## A-II-1 Vzostup a pád Kocúrkova

V tejto súťažnej úlohe sme k danému poľu $P[1..n]$ mali nájsť trojicu indexov $a < b < c$ takú, že platí $P[c] < P[a] < P[b]$. Existuje veľa rôznych spôsobov, ako túto úlohu riešiť. Najpriamočiarejším, ale najmenej efektívnym, je vyskúšanie všetkých možností. Takéto riešenie má časovú zložitosť $\Theta(n^3)$.

Šikovnejšie riešenie môžeme začať nejakými pozorovaniami. Začnime napríklad tým, že si zvolíme index $b$. Teraz chceme naľavo od neho zvoliť index $a$ a napravo od neho zase index $c$. Aj $P[a]$ aj $P[c]$ má byť menšie ako $P[b]$. Do úvahy budú teda pripadať len niektoré možnosti pre $a$ a $c$.

Navyše ešte musíme zabezpečiť, aby bolo $P[a] > P[c]$. Ako na to? Jednoducho: spomedzi všetkých prípustných $a$ vyberieme to, pre ktoré je $P[a]$ najväčšie, a spomedzi všetkých $c$ vyberieme to, pre ktoré je $P[c]$ najmenšie. Ak táto dvojica $(a, c)$ vyhovuje, našli sme riešenie, a ak nie, tak vieme, že pre nami zvolené $b$ žiadne riešenie neexistuje.

Vyššie popísané riešenie vieme ľahko implementovať s časovou zložitosťou $\Theta(n^2)$: postupne vyskúšame všetky možnosti pre $b$ a pre každú z nich v lineárnom čase nájdeme optimálne voľby $a$ a $c$.

Iný možný postup, tiež vedúci k riešeniu s kvadratickou časovou zložitosťou: Tentokrát začneme tým, že si zvolíme $a$. Ako je optimálne zvoliť $b$? Môžeme použiť hocijaké $b$, pre ktoré $P[a] < P[b]$, ale potom za ním potrebujeme ešte nájsť $c$, pre ktoré $P[a] > P[c]$. Najlepšie, čo môžeme spraviť, je spomedzi všetkých $b$, ktoré spĺňajú $P[a] < P[b]$, zobrať hneď to prvé, teda najmenšie. To preto, že oproti iným voľbám $b$ nám ostane najdlhší možný úsek, z ktorého môžeme následne vyberať $c$.

Celý algoritmus môžeme teda sformulovať takto: Pre každé možné $a$ sprav nasledovné: Najskôr choď v poli doprava, kým nenájdeš prvý index $b$ taký, že $P[a] < P[b]$. Potom choď od $b$ ďalej doprava, kým nenájdeš index $c$ taký, že $P[a] > P[c]$. (Samozrejme, občas vyhovujúce $b$ alebo $c$ nenájdeme, čo znamená, že pre dotyčné $a$ riešenie neexistuje.)

Vyššie popísané riešenia vieme aj ďalej optimalizovať. Opäť existuje veľa rôznych spôsobov. Niektoré z nich vedú k riešeniam s časovou zložitosťou $O(n \log n)$, iné dokonca k riešeniam s lineárnou časovou zložitosťou. Jedno takéto riešenie si ukážeme. Vznikne tak, že vylepšíme druhé z vyššie uvedených riešení bežiacich v kvadratickom čase.

V tomto riešení potrebujeme vedieť zefektívniť dve operácie:

1. Keď si zvolíme index $a$, potrebujeme rýchlo nájsť index $b$ najbližšieho väčšieho prvku napravo od $a$.
2. Keď už máme index $b$, potrebujeme rýchlo nájsť vyhovujúci index $c$. Toto vieme spraviť tak, že rýchlo nájdeme index najmenšieho spomedzi prvkov napravo od $b$. Totiž ak nevyhovuje ten, tak ani žiaden iný.

Začneme druhou operáciou. Tú si vieme veľmi ľahko v lineárnom čase predpočítať. Postupne pre každé $k$ od 1 do $n$ nájdeme minimum posledných $k$ prvkov poľa (presnejšie, jeho index) a všetky tieto minimá si uložíme do poľa. Keď potom budeme mať konkrétny index $b$, stačí sa pozrieť do tohto poľa na správne políčko a tam si nájsť jemu zodpovedajúce najlepšie $c$.

