52. ročník Matematické olympiády – 2002/2003

Úlohy celostátního kola kategorie P – 1. soutěžní den

Na řešení úloh máte 4,5 hodiny čistého času.

Řešení každého příkladu musí obsahovat:

- **Popis řešení,** to znamená slovní popis použitého algoritmu, argumenty zdůvodňující jeho správnost (případně důkaz správnosti algoritmu), diskusi o efektivitě vašeho řešení (časová a paměťová složitost). Slovní popis řešení musí být jasný a srozumitelný i bez nahlédnutí do samotného zápisu algoritmu (do programu).
- **Program.** V úlohách **P-III-1** a **P-III-2** je třeba uvést dostatečně podrobný zápis algoritmu, nejlépe ve tvaru zdrojového textu nejdůležitějších částí programu v jazyce Pascal nebo C. Ze zápisu můžete vynechat jednoduché operace jako vstupy, výstupy, implementaci jednoduchých matematických vztahů apod. V úloze **P-III-3** místo toho zapište své řešení jako reverzibilní proceduru.

Hodnotí se nejen správnost programu, ale také kvalita popisu řešení a efektivita zvoleného algoritmu.

P-III-1 Hračkářství

V hračkářství Prcek a otec proběhla velká soutěž „O nejhezčí hračku“. Děti měly za úkol nakreslit obrázek své nejoblíbenější hračky. Po ukončení soutěže byla uspořádána výstavka a děti, které nakreslily nejpěknější obrázky, dostaly od hračkářství nějakou hračku. Jak ale asi víte, ne každému dítěti se líbí každá hračka, a tak už před vyhlášením soutěže měl každý malý výtvarník vyhlédnutou tu odměnu, kterou chtěl za svůj obrázek dostat. Tu a žádnou jinou. Svůj názor pak děti po vyhlášení dávaly dost hlasitě najevo. Maminky ječících potomků se tedy rozhodly, že děti si hračky mezi sebou povyměňují tak, aby pokud možno co nejvíce dětí bylo se svou výhrou spokojeno. Situaci ještě navíc komplikuje skutečnost, že k výměně jsou ochotné pouze ty děti, které nakonec dostanou hračku, po níž touží. S tak náročným úkolem si maminky nevěděly rady, a tak poprosily vás, abyste napsali program, který problém vyřeší.

Váš program dostane na vstupu zadán počet odměněných dětí N a dále pro každé dítě číslo hračky, kterou dostalo, a číslo hračky, kterou by chtělo dostat (protože hraček je stejně jako dětí, očíslujeme si je pro jednoduchost od jedné do N). Na výstup program vypíše největší skupinu dětí takovou, že když si děti ve skupině mezi sebou vhodným způsobem vymění hračky, budou všechny spokojené.

P-III-2 Knihovna

Knihovnice Míla opět potřebuje objednat další skříň do své knihovny a protože se jí vaše pomoc osvědčila, opