INFORMATIKA

Jak skáče žabka
(Úlohy z MO kategorie P, 34. část)

PAVEL TÖPFER

Matematicko-fyzikální fakulta UK, Praha

Tentokrát se blíže seznámíme s jednou soutěžní úlohou z aktuálního 65. ročníku Matematické olympiády kategorie P (školní rok 2015/16). Byla zadána jako praktická úloha domácího kola a ačkoliv za ni téměř každý z řešitelů získal nějaké body, ve většině případů to byly jenom 2 až 4 body z 10 možných, což odpovídá nějakému velmi primitivnímu a neefektivnímu řešení založenému na zkoušení všech možností hrubou silou. Z téměř stovky účastníků domácího kola olympiády vyřešilo tuto úlohu zcela správně na plný počet bodů pouze jedenáct.

Všechny soutěžní úlohy pro celý 65. ročník MO kategorie P připravili naši slovenští kolegové z Fakulty matematiky, fyziky a informatiky Univerzity Komenského v Bratislavě. Zadání úlohy si zde uvedeme v českém překladu jenom s mírnými textovými úpravami, řešení rozebereme trochu podrobněji a doplníme ho nově napsanými programovými ukázkami.

* * * * * * * * * * * *

Žabka Šandyna ráda skáče po kamenech v rybníku. V rybníku jich je celkem n a jsou očíslovány od 1 do n. Kameny jsou malé, takže si je představíme jako body. Kámen s číslem i leží na souřadnicích x_i, y_i. Šandyna dnes doskákala z kamene číslo 1 na kámen číslo n. Cestou mohla některé kameny (včetně kamenů 1 a n) navštívit i vícekrát. Žabka dokáže skočit libovolně daleko. Skákání ji ale unavuje, a tak každý její skok kromě prvního je vždy (ostře) kratší než skok bezprostředně předcházející.

Soutěžní úloha

Pro dané polohy kamenů spočítejte, kolik nejvýše skoků mohla žabka Šandyna provést během své cesty z kamene 1 na kámen n.

Formát vstupu

Na prvním řádku vstupu je zadán počet kamenů n. Na i-tém z následujících n řádků jsou uvedeny souřadnice kamene číslo i. Jediný řádek výstupu obsahuje jedno celé číslo – maximální možný počet skoků.

Omezení a hodnocení

Řešení bude testováno s deseti sadami vstupních dat. Za každý testovací vstup můžete získat 1 bod. V jednotlivých vstupních sadách je maximální hodnota n následující: 2, 3, 7, 18, 50, 100, 200, 1000, 2000, 3000.

Všechny souřadnice jsou z rozsahu od 0 do 10^9 včetně. V prvních pěti testovacích vstupech dokonce žádná souřadnice nepřekročí hodnotu 10^4.

Příklad

Vstup:	Výstup:
6	7
0 1	
5 0	
8 3	
3 3	
3 5	
3 2	

Vysvětlení: Jedna optimální posloupnost sedmi skoků vedoucí z kamene číslo 1 na kámen číslo 6 vypadá takto: $1 \to 3 \to 5 \to 1 \to 6 \to 5 \to 4 \to 6$. Tyto skoky mají postupně délky: $2\sqrt{17} > \sqrt{29} > 5 > \sqrt{10} > 3 > 2 > 1$.

* * * * * * * * * * * *

Nejjednodušší řešení je založeno na prostém zkoušení všech možných cest z kamene číslo 1 na kámen číslo n, jak by žabka mohla skákat. Takové řešení velmi snadno naprogramujeme pomocí rekurze. Rekurzivní procedura dostane ve vstupních parametrech číslo kamene, kde právě stojí žabka, počet již uskutečněných skoků od startu a délku posledního provedeného skoku. Procedura v cyklu postupně zkusí provést skok na každý z ostatních kamenů a je-li takový skok možný (tzn. je-li kratší než dosud