Prvú operáciu tiež vyriešime tak, že si všetky potrebné odpovede predpočítame. Chceme teda k úplne každému prvku nájsť najbližší väčší napravo od neho. Ako takéto niečo spraviť šikovne? Prvky budeme postupne spracúvať zľava doprava, pričom si budeme pamätať množinu všetkých prvkov, od ktorých sme zatiaľ nevideli väčší. Predstavme si, že si takto napr. pamätáme prvky $(10, 20, 30, 40)$. Čo sa teraz stane, ak spracujeme ďalší prvok a ten má hodnotu 25? K prvkom s hodnotou 10 a 20 si môžeme poznačiť, že práve toto je ten prvok, na ktorý čakali – najbližší, ktorý je od nich väčší. Prvky 30 a 40 čakajú ďalej. No a pribudne k nim aj prvok, ktorý sme práve spracovali. Aktuálne pamätaná množina prvkov sa teda zmenila na $(25, 30, 40)$.

Ako si šikovne pamätať aktuálnu množinu prvkov? Stačí na to obyčajný zásobník, v ktorom budú aktuálne prvky uložené usporiadané, a to tak, že najmenší je na vrchu. Spracovanie nového prvku $p$ potom vyzerá nasledovne:

- Kým je na vrchu zásobníka prvok menší ako $p$, vyber ho a zaznač si, že pre neho je najbližším väčším prvkom práve spracúvaný prvok $p$.

- Vlož $p$ na vrch zásobníka. Keďže v zásobníku už boli len prvky väčšie alebo rovné $p$, aj po vložení $p$ je zásobník usporiadaný.

Aj keď sa to možno na prvý pohľad nezdá, takéto spracovanie poľa má lineárnu časovú zložitosť. Áno, môže sa nám stať, že niektoré konkrétne prvky budeme spracúvať dlho. Pozrime sa však na beh nášho programu šikovnejším spôsobom: všetko, čo počas neho robíme, je, že nejaké prvky vkladáme do zásobníka a nejaké prvky z neho vyberáme. No a keďže každý prvok vložíme práve raz, je vložení presne $n$, a teda vybratí dokopy tiež nemôže byť viac ako $n$.

Zhrňme si teda ešte raz celú myšlienku nášho vzorového riešenia. V prvej fáze prejdeme celé pole zľava doprava a pomocou zásobníka si ku každému prvku nájdeme najbližší väčší napravo od neho. V druhej fáze prejdeme pole sprava doľava, pričom si ku každému sufixu poľa zapamätáme, kde má minimum. V tretej fáze potom vyskúšame všetky možnosti pre $a$. Ku každému $a$ pomocou predpočítaných informácií nájdeme najlepšie $b$ a $c$ a otestujeme, či spĺňajú podmienky zo zadania. Každá fáza má lineárnu časovú zložitosť, a teda celé riešenie beží v čase $O(n)$. V programe predpokladáme, že $n$ je aspoň 3.

### Listing programu (Python)

```
N = int( input() )
P = [ int(x) for x in input().split() ]

# prvá fáza: ku každému prvku nájdi najbližší väčší
bestB = [ None for n in range(N) ]
aktivne = []
for n in range(N):
    while len(aktivne) > 0 and P[ aktivne[-1] ] < P[n]:
        bestB[ aktivne.pop() ] = n
    aktivne.append(n)

# druhá fáza: ku každému indexu n nájdi index minima v úseku napravo od n
bestC = [ None for n in range(N) ]
bestC[N-2] = N-1
for n in reversed(range(N-2)):
    bestC[n] = bestC[n+1]
    if P[n+1] < P[bestC[n]]: bestC[n] = n+1

# tretia fáza: ku každému a nájdi najlepšie b, c
for a in range(N):
    b = bestB[a]
    if b is None: continue
    c = bestC[b]
    if c is None: continue
    if P[c] < P[a] < P[b]:
        print( a+1, b+1, c+1 )
        import sys
        sys.exit()
print('neexistuje')
```

## A-II-2 Bicykle

Túto súťažnú úlohu vieme pomerne priamočiaro vyriešiť pomocou Dijkstrovho algoritmu na nájdenie najkratšej cesty v grafe. Grafom však nebude pôvodný graf zo zadania úlohy. Keď si kladieme otázku „ako sa najrýchlejšie dostanem z bodu A do bodu B“, záleží odpoveď aj na tom, či práve pri sebe máme bicykel.

My si preto postavíme z pôvodného grafu nový graf s $2n$ vrcholmi: ku každému vrcholu $v$ v pôvodnom grafe budeme mať vrcholy $(v, 0)$ a $(v, 1)$. Prvý z nich predstavuje situáciu „som vo $v$ bez bicykla“ a druhý situáciu „som vo $v$ a mám bicykel“.

Ak som vo vrchole $v$, v ktorom sa nachádza požičovňa, sú vrcholy $(v, 0)$ a $(v, 1)$ prepojené hranou dĺžky 0. V nulovom čase tu totiž viem bicykel vrátiť alebo naopak získať. V ostatných prípadoch tieto dva vrcholy nie sú priamo spojené hranou – ak mám bicykel, musím ísť ďalej na ňom, ak nie, musím ísť ďalej bez neho.

Ak v pôvodnom grafe máme hranu medzi vrcholmi $u$ a $v$, v novom grafe budeme mať hranu medzi $(u, 0)$ a $(v, 0)$ a tiež hranu medzi $(u, 1)$ a $(v, 1)$. Prvej dĺžka zodpovedá času pešieho presunu, druhej dĺžka zase presunu na bicykli. No a to už je všetko. V takto postavenom grafe jednoducho nájdeme najkratšiu cestu z vrcholu $(1, 0)$ do vrcholu $(n, 0)$.

Detailný popis Dijkstrovho algoritmu nájdete napríklad vo vzorovom riešení úlohy A-III-4 z 26. ročníka OI.

V programe používame implementáciu Dijkstrovho algoritmu pomocou prioritnej fronty, na jej uloženie používame `set` v ktorom si udržiavame záznamy tvaru $(d(v), v)$, kde $d(v)$ je najmenšia zatiaľ známa vzdialenosť do vrcholu $v$.

Takáto implementácia má časovú zložitosť $\Theta(m \log n)$. Pre jednoduchosť vrcholy číslujeme od 0 a namiesto vrcholov $(v, 0)$ a $(v, 1)$ máme vrcholy $2v$ a $2v + 1$.

### Listing programu (C++)

```
#include <bits/stdc++.h>
using namespace std;

struct edge { int to, length; };

int main() {
    // načítame základné parametre
    int N, M, P;
    cin >> N >> M >> P;
    vector< vector<edge> > graph(2*N);

    // načítame polohy požičovní a pridáme im zodpovedajúce hrany
    while (P--) {
        int x;
        cin >> x;
        --x;
        graph[2*x  ].push_back( {2*x+1,0} );
        graph[2*x+1].push_back( {2*x  ,0} );
    }

    // načítame ulice a pridáme im zodpovedajúce hrany
    while (M--) {
        int x, y, tp, tb;
        cin >> x >> y >> tp >> tb;
        --x; --y;
        graph[2*x  ].push_back( {2*y  ,tp} );
        graph[2*y  ].push_back( {2*x  ,tp} );
        graph[2*x+1].push_back( {2*y+1,tb} );
        graph[2*y+1].push_back( {2*x+1,tb} );
    }

    // pomocou Dijkstrovho algoritmu nájdeme najkratšiu cestu
    int source = 0, target = 2*(N-1);
    vector<int> min_distance( 2*N, INT_MAX );
    min_distance[ source ] = 0;
    set< pair<int,int> > active_vertices;
    active_vertices.insert( {0,source} );

    while (!active_vertices.empty()) {
        int where = active_vertices.begin()->second;
        if (where == target) { // dostali sme sa do cieľa, vypíšeme odpoveď a skončíme
            cout << min_distance[where] << endl;
            return 0;
        }
        active_vertices.erase( active_vertices.begin() );
        for (auto ed : graph[where])
            if (min_distance[ed.to] > min_distance[where] + ed.length) {
                active_vertices.erase( { min_distance[ed.to], ed.to } );
                min_distance[ed.to] = min_distance[where] + ed.length;
                active_vertices.insert( { min_distance[ed.to], ed.to } );
            }
    }
}
```

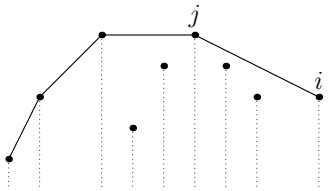
## A-II-3 Wi-Fi

Úlohu zjavne stačí vedieť riešiť v jednom smere, napr. pre každý dom nájsť najľavejší iný, s ktorým vie komunikovať. Akonáhle máme takéto riešenie, môžeme ho použiť dvakrát (raz na pôvodný vstup a raz na jeho reverz) a následne si už len pre každý dom vybrať lepšiu z oboch vypočítaných možností.

