67. ročník Matematické olympiády – 2017/2018

Řešení úloh domácího kola kategorie P

P-I-1 Trojice

Získat *nějaké* body v této úloze je snadné: stačí vygenerovat všechny trojice a uspořádat je podle součtu:

```
vector<long long> soucty;
for (int p=0; p<N; ++p)
    for (int q=0; q<p; ++q)
        for (int r=0; r<q; ++r)
            soucty.push_back( A[p] + A[q] + A[r] );
sort( soucty.begin(), soucty.end() );
cout << soucty[K-1] << endl;
```

Uvedené řešení má časovou složitost $\mathcal{O}(n^3 \log n)$ kvůli třídění. Tato časová složitost se dá ještě zlepšit na $\mathcal{O}(n^3)$ tím, že použijeme efektivnější třídění (např. radixsort), nebo tak, že namísto třídění použijeme lineární algoritmus na nalezení k-tého nejmenšího prvku. Protože ale existují mnohem lepší řešení, detaily těchto drobných zlepšení si odpustíme.

Kvadratické řešení

Jednou technikou, která vede k lepšímu řešení, je *binární vyhledávání odpovědi*. Použijeme nové značení $\varphi(x)$ pro počet trojic, které mají součet nejvýše x. Soutěžní úlohu si nyní můžeme zformulovat následovně: k danému k určete nejmenší x, pro něž je $\varphi(x) \geq k$. (Jinými slovy: existuje aspoň k trojic, které mají součet rovný x nebo méně.)

Víme, že hledaná hodnota x leží někde mezi 0 a $3 \cdot 10^{12}$. Kdybychom uměli hodnotu $\varphi(x)$ efektivně počítat, můžeme to správné x nalézt binárním vyhledáváním v tomto intervalu.

Jak dokážeme hodnotu $\varphi(x)$ spočítat v lepším než kubickém čase? Uvažujme výše uvedený algoritmus generující všechny trojice indexů (p, q, r). Když už jsme zvolili p a q, řešíme následující otázku: kolik různých možností pro r nám dá dostatečně malý součet? Jelikož čísla na vstupu máme uspořádána od nejmenšího po největší, místo zkoušení všech možných r stačí nalézt největší vyhovující r. To dokážeme nalézt v logaritmickém čase (opět) binárním vyhledáváním. Takto tedy můžeme konkrétní hodnotu $\varphi(x)$ spočítat v čase $\mathcal{O}(n^2 \log n)$. A protože správné x hledáme v intervalu konstantní délky, budeme potřebovat určit jenom konstantně mnoho hodnot φ, takže celková asymptotická časová složitost našeho řešení bude také $\mathcal{O}(n^2 \log n)$.

Ještě o něco lepší řešení se obejde bez vnitřního binárního vyhledávání. Namísto toho stačí všimnout si, že když pro pevně zvolené p postupně zvětšujeme q, tak se postupně zmenšuje maximální vyhovující r. Přesněji, jsou jen dvě možnosti: buď je maximální vyhovující r ještě stále rovno $q - 1$, nebo je už hodnota $A[p] + A[q]$ tak

velká, že trojice indexů $(p, q, q-1)$ nevyhovuje. Jakmile nastane tato druhá situace, už bude stále platit, že se při zvyšování q nikdy nezvýší maximální vyhovující r. Proto stačí udržovat si proměnnou s maximálním vyhovujícím r a pokaždé, když zvýšíme q, budeme tuto proměnnou podle potřeby snižovat. Toto řešení má časovou složitost $\mathcal{O}(n^2)$.

```
long long lo = A[0]+A[1]+A[2]-1, hi = A[N-1]+A[N-2]+A[N-3];

// Invariant: je málo trojíc se součtem <= lo, je dost trojic se součtem <= hi.
while (hi - lo > 1) {
    long long med = (lo+hi) / 2;
    // Spočítáme, kolik trojic má součet <= med.
    long long cnt = 0;
    for (int p=2; p<N; ++p) {
        int max_r = 0;
        for (int q=1; q<p; ++q) {
            // Upravíme hodnotu max_r: největší r takové, že A[p]+A[q]+A[r] <= med.
            if (A[p] + A[q] + A[q-1] <= med) {
                max_r = q-1;
            } else {
                while (max_r >= 0 && A[p] + A[q] + A[max_r] > med) --max_r;
            }
            cnt += (max_r + 1);
        }
    }
    // Podle výsledku zmenšíme na polovinu interval, v němž hledáme odpověď.
    if (cnt >= K) hi = med; else lo = med;
}

cout << hi << endl;
```

