůže být levnější než řešení G (platí tu stejný argument, jako v předchozím případě). Proto ani řešení O nemůže být levnější než G. Dokázali jsme, že žádné levnější řešení než G neexistuje, řešení určené naším algoritmem je tedy optimální.

V každém kroku algoritmu musíme spočítat příslušnou pokutu pro každý program, který jsme dosud nezařadili do rozvrhu. Proto časová složitost algoritmu je $O(N^2)$.

Při výpočtu si ještě musíme dávat pozor na to, že pokuta může být až $5\,000 \cdot 100\,000^3 + 5\,000 \cdot 100\,000^2 + \ldots + 5\,000$, tedy přibližně $5 \cdot 10^{18}$, a tak velké číslo se již nevejde do *longint*u a nemůžeme si dovolit použít ani typ *real*, jelikož potřebujeme i u tak velkých čísel rozlišovat rozdíly na řádu jednotek, Většina překladačů naštěstí nabízí 64-bitový celočíselný typ – v Turbo Pascalu je to typ *comp,* ve Free Pascalu například typ *QWord.*

```
program Attosoft;

const MAXN = 10000;

var a,b,c,d,l: array [1..MAXN] of longint;
    pouzite: array [1..MAXN] of boolean;
    N,S: longint;

function Cena(prog:integer; den: comp): comp;
begin
   cena:=((a[prog]*den+b[prog])*den+c[prog])*den+d[prog];
end; {function Cena}

procedure Nacti;
var i: integer;
begin
   readln(N);
   S:=0;
   for i:=1 to N do begin
      readln(l[i],a[i],b[i],c[i],d[i]);
      S:=S+l[i];
   end;
end; {procedure Nacti}

procedure Spocitej_rozvrh(d: longint);
var minprog: integer;
    min,cc: comp;
    i: integer;
begin
   {najdi nepoužitý program, který má nejnižší pokutu po d dnech}
   minprog:=-1;
   for i:=1 to N do begin
      if not pouzite[i] then begin
         cc:=Cena(i,d);
         if (minprog=-1) or (cc<min) then begin
            min:=cc;
            minprog:=i;
         end;
      end;
   end;

   if minprog>-1 then begin
      {označ program jako použitý}
      pouzite[minprog]:=true;
      {sestav zbytek rozvrhu}
      Spocitej_rozvrh(d-l[minprog]);
      {vypiš poslední program na konci rozvrhu}
      writeln(minprog);
   end;
end; {procedure Spocitej_rozvrh}

var i:integer;

begin
   Nacti;
   for i:=1 to N do pouzite[i]:=false;
   Spocitej_rozvrh(S);
end.
```