Antény sa nachádzajú v bodoch $(x_1, v_1)$, ..., $(x_n, v_n)$, pričom $x_1 < \cdots < x_n$. Ako nájsť pre $i$-tu anténu najľavejšiu inú, s ktorou sa priamo vidia?

Predstavme si, že sme našli horný konvexný obal bodov $(x_1, y_1)$ až $(x_i, y_i)$ – inými slovami, takú lomenú čiaru, ktorá začína v $(x_1, y_1)$, prechádza cez niektoré iné z našich bodov (a len v nich smie meniť smer) a končí

v $(x_i, y_i)$, pričom platí, že všetky body ležia buď na nej alebo pod ňou. Potom tvrdíme, že hľadaným bodom je predposledný bod $j$, ktorý leží na tomto konvexnom obale.



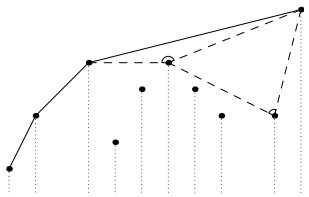

Horný konvexný obal množiny bodov.

Prečo je to tak? V prvom rade, keďže body $i$ a $j$ ležia na konvexnom obale, ich spojnica neprechádza žiadnym domom, a teda sa tieto dva body naozaj priamo vidia. No a žiaden bod naľavo od bodu $j$ už nemôže na bod $i$ vidieť. Sporom: keby pre nejaký bod $k < j$ prechádzala spojnica bodov $k$ a $i$ ponad bod $j$, znamenalo by to, že bod $j$ na hornom konvexnom obale vôbec neleží.

Keby sme teda pre každý bod $i$ vedeli, ako vyzerá horný konvexný obal bodov 1 až $i$, mali by sme našu súťažnú úlohu vyriešenú. Toto ale naozaj vieme spraviť, a dokonca v lineárnom čase. My totiž vieme ľahko zobrať horný konvexný obal bodov 1 až $i$ a prerobiť ho na konvexný obal bodov 1 až $i + 1$.

Princíp je jednoduchý. Do našej množiny bodov pridáme bod $i + 1$, ktorý leží viac napravo ako všetky doteraz spracované body. Z toho je jasné, že tento bod bude ležať na novom konvexnom obale. Všetky body, ktoré ležali pod pôvodným konvexným obalom budú zjavne ležať aj pod tým novým, na tie už teda môžeme spokojne zabudnúť. Jediné, čo sa zmení, je že niektoré body z konvexného obalu ubudnú, keďže po pridaní bodu $i + 1$ sa ocitli pod spojnicou tohto bodu s nejakým iným. Nie je ťažké nahliadnuť, že tieto zmeny sa vždy udejú len na pravom konci konvexného obalu – niekoľko posledných bodov z neho ubudne.

Exaktne môžeme tento algoritmus sformulovať nasledovne: predstavme si, že sme len zobrali doterajší konvexný obal a na jeho koniec sme pridali úsečku z $i$ do $i + 1$. Správny konvexný obal vyzerá tak, že keď po ňom ideme zľava doprava, tak v každom jeho vrchole zatočíme doprava. Jediné miesto, kde toto nemusí byť pravda, je práve v bode $i$. Toto miesto teda skontrolujeme, a ak zistíme, že naša lomená čiara zatáča do nesprávnej strany, bod $i$ z nej vyhodíme. Túto úvahu následne opakujeme s ďalšími bodmi, až kým nedostaneme správny nový konvexný obal.



Výpočet nového horného konvexného obalu množiny bodov.