Vzorové řešení

Dosud jsme nijak nevyužili předpoklad uvedený v zadání, že k je rozumně malé. Vzorové řešení tuto skutečnost využívá: k-tý nejmenší součet najdeme tak, že postupně vygenerujeme všech k nejmenších součtů. Přesněji, vygenerujeme jim odpovídající trojice indexů (p, q, r), pričemž tentokrát bude vždy platit $p < q < r$.

Začátek výpočtu je snadný, nejmenší součet zjevně odpovídá trojici indexů $(0, 1, 2)$. Také druhý krok je vždy stejný – druhý nejmenší součet odpovídá trojici $(0, 1, 3)$. Jak ale budeme pokračovat?

V každém okamžiku si budeme udržovat množinu *kandidátů:* trojic, které by mohly být následující trojicí s nejmenším součtem mezi všemi ještě nezpracovanými trojicemi.

Na začátku výpočtu jsme ještě nevygenerovali žádnou trojici a máme jediného kandidáta: trojici $(0, 1, 2)$. Nyní k-krát zopakujeme tento postup:

1. Mezi všemi kandidáty vybereme tu trojici indexů, které odpovídá nejmenší součet (je-li takových více, tak libovolnou z nich). Vybranou trojici odstraníme ze seznamu kandidátů. Pokud je to už k-tá zpracovaná trojice, vypíšeme na výstup její odpovídající součet a skončíme.
2. Když jsme právě zpracovali trojici (a, b, c), přidáme mezi kandidáty trojice

$(a+1,b,c)$, $(a,b+1,c)$ a $(a,b,c+1)$. Přesněji přidáme jenom ty z nich, které jsou platnými uspořádanými trojicemi indexů.

Ukážeme, že toto řešení opravdu funguje, tedy že skutečně generuje trojice indexů ve správném pořadí. K tomu stačí dokázat následující tvrzení: V libovolném okamžiku platí, že když ještě nezpracovaná trojice indexů není mezi kandidáty, tak určitě nemá nejmenší součet mezi všemi nezpracovanými trojicemi.

Platnost tvrzení plyne z následujícího pozorování: trojice indexů (a,b,c) má ostře větší součet než každá z platných trojic $(a-1,b,c)$, $(a,b-1,c)$, $(a,b,c-1)$. Trojici (a,b,c) tedy chceme zpracovat až po všech těchto trojicích. Rozhodně tedy platí, že dokud žádná z nich nebyla zpracována, nemá smysl zařazovat (a,b,c) mezi kandidáty, když víme o nezpracované trojici s menším součtem.

Stačilo by zařadit (a,b,c) mezi kandidáty až poté, co zpracujeme *poslední* z výše uvedených „o jedna menších“ trojic, zatímco my to uděláme už po zpracování první z nich. To ale zjevně nepokazí korektnost našeho algoritmu – jenom máme občas množinu kandidátů o něco větší, než by bylo nutné. Jak uvidíme později, nepokazí to ani časovou složitost.

Při udržováni množiny kandidátů potřebujeme efektivně přidávat nové kandidáty a vybírat nejlepšího. Můžeme proto na jejich uložení použít prioritní frontu a tu implementovat buď pomocí haldy, nebo pomocí vyvažovaného stromu. V obou případech se bude každá operace s množinou kandidátů provádět v čase logaritmickém vzhledem k jejich počtu. Jelikož za každého zpracovaného kandidáta přidáme nejvýše tři nové, bude počet kandidátů lineárně narůstat s počtem už zpracovaných trojic, takže kandidátů bude $\mathcal{O}(k)$. Každého kandidáta zpracujeme v čase $\mathcal{O}(\log k)$, takže toto řešení má celkovou časovou složitost $\mathcal{O}(k \log k)$.