poslední provedený skok), procedura se rekurzívně zavolá na cílový kámen tohoto skoku. Kdykoliv se žabka ocitne na cílovém kameni číslo n, může zlepšit (tj. zvýšit) dosud maximální nalezený počet skoků z kamene 1 na kámen n. I po dosažení kamene číslo n ale bude ještě zkoušet skákat dál.

```
program Zabka1;

const MaxKamenu = 3000;                {maximální počet kamenů}
var N: integer;                        {skutečný počet kamenů}
    Kameny: array[1..MaxKamenu] of
      record x, y: longint end;        {souřadnice kamenů}
    Max: integer;                      {výsledný max. počet skoků}
    i: integer;

function Vzdal(a, b: integer): int64;
{počítá kvadrát vzdálenosti kamenů číslo a, b}
var dx, dy: longint;
begin
dx:=Kameny[a].x - Kameny[b].x;
dy:=Kameny[a].y - Kameny[b].y;
Vzdal:=dx*dx + dy*dy
end; {Vzdal}

procedure Cesta(Kde: integer; Skoky: longint;
                Naposled: int64);
{parametry: kde žabka stojí, kolik skoků dosud udělala, kvadrát
délky posledního skoku; procedura používá globální proměnné N
a Max, výsledný maximální počet skoků ze startu do cíle bude
v proměnné Max}
var i: integer;
begin
if (Kde = N) and (Skoky > Max) then Max:= Skoky;
                                       {žabka je v cíli}
for i:=1 to N do                  {žabka zkusí skočit na i-tý kámen}
  if (i <> Kde) and (Vzdal(i, Kde) < Naposled) then
                                       {skok na i-tý kámen je možný}
    Cesta(i, Skoky+1, Vzdal(i, Kde));
end; {Cesta}

begin
read(N);
for i:=1 to N do
  read(Kameny[i].x, Kameny[i].y);
Max:=0;
Cesta(1, 0, 99999999999999999);
writeln(Max)
end.
```

V ukázkovém programu si všimněte, že místo délky skoku počítáme všude s kvadrátem délky skoku. Díky tomuto technickému triku se vyhneme výpočtu druhé odmocniny a celý výpočet tak můžeme provádět v celočíselné aritmetice. Ta je rychlejší než aritmetika reálná a hlavně v ní nehrozí žádné riziko vzniku zaokrouhlovacích chyb.

Výše uvedený postup „hrubou silou“ je sice teoreticky správný, ale je velmi neefektivní, neboť se při něm mnohokrát opakovaně zkouší stejné posloupnosti skoků. Takové řešení bude proto použitelné jen pro velmi malé hodnoty n. Vzhledem k odstupňované velikosti testovacích vstupních dat (viz zadání úlohy) jste ovšem i s takto primitivním řešením mohli v soutěži získat tři až čtyři body z celkových deseti možných.

Zásadního zlepšení dosáhneme, když se dobrou organizací práce vyhneme opakovanému provádění toho, co jsme už jednou udělali. To je základní myšlenka programovací techniky zvané dynamické programování. Časová složitost algoritmu se tak rázem sníží z exponenciální na polynomiální (v našem konkrétním případě na kubickou) a program začne zvládat zpracovat dostatečně rychle i mnohem rozsáhlejší vstupní data.

Představte si, že žabka už provedla několik skoků a zajímá nás, jak nejlépe může z této situace pokračovat dále do cíle. Odpověď na tuto otázku vůbec nezávisí na tom, kudy žabka dosud skákala, závisí pouze na jejím zatím posledním skoku. Poslední skok totiž přesně určuje, kde se nyní žabka nachází a jak nejvýše dlouhý skok může následně provést.

Při celkovém počtu n kamenů existuje $n \cdot (n-1)$ možností, odkud kam mohl vést zatím poslední skok žabky. Pro každou z těchto možností chceme určit, kolik nejvýše skoků ještě může žabka udělat cestou do cíle. Každou z těchto $\mathrm{O}(n^2)$ možností dokážeme vyřešit tak, že postupně vyzkoušíme $\mathrm{O}(n)$ variant, kam žabka skočí následujícím skokem. Pro každou z těchto variant tak dostaneme jednu novou otázku téhož typu, k jejímuž zodpovězení použijeme rekurzivní volání.

Kdybychom přímo implementovali právě popsaný postup, nedostali bychom nic jiného, než dříve uvedené řešení hrubou silou. Můžeme zde ale využít programovací techniku, které se říká „chytrá rekurze“. Jakmile vyřešíme některou z uvažovaných možností skoku, nalezenou odpověď nejen bezprostředně použijeme, ale také si ji zaznamenáme do tabulky. Pokud se někdy později při výpočtu programu objeví stejná otázka, jednoduše vrátíme uloženou hodnotu z tabulky – tedy nebudeme už provádět žádné zkoušení možností ani žádné rekurzivní volání.