Na obrázku je príklad toho, ako to celé môže vyzerať. Oproti prvému obrázku nám napravo pribudol nový bod, ktorý spôsobí, že zo starého obalu postupne vyhodíme jeho dva najpravejšie body. Čiarkované čiary ukazujú, kadiaľ viedla postupne upravovaná lomená čiara počas výroby nového konvexného obalu. Všimnite si, že oba vyznačené uhly sú menšie ako 180°. (V programe toto vieme ľahko kontrolovať pomocou vhodného vektorového súčinu.)

Body, ktoré ležia na aktuálnom hornom konvexnom obale, si budeme udržiavať v zásobníku s tým, že na vrchu bude najpravejší z nich. Takto vieme vždy pri pridávaní nového bodu v konštantnom čase vyhodnotiť, či ešte treba nejaký bod vyhodiť, a ak áno, tak to spraviť.

Prechod celou množinou bodov a postupné zostrojovanie všetkých čiastočných konvexných obalov má dokopy lineárnu časovú zložitosť. To preto, že každý bod na obal (do zásobníka) raz pridáme a každý bod odtiaľ nanajvýš raz odstránime. V programe predpokladáme, že $n$ je aspoň 2.

### Listing programu (Python)

```
def zataca_dolava(P,Q,R):
    ux, uy, vx, vy = Q[0]-P[0], Q[1]-P[1], R[0]-Q[0], R[1]-Q[1]
    return ux*vy - vx*uy > 0
```

```
def kolko_dolava(body):
    # na vstupe dostaneme súradnice bodov
    # na výstupe pre každé i vrátime, koľko najďalej doľava dovidí bod i

    # spracujeme prvé dva body
    odpoved = [ 0, body[1][0] - body[0][0] ]
    obal = [ 0, 1 ]

    # postupne pridávame ďalšie body
    for n in range( 2, len(body) ):
        while len(obal) >= 2 and zataca_dolava( body[ obal[-2] ], body[ obal[-1] ], body[n] ):
            obal.pop()
        odpoved.append( body[n][0] - body[ obal[-1] ][0] )
        obal.append(n)

    return odpoved

# načítame vstup
N = int( input() )
body = []
for n in range(N):
    body.append( [ int(_) for _ in input().split() ] )

# spracujeme vstup aj jeho reverz, vypíšeme odpoveď
dolava = kolko_dolava(body)
body = [ (-x,y) for x,y in reversed(body) ]
doprava = list(reversed(kolko_dolava(body)))

for n in range(N): print( max( dolava[n], doprava[n] ) )
```

## A-II-4 Stromochod

Ukážeme si najskôr, ako skontrolovať, či je vybratá množina vrcholov naozaj nezávislá. Dobrá formulácia podmienky, ktorú máme skontrolovať, je nasledovná: ak je vrchol vybratý, žiaden jeho syn nesmie byť vybratý.

Naprogramujeme teda stromochod tak, aby celý strom prehľadal do hĺbky (ako v príklade v študijnom texte), len navyše pridáme, že v každom vrchole, keď doň posledný krát prídeme, skontrolujeme, či náhodou on spolu s niektorým jeho synom neporušujú našu podmienku. Stromochod bude mať svoju premennú, ktorú nosí so sebou a v ktorej si pamätá, či už našiel nejaký spor. Keď skončí celé prehľadávanie, negáciu tejto premennej zapíše do premennej `nezavisla` v koreni. Program má zjavne lineárnu časovú zložitosť.

### Listing programu (Pascal)

```
var nasiel_spor : boolean; { na začiatku nastavená na false }

type vrchol = record     { typ "vrchol" popisuje, aké značky ukladáme do vrcholov }
    stav: 0..3;          { počítadlo koľkokrát sme už tento vrchol navštívili }
    vybraty: boolean;    { vstup }
    nezavisla: boolean;  { v koreni táto značka slúži ako výstup, inde ju nepoužívame }
end;                     { na začiatku je v každom vrchole stav = 0 }

begin
    while true do begin { nekonečný cyklus }
        V.stav := V.stav + 1;
        case V.stav of
            1: if ex_l then krok_l;
            2: if ex_p then krok_p;
            3: begin
                { naposledy opúšťame vrchol, skontrolujeme, či tu nevzniká spor }
                if V.vybraty and ex_l then begin
                    krok_l;
                    if V.vybraty then nasiel_spor := true;
                    krok_o;
                end;
                if V.vybraty and ex_p then begin
                    krok_p;
                    if V.vybraty then nasiel_spor := true;
                    krok_o;
                end;

                { a po kontrole sa už len chceme vrátiť do nášho otca }
                if ex_o then krok_o else begin
                    { nemáme otca, sme teda v koreni stromu a program končí }
                    V.nezavisla := not nasiel_spor;
                    halt;
                end;
            end;
        end;
```

```
    end;
end.
```