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

int N;
long long K;
vector<long long> A;

struct triple {
    int a,b,c;
    long long sum() const { return A[a]+A[b]+A[c]; }
};

set<triple> candidates;

bool operator<(const triple &X, const triple &Y) {
    long long xs = X.sum(), ys = Y.sum();
    if (xs != ys) return xs < ys;
    return make_tuple(X.a,X.b,X.c) < make_tuple(Y.a,Y.b,Y.c);
}

int main() {
    cin >> N >> K;
    A.resize(N);
    for (int n=0; n<N; ++n) cin >> A[n];

    candidates.insert( {0,1,2} );
```

```
    while (true) {
        triple curr = *candidates.begin();
        candidates.erase( candidates.begin() );
        --K;
        if (K == 0) { cout << curr.sum() << endl; break; }
        if (curr.a+1 < curr.b) candidates.insert( {curr.a+1,curr.b,curr.c} );
        if (curr.b+1 < curr.c) candidates.insert( {curr.a,curr.b+1,curr.c} );
        if (curr.c+1 < N     ) candidates.insert( {curr.a,curr.b,curr.c+1} );
    }
}
```

P-I-2 Telenovela

Nejprve zajistíme, abychom viděli první a poslední díl telenovely. Je zjevné, že první díl chceme vidět co nejdříve a poslední co nejpozději, aby nám mezi nimi zůstalo co nejvíce jiných dílů. Najdeme proto první vysílání prvního dílu a poslední vysílání posledního dílu. Pokud některý z nich vůbec nevysílají, nebo když první díl vysílají jen později než poslední, úloha nemá řešení. V opačném případě nám zůstala jednodušší verze soutěžní úlohy: v části televizního programu, která odpovídá úseku mezi první a poslední epizodou, potřebujeme vybrat co nejvíce ostatních epizod ve správném pořadí.

Jinými slovy, z dané posloupnosti čísel chceme vybrat co nejdelší *rostoucí podposloupnost* tvořenou čísly 2 až $e - 1$. Ukážeme si dva různě efektivní algoritmy, které řeší tento problém.

Kvadratické řešení

Uvažujme následující verzi naší úlohy: na vstupu je posloupnost čísel $a_1, \ldots, a_n$ a nás zajímá délka nejdelší ostře rostoucí podposloupnosti této posloupnosti.

Při řešení použijeme dynamické programování. Označme b_i délku nejdelší rostoucí podposloupnosti, jejíž poslední prvek leží na indexu i. Celkovým výsledkem je pak zjevně $\max_i b_i$, takže k vyřešení úlohy nám stačí postupně spočítat všechny hodnoty od b_1 po b_n.

Jak to uděláme? Je-li na indexu i poslední prvek rostoucí podposloupnosti, jsou jen dvě možnosti: buď má tato podposloupnost délku 1 (tzn. už žádný jiný prvek neobsahuje), nebo má předposlední prvek na nějakém indexu $j < i$. Jelikož celá posloupnost má být rostoucí, připadají do úvahy jen taková j, pro která platí $a_j < a_i$. Pro každé takové j nejdelší rostoucí podposloupnost, která končí na indexu j, má délku b_j. Když tedy chceme získat nejdelší rostoucí podposloupnost končící na indexu i, musíme mezi vhodnými j vybrat to s největší hodnotou b_j. Příslušná podposloupnost končící na indexu j spolu s prvkem na indexu i potom vytvoří rostoucí podposloupnost délky $b_j + 1$ končící na indexu i.

Všechny hodnoty b_i spočítáme v čase $\mathcal{O}(n^2)$ následovně:

```
// Vstup máme uložen v proměnné N a v poli hodnot A[0..N-1].
vector<int> B(N);

for (int i=0; i<N; ++i) {
    // Hodnotu B[i] inicializujeme na 1, což odpovídá jednoprvkové posloupnosti.
```