Takto upravený program bude řešit každou z uvažovaných $O(n^2)$ možností skoku pouze jednou a vyřeší ji v čase nejvýše $O(n)$. Celkově tedy program vykoná $O(n^3)$ kroků výpočtu, po nichž budeme mít zaplněnou celou uvažovanou tabulku. Nakonec v tabulce projdeme všechny skoky vycházející z kamene číslo 1 a z jejich hodnot určíme maximum. Tím zjistíme, jaký maximální počet skoků žabky může následovat po jejím prvním skoku z kamene číslo 1. Výslednou hodnotu získáme tak, že toto maximum zvýšíme o 1, tzn. započítáme ještě první skok.

```
program Zabka2;

const MaxKamenu = 3000;              {maximální počet kamenů}
var N: integer;                      {skutečný počet kamenů}
    Kameny: array[1..MaxKamenu] of
      record x, y: longint end;      {souřadnice kamenů}
    Max: integer;                    {výsledný max. počet skoků}
    T: array[1..MaxKamenu, 1..MaxKamenu] of longint;
    {T[a,b]= max. počet skoků do cíle
     následujících po skoku a->b,
     hodnota -1 = cesta neexistuje,
     hodnota -2 = zatím neznáme}
    i, j: integer;

function Vzdal(a, b: integer): int64;
{kvadrát vzdálenosti kamenů číslo a, b}
var dx, dy: longint;
begin
dx:=Kameny[a].x - Kameny[b].x;
dy:=Kameny[a].y - Kameny[b].y;
Vzdal:=dx*dx + dy*dy
end; {Vzdal}

function Skok(a, b: integer): longint;
{počítá, kolik nejvýše skoků může ještě následovat po skoku
mezi kameny a->b cestou na cílový kámen N;
pokud taková cesta neexistuje, funkce vrací -1;
funkce používá globální proměnné N a T}
var Naposled: int64;  {kvadrát délky posledního skoku a->b}
    DoCile: longint;  {maximální počet navazujících
                       skoků do cíle}
    i: integer;
begin
Naposled:= Vzdal(a, b);
DoCile:= -1;
for i:=1 to N do          {žabka zkusí skočit dále z kamene b}
  if (i <> b) and (Vzdal(i, b) < Naposled) then
                          {navazující skok b->i}
```

```
    begin
    if T[b, i] = -2 then T[b, i]:= Skok(b, i);
    if T[b, i] > DoCile then DoCile:= T[b, i]
    end;
if DoCile >= 0 then {existuje navazující cesta z b až do cíle}
  Skok:=DoCile + 1
else if b = N then  {jsme v cíli, nelze pokračovat až do cíle}
  Skok:= 0
else              {nejsme v cíli a nelze pokračovat až do cíle}
  Skok:=-1
end; {Skok}

begin
read(N);
for i:=1 to N do read(Kameny[i].x, Kameny[i].y);
Max:= 0;
for i:=1 to N do
  for j:=1 to N do
    T[i, j]:= -2;           {příznak, že hodnotu ještě neznáme}
for i:=2 to N do
  begin
  if T[1, i] = -2 then T[1, i]:= Skok(1, i);
  if T[1, i] > Max then Max:= T[1, i];
  end;
writeln(Max + 1)
end.
```

Předchozí řešení úlohy můžeme ještě více vylepšit. Ušetřit se dá na tom, že při výpočtu každé hodnoty v tabulce procházíme vždy všechny navazující skoky, a to včetně těch, které už v dané situaci nemůžeme provést, neboť jsou delší. Nabízí se proto možnost uspořádat všechny možné skoky z kamene na kámen sestupně podle jejich délky. (V programu opět použijeme raději druhou mocninu délky, jelikož tu si můžeme uložit do celočíselné proměnné.) Když pak budeme zpracovávat skoky v tomto pořadí, budeme mít jistotu, že každý právě zpracovávaný skok má kratší délku, než všechny předchozí již zpracované skoky, takže na ně může navázat při cestě žabky ze startu do cíle.

Uvedená úvaha by takto přesně fungovala pouze v případě, že by délky všech skoků byly navzájem různé. Některé skoky ale mohou mít stejnou délku a ty budeme muset zpracovat vždy všechny najednou, neboť žabka nemůže provést dva stejně dlouhé skoky po sobě. Všechny možné skoky z jednoho kamene na druhý si proto předem roztřídíme do skupin podle jejich délky a až takto vzniklé skupiny skoků uspořádáme podle délky skoku, počínaje od největší délky.