Vo všeobecných grafoch je problém nájdenia najväčšej nezávislej množiny veľmi ťažký – dokonca až tak ťažký, že nepoznáme žiaden algoritmus riešiaci túto úlohu v polynomiálnom čase a domnievame sa, že taký algoritmus vôbec neexistuje. Našťastie, na stromoch je hľadanie najväčšej nezávislej množiny výrazne ľahšie: vieme ju vybrať pažravo. Existuje viacero spôsobov, ako to robiť, ukážeme si jeden, ktorý sa bude dať ľahko implementovať pomocou stromochodu.

Predstavme si, že už niekto našiel v našom strome najväčšiu nezávislú množinu a označil ju tak, že na každý vybratý vrchol položil kamienok. My teraz budeme tieto kamienky po strome posúvať. Vyrobíme si tak možno inú množinu, ale bude tiež nezávislá a bude rovnako veľká ako tá pôvodná (čiže tiež najväčšia možná), lebo kamienky len presúvame, nepridávame ich.

Ako budeme kamienky presúvať? Veľmi jednoducho: kedykoľvek, keď môžeme kamienok posunúť dodola (t.j. preč od koreňa) bez toho, aby sme pokazili nezávislosť našej množiny, tak to spravíme. Skončíme, keď už žiaden kamienok nevieme posunúť.

Ako bude vyzerať nezávislá množina, ktorú takto vyrobíme? V prvom rade vieme povedať, že v každom liste stromu bude kamienok. (Sporom. Nech nie je v nejakom liste $a$. Označme jeho otca $b$. Ak nie je kamienok ani v $b$, máme spor s maximálnosťou našej nezávislej množiny, lebo vieme pridať kamienok na $a$. A ak je kamienok na $b$, máme spor s tým, že by sme ho posunuli na $a$.)

Na vrcholoch, ktoré susedia s listami, potom žiadne kamienky byť nemôžu. No a podobnú úvahu môžeme robiť aj ďalej: ak mám vrchol v ktorého niektorom synovi je kamienok, on sám kamienok mať nesmie. A naopak, ak žiaden z jeho synov kamienok nemá, on kamienok mať musí. (Opäť sporom, ako vyššie. Keby kamienok nemal ani jeho otec, môžeme kamienok pridať, keby kamienok jeho otec mal, môžeme ho sem presunúť.)

No a použitím vyššie popísaného pravidla vieme našu maximálnu nezávislú množinu zostrojiť počas jedného prechodu stromu prehľadávaním do hĺbky, teda v lineárnom čase.

### Listing programu (Pascal)

```
var mozem_vybrat: boolean;

type vrchol = record     { typ "vrchol" popisuje, aké značky ukladáme do vrcholov }
    stav: 0..3;          { počítadlo koľkokrát sme už tento vrchol navštívili }
    vybraty: boolean;    { výstup, na začiatku je false }
end;                     { na začiatku je v každom vrchole stav = 0 }

begin
    while true do begin { nekonečný cyklus }
        V.stav := V.stav + 1;
        case V.stav of
            1: if ex_l then krok_l;
            2: if ex_p then krok_p;
            3: begin
                { naposledy opúšťame vrchol, skontrolujeme, či ho vybrať }
                mozem_vybrat := true;
                if ex_l then begin
                    krok_l;
                    if V.vybraty then mozem_vybrat := false;
                    krok_o;
                end;
                if ex_p then begin
                    krok_p;
                    if V.vybraty then mozem_vybrat := false;
                    krok_o;
                end;
                vybraty := mozem_vybrat;
                { a už sa len chceme vrátiť do nášho otca, resp. skončiť ak sme v koreni }
                if ex_o then krok_o else halt;
            end;
        end;
    end;
end.
```

**TRIDSIATY DRUHÝ ROČNÍK OLYMPIÁDY V INFORMATIKE**

Príprava úloh: Michal Anderle, Michal Forišek
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava 2017