```
    B[i] = 1;
    // Je-li optimální posloupnost delší, vyzkoušíme všechny možnosti,
    // kde má předposlední prvek, a vybereme nejlepší z nich.
    for (int j=0; j<i; ++j)
        if (A[j] < A[i])
            B[i] = max( B[i], 1+B[j] );
}
```

Vzorové řešení

Existuje více různých algoritmů, které najdou nejdelší rostoucí podposloupnost v čase $\mathcal{O}(n \log n)$. Ukážeme si jeden z nich. Bude zajímavý také tím, že je to online algoritmus, který bude fungovat, aniž by předem znal délku zpracovávané posloupnosti. Jednoduše budeme ze vstupu postupně číst prvky posloupnosti a po přečtení každého z nich budeme vědět, jaká nejlepší podposloupnost je obsažena v dosud zpracovaných datech.

Algoritmus založíme na následujícím základním pozorování: Představme si, že jsme už zpracovali nějakou část vstupní posloupnosti, například

$$(10, 3, 8, 6, 9, 4, 6, 22, 7, 5).$$

V této posloupnosti mohlo být mnoho různých dvouprvkových rostoucích podposloupností, například $(10, 22)$, $(6, 7)$, nebo $(3, 4)$. Nyní nám na vstupu přijde nový prvek x. Chceme vytvořit tříprvkovou rostoucí podposloupnost končící tímto x. To musíme udělat tak, že vezmeme některou dvouprvkovou rostoucí podposloupnost, kterou jsme měli v už zpracovaných datech, a na její konec přidáme x. Která ze všech možných dvouprvkových podposloupností je k tomu nejvhodnější? Je zjevné, že když je například posloupnost $(5, 7, x)$ rostoucí, pak také posloupnost $(3, 4, x)$ bude rostoucí, neboť když $x > 7$, tak tím spíše platí i $x > 4$. Jinými slovy, nejlepší je ta posloupnost, která končí nejmenším možným číslem. Každé x, které by šlo připojit za jinou podposloupnost, půjde připojit i za tuto.

Budeme si tedy udržovat hodnoty c_j s následujícím významem: když se podíváme na všechny možné rostoucí j-prvkové podposloupnosti v již zpracovaných datech a vezmeme poslední prvek každé z nich, nejmenší z takto získaných hodnot bude právě c_j. Speciálně položíme $c_0 = -\infty$ (za posloupnost délky 0 lze připojit cokoliv) a $c_j = +\infty$, pokud ve zpracovaných datech žádná rostoucí j-prvková podposloupnost neexistuje.

Příklad: Pokud jsme již zpracovali $(10, 3, 8, 6, 9, 4, 6, 22, 7, 5)$, potom budeme mít $c_1 = 3$, $c_2 = 4$, $c_3 = 5$, $c_4 = 7$, a $c_5 = +\infty$. Všimněte si, že různé hodnoty c_i mohou odpovídat různým podposloupnostem. Například v této chvíli nejlepší podposloupností délky 3 je $(3, 4, 5)$, zatímco pro délku 4 to je $(3, 4, 6, 7)$.

Použijeme následující pozorování: konečné hodnoty c_i jsou vždy uspořádané podle velikosti v ostře rostoucím pořadí. Například vždy platí $c_3 < c_4$, neboť když vezmeme *nejlepší* rostoucí podposloupnost délky 4 a odstraníme z ní její poslední prvek, dostaneme *nějakou* (dokonce ne nutně nejlepší) rostoucí podposloupnost délky 3, která končí hodnotou ostře menší než c_4.

Představme si nyní, že přečteme ze vstupu další prvek x a zajímá nás, jaká nejdelší rostoucí podposloupnost končí tímto prvkem. Abychom to zjistili, chceme

nalézt největší i takové, že $c_i < x$. Potom je zjevné, že nejdelší rostoucí podposloupnost končící právě přečteným prvkem x má délku $i + 1$. Protože víme, že hodnoty c jsou uspořádány podle velikosti, dokážeme hledané i určit v logaritmickém čase binárním vyhledáváním.

Zbývá nám poslední krok: zjistit, jaké nové rostoucí podposloupnosti vznikly tím, že jsme do posloupnosti přidali právě přečtený prvek x. Přesněji, chceme zjistit, které z hodnot c se změnily a jak.

Ukáže se, že změna je vždy minimální. V předchozím kroku jsme našli největší i takové, že $c_i < x$. Nyní platí:

- Pro každé $j \leq i$ existuje j-prvková rostoucí podposloupnost končící prvkem menším než x. Hodnoty c_1 až c_i se proto nezmění.
- Nelze vytvořit rostoucí podposloupnost délky $i + 2$ nebo větší, která by končila právě přečteným x. Ani hodnoty od c_{i+2} dále se proto nezmění.
- Jediné, co se může změnit, je hodnota c_{i+1}. Dosud mohlo být $c_{i+1} > x$, nyní zjevně bude $c_{i+1} = x$.

Například když budeme pokračovat ve výše uvedeném příkladu tím, že přečteme ze vstupu jako další prvek posloupnosti hodnotu $x = 6$, změní se hodnota c_4 ze 7 na 6. Kdybychom namísto toho přečetli $x = 100$, zůstalo by $c_4 = 7$ a změnilo by se c_5 z ∞ na 100.

Po přečtení a zpracování n prvků budeme mít nejvýše n-prvkovou rostoucí podposloupnost, proto v libovolném okamžiku jen $\mathcal{O}(n)$ prvků posloupnosti c má konečnou hodnotu. Binární vyhledávání mezi nimi proto proběhne v čase $\mathcal{O}(\log n)$. Každý prvek ze vstupu tedy přečteme a zpracujeme v logaritmickém čase, proto celková časová složitost tohoto řešení je $\mathcal{O}(n \log n)$.

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

// V čase n log n vypočítáme délku nejdelší rostoucí podposloupnosti v A.
int lis_nlogn(const vector<int> &A) {
    // Inicializujeme si C tak, aby C[0]=-inf;
    // v každém okamžiku si budeme pamatovat jen tu část C, která je < inf.
    vector<int> C(1, -1);
    for (int a : A) {
        // Najdeme nejmenší i takové, že C[i] >= a.
        unsigned i = upper_bound( C.begin(), C.end(), a-1 ) - C.begin();
        // Upravíme hodnotu C[i] na a.
        if (i == C.size()) C.push_back(a); else C[i] = a;
    }
    return C.size()-1;
}

int main() {
    int N, E;               cin >> N >> E;
    vector<int> epizody(N); for (int n=0; n<N; ++n) cin >> epizody[n];

    // Najdeme první vysílání první a poslední vysílání poslední epizody.
    int prvni = 0, posledni = N-1;
    while (prvni < N && epizody[prvni] != 1) ++prvni;
    while (posledni >= 0 && epizody[posledni] != E) --posledni;
```

```
    // Pokud se nestíhají obě nebo některou vůbec nevysílají, řešení neexistuje.
    if (posledni < prvni) { cout << -1 << endl; return 0; }

    // Sestrojíme posloupnost vysílání ostatních epizod, které stihneme shlédnout.
    vector<int> zustalo;
    for (int i=prvni+1; i<posledni; ++i)
        if (1 < epizody[i] && epizody[i] < E)
            zustalo.push_back( epizody[i] );

    // Najdeme mezi nimi nejdelší rostoucí podposloupnost
    // a přičteme 2 za první+poslední.
    cout << 2 + lis_nlogn(zustalo) << endl;
}
```