Pro každý kámen si budeme pamatovat, jakým nejvyšším počtem skoků se na něj dokážeme dostat z kamene číslo 1. Na začátku je to 0 pro startovní kámen číslo 1 a „nekonečno“ pro každý jiný kámen (tam zatím neznáme žádnou cestu). Postupně budeme zkoušet použít skoky v pořadí od nejdelších po nejkratší a vždy přepočítáme jenom ty údaje, které se týkají právě zpracovávaného skoku. Když zpracováváme skok z kamene A na kámen B, podíváme se, kolika nejvýše skoky (jedná se zcela jistě o delší skoky) jsme se dokázali dostat ze startu na kámen A. Jestliže je to K skoků, pak se nyní umíme dostat na kámen B pomocí $K + 1$ skoků. Pokud byla dosud uložená hodnota pro kámen B menší, zvýšíme ji na $K + 1$.

Časová složitost posledního popsaného řešení je $O(n^2 \log n)$. Jeho nejpomalejší částí je uspořádání všech $O(n^2)$ různých existujících skoků mezi kameny podle délky. Samotné zpracování skoků je pak už rychlejší, každý z $O(n^2)$ skoků zpracujeme v konstantním čase.

```
program Zabka3;

const MaxKamenu = 3000;                {maximální počet kamenů}
type Skok = record
       odkud, kam: integer; {čísla kamenů, odkud kam se skáče}
       delka: int64;        {kvadrát délky skoku odkud -> kam}
     end;
     PoleSkoku = array[1..MaxKamenu*MaxKamenu] of Skok;

var N: integer;                        {skutečný počet kamenů}
    Kameny: array[1..MaxKamenu] of
      record x, y: longint end;        {souřadnice kamenů}
    Skoky: PoleSkoku;
    M_stare, M_nove: array[1..MaxKamenu] of longint;
      {maximální počet skoků ze startu na tento kámen;
       staré a nové hodnoty řeší více skoků téže délky}
    i, j: integer;
    s: longint;

function Vzdal(a, b: integer): int64;
{kvadrát vzdálenosti kamenů číslo a, b}
var dx, dy: longint;
begin
dx:=Kameny[a].x - Kameny[b].x;
dy:=Kameny[a].y - Kameny[b].y;
Vzdal:=dx*dx + dy*dy
end; {Vzdal}

procedure SeraditSestupne(var P: PoleSkoku; N: longint);
{libovolný třídicí algoritmus s časovou složitostí O(N.logN);
```

```
my zde pro jednoduchost použijeme jenom bublinkové třídění}
var i, j: longint;
    x: Skok;
begin
for i:=N-1 downto 1 do
  for j:=1 to i do
    if P[j].delka < P[j+1].delka then
      begin x:=P[j]; P[j]:=P[j+1]; P[j+1]:=x end
end; {SeraditSestupne}

begin
{Načtení vstupu a inicializace polí M:}
read(N);
for i:=1 to N do
  begin
  read(Kameny[i].x, Kameny[i].y);
  M_stare[i]:=-1; {nekonečno}
  end;
M_stare[1]:=0;     {na startu jsme na nula skoků}
M_nove:=M_stare;

{Vytvoření seznamu všech N*(N-1) možných skoků z kamene na kámen,
řazeno sestupně podle délky skoku:}
s:=0;
for i:=1 to N do
  for j:=1 to N do
    if j <> i then
      begin  {skok i->j}
      s:=s + 1;
      Skoky[s].odkud:=i;
      Skoky[s].kam:=j;
      Skoky[s].delka:=Vzdal(i, j);
      end;
SeraditSestupne(Skoky, s);

{Výpočet hodnot M:}
for s:=1 to N*(N-1) do   {zpracujme skok Skoky[s]}
  begin
  if (s > 1) and (Skoky[s].delka < Skoky[s-1].delka) then
    M_stare:=M_nove;      {začínají skoky kratší délky}
  if (M_stare[Skoky[s].odkud] >= 0) and
     (M_nove[Skoky[s].kam] < M_stare[Skoky[s].odkud] + 1) then
     M_nove[Skoky[s].kam] := M_stare[Skoky[s].odkud] + 1;
  end;

writeln(M_nove[N])
end.
```