P-I-3 Převracení prefixů

Řešení jednotlivých částí úlohy ukážeme v trochu jiném pořadí: část C si necháme až na konec.

Část A: uspořádání pole

Uspořádané pole budeme vytvářet od konce. Pomocí nejvýše dvou operací *flip* umíme dostat největší prvek na konec pole. Stačí ho najít v poli (nechť je na indexu k), provést *flip*(k), čímž ho dostaneme na začátek pole, a následně provést *flip*(n), čímž ho dostaneme na konec pole. Od této chvíle budeme provádět příkaz *flip* pouze s parametrem menším než n, takže největší prvek už zůstane trvale na konci pole. Analogicky budeme postupovat dále. Takto celé pole uspořádáme pomocí nejvýše $2n - 2$ operací *flip*. Před každou z nich potřebujeme nejvýše $\mathcal{O}(n)$ času, proto má tento algoritmus polynomiální (přesněji kvadratickou) časovou složitost.

Část B: testování pro 10 prvků

V této části úlohy máme postupně projít všech 10! permutací deseti prvků a pro každou z nich odsimulovat deterministický postup ze zadání.

Implementovat postup ze zadání je snadné a kromě toho potřebujeme už jenom procházet přes všechny permutace. Při praktické implementaci k tomu lze využít vhodnou knihovní funkci (např. `next_permutation` v C++, případně `itertools.permutations` v Pythonu). Pokud bychom chtěli takovou funkci implementovat sami, můžeme naprogramovat pomocnou funkci, která z dané permutace vytvoří permutaci bezprostředně následující v lexikografickém uspořádání. Podrobnější popis implementace této funkce najdete například v řešení úlohy P-II-3 z 60. ročníku MO – kategorie P.

Ukázka implementace:

```
def simuluj(pole):
    kroku = 0
    while pole[0] != 1:
        k = pole[0]
        pole = pole[:k][::-1] + pole[k:]
        kroku += 1
    return kroku

from itertools import permutations
```

```
pocty_kroku = [ ( simuluj(perm), perm ) for perm in permutations(range(1,11)) ]
pocty_kroku.sort()
for row in pocty_kroku[-10:]: print(row)
```

Spuštěním programu zjistíme, že nejvyšší počet kroků je 38, a to pro jedinou permutaci $(5, 9, 1, 8, 6, 2, 10, 4, 7, 3)$.

Část D: vždy skončíme?

Jak už naznačují naše experimenty, deterministický proces popsaný v zadání úlohy vždy konverguje k situaci, že se na začátek posloupnosti dostane číslo 1. Toto pozorování dokážeme sporem. Předpokládejme, že to neplatí, tedy že pro nějakou permutaci se číslo 1 nikdy na začátek nedostane.

Představme si, že náš proces budeme opakovat do nekonečna. Jelikož různých permutací je jen konečně mnoho, časem se jistě některá permutace zopakuje. A protože náš proces je deterministický, od tohoto okamžiku se celý proces zacyklí a bude se stále opakovat stejná posloupnost permutací.

Označme k největší číslo, které se během tohoto cyklu objeví na začátku permutace. Předpokládejme, že $k > 1$. Co se stane? Hned v následujícím kroku provedeme *flip*(k), čímž se číslo k dostane na index k. Z definice k víme, že všechna ostatní čísla, která se během cyklu objeví na začátku permutace (včetně toho, které je tam nyní) jsou menší než k. To ale znamená, že nyní budeme provádět příkaz *flip*(ℓ) pro nějaké $\ell < k$. Takový *flip* ovšem nechá prvek k na místě, proto musí na začátek pole přesunout nějakou hodnotu menší než k. Zde ovšem dostáváme hledaný spor – na jedné straně jsme předpokládali, že k se na začátku objevuje periodicky, na druhé straně ale vidíme, že k se na začátku již nikdy neobjeví, protože tam stále budeme mít hodnotu ostře menší než k.

Zbývá tudíž jediná možnost: $k = 1$, a proto se náš proces nutně vždy zacyklí v situaci, v níž je na začátku pole číslo 1.

(Jiná formulace důkazu: Nechť x je největší číslo které se aspoň jednou objeví na začátku permutace. Jestliže $x > 1$, pak podobnou úvahou jako výše můžeme ukázat, že se x na začátku permutace objeví právě jednou a od té chvíle už zůstane trvale na indexu x. Označme nyní t_1 ten okamžik, kdy to nastane, a od něj dále zopakujeme stejnou úvahu: jaké největší číslo se objeví na začátku po okamžiku t_1? Každou iterací této úvahy se dostaneme k menšímu číslu, a protože těchto čísel máme jen konečně mnoho, po konečném počtu opakování této úvahy se dostaneme k tomu, že největším číslem, které se může objevit na začátku permutace, je už jen číslo 1.)

Část C: hledání dobré permutace

Na závěr vzorového řešení jsme si ponechali tu část úlohy, která byla úmyslně zformulována otevřeně: bylo třeba nalézt libovolnou dostatečně dobrou permutaci.

Už z části B máme implementovánu funkci, která nám pro konkrétní permutaci spočítá počet kroků. Nejsnadnějším způsobem, jak získat nějaké body, je zkoušet tuto funkci spustit na různých náhodných vstupech a zaznamenat si nejlepší z nich.

Podle našich experimentů by mělo několik milionů pokusů téměř jistě stačit k překročení 500-krokové hranice. Při několika miliardách pokusů (tedy pokud tentýž

program necháte počítat přes noc) byste už měli narazit na nějakou permutaci, která potřebuje více než 750 kroků.

Existují i efektivnější způsoby hledání. Můžeme si například všimnout, že pro permutace, které se liší jen na několika pozicích, bude často značná část kroků našeho algoritmu probíhat stejně. Proto když najdeme nějakou permutaci, která vyžaduje mnoho kroků, je dobré podívat se i na další podobné permutace. Na tomto pozorování lze založit například algoritmus, který při hledání dobrých permutací používá optimalizační techniku *hill climbing*. Nejlepší permutace, kterou náš šikovnější program našel za několik hodin, potřebuje 899 kroků. Vypadá následovně: (35, 45, 23, 38, 11, 6, 15, 39, 43, 32, 20, 18, 30, 16, 5, 29, 7, 12, 14, 37, 31, 3, 17, 41, 36, 13, 19, 2, 10, 28, 44, 42, 47, 1, 25, 46, 21, 8, 27, 34, 4, 26, 22, 33, 24, 9, 40). Ani tato permutace zdaleka ještě není nejlepší možná.

Dodejme ještě, že neznáme žádný exaktní postup, jenž by pro dané n zaručeně vytvořil permutaci, která bude potřebovat velké množství kroků.

P-I-4 Stavebnice funkcí

Část A: ternární nula

Máme sestrojit funkci zzz^3, která má tři vstupy, ale bez ohledu na jejich hodnotu vždy vrátí nulu.

Ve studijním textu jsme v Příkladu 5 sestrojili unární konstantní nulu, tedy funkci zz^1, která má jeden vstup a vždy vrátí nulu. Nejsnadnější způsob, jak nyní vytvořit funkci zzz^3, je pomocí Kompozitoru. Složíme dohromady například vybírací funkci v_1^3 s funkcí zz^1. Vznikne nám tím funkce se třemi vstupy. Ty vložíme do funkce v_1^3, která z nich spočítá nějakou pomocnou hodnotu. Je jedno jakou, neboť tu následně vložíme do funkce zz^1, která na výstup vrátí nulu. Formálně můžeme tuto konstrukci zapsat následovně: $zzz^3 \equiv K[v_1^3, zz^1]$.

(Namísto v_1^3 jsme mohli použít libovolnou jinou ternární funkci. Popsaná konstrukce by stále fungovala, jelikož na výstupu použité ternární funkce vůbec nezáleží.)

Jiné řešení této úlohy využívá Cyklovač. V Příkladu 5 jsme uvedli postup, kterým jsme z konstantní funkce s aritou 0 sestrojili konstantní funkci s aritou 1. Když analogický postup zopakujeme ještě dvakrát, dostaneme postupně konstantní funkci s aritou 2 a následně s aritou 3.

Formálně: nejprve si sestrojíme pomocnou funkci $zzzz^2 \equiv C[zz^1, v_3^3]$ a z ní potom získáme hledanou funkci $zzz^3 \equiv C[zzzz^2, v_4^4]$.

Část B: přerovnání parametrů

V této části úlohy máme z neznámé funkce φ^4 vytvořit novou funkci ψ^3 definovanou následovně: $\forall a, b, c : \psi^3(a, b, c) = \varphi^4(b, a, c, a)$.

Pro takovéto „technické úpravy“ používáme vybírací funkce. Funkci ψ sestrojíme pomocí Kompozitoru. Řekneme mu, že ve druhém kroku chceme použít funkci φ^4 a že v prvním kroku jí chceme vybrat jednotlivé vstupy vhodnými vybíracími funkcemi. Přesněji, do φ^4 chceme postupně zadat druhý, první, třetí a znovu první vstup.

Výsledná konstrukce tedy vypadá následovně: $\psi \equiv K[\, v_2^3, v_1^3, v_3^3, v_1^3,\ \varphi^4\,]$.

Část C: násobení

Podobně jako sčítání vzniklo opakovaným použitím funkce následníka, tak násobení dostaneme opakovaným použitím funkce sčítání. Bude tedy třeba udělat něco podobného, jako ve studijním textu v Příkladu 4 při konstrukci sčítací funkce *add*.

Hledanou funkci *mul* chceme sestrojit pomocí Cyklovače. Musíme zjistit, jaké dvě funkce f a g do něj máme vložit, aby nám vypadlo právě násobení. Když do Cyklovače vložíme unární funkci f a ternární funkci g, dostaneme následující binární funkci:

```
def mul(x,y):
    tmp = f(y)
    for i = 0 to x-1:
        tmp = g(i,y,tmp)
    return tmp
```

Pro $x = 0$ tato funkce vrátí hodnotu $f(y)$. Protože nula krát cokoliv je nula, potřebujeme, aby $f(y)$ vždy vrátila nulu. Jinými slovy, jako f chceme použít unární konstantní nulu, tedy funkci zz^1 ze studijního textu.

Program se nám tím upravil do tvaru:

```
def mul(x,y):
    tmp = 0
    for i = 0 to x-1:
        tmp = g(i,y,tmp)
    return tmp
```

Nyní bychom chtěli, aby každé z x použití funkce g zvýšilo proměnnou *tmp* o y. Funkce g tudíž musí vrátit na výstup hodnotu $y + tmp$. Jako g proto chceme použít funkci *tří* proměnných, která na výstup vrátí součet druhého a třetího vstupu. To je *skoro* funkce *add*, ale ne úplně. Naštěstí jsme již v části B vymysleli, jak funkci *add* upravit do požadované podoby. Nejprve si tedy z funkce *add* vytvoříme Kompozitorem vhodnou funkci $g \equiv K[v_2^3, v_3^3, add]$, potom už Cyklovačem sestrojíme násobení: $mul \equiv C[zz^1, g]$.

Část D: předchůdce

K sestrojení funkce předchůdce šikovně (dalo by se říci, že až trikově) použijeme Cyklovač. Začátek je snadný: předchůdcem nuly je nula. Co ale udělat pro $x > 0$? Podíváme se opět na pseudokód funkce, kterou nám Cyklovač vyrobí, jestliže do něj vložíme nulární funkci z a neznámou binární funkci g:

```
def p(x):
    tmp = z()              # t.j. tmp = 0
    for i = 0 to x-1:
        tmp = g(i,tmp)
    return tmp
```

Dosud jsme vždy při použití Cyklovače ignorovali proměnnou i, tedy řídicí proměnnou cyklu. Tentokrát ji použijeme. Všimněte si, že tato proměnná postupně

nabývá hodnoty od 0 do $x-1$. Poslední hodnota i je tedy přesně tou hodnotou, kterou chceme vrátit na výstup. Úplně proto postačí, když funkce g vždy vrátí na výstup svůj první vstup. Tím dostaneme následující pseudokód:

```
def p(x):
    tmp = 0
    for i = 0 to x-1:
        tmp = i
    return tmp
```

Je to sice neefektivní postup (kdo to kdy viděl počítat předchůdce v lineárním čase, že?), ale dělá to přesně to, co má.

Vidíme tedy, že unární funkci předchůdce sestrojíme Cyklovačem z funkce z a vybírací funkce v_1^2: $p \equiv C[z, v_1^2]$.

Část E: odčítání

Chceme sestrojit funkci, která se co nejvíce podobá odčítání: binární funkci *sub* takovou, že pro $x > y$ je $sub(x, y) = x - y$ a pro $x \leq y$ je $sub(x, y) = 0$.

Základní myšlenka je zjevná: když je sčítání opakovaným použitím následníka, tak odčítání je opakovaným použitím předchůdce. Tím automaticky zabezpečíme i to, že pro $x < y$ dostaneme jako výsledek nulu. Kdybychom chtěli například spočítat 3 mínus 7, tak na číslo 3 použijeme 7-krát funkci p, čímž dostaneme nulu, neboť $p(0) = 0$.

Drobný problém spočívá v tom, že Cyklovač vždy použije první parametr funkce jako počet opakování. Nemůžeme tedy odčítání sestrojit přímo. Vytvoříme si proto nejprve pomocnou funkci *bus*, která bude mít parametry v opačném pořadí: $bus(x, y)$ bude y mínus x.

Funkci *bus* sestrojíme pomocí Cyklovače. Pro $x = 0$ platí, že y mínus 0 je y, jako první funkci do Cyklovače proto vložíme identitu. Druhou funkcí bude, analogicky k Příkladu 4 ze studijního textu, funkce $K[v_3^3, p]$ – tedy funkce „vezmi dosavadní hodnotu *tmp* a použij na ni funkci předchůdce“. Formálně $bus \equiv C[v_1^1, K[v_3^3, p]]$.

Zaměnit pořadí parametrů již umíme z řešení části B: $sub \equiv K[v_2^2, v_1^2, bus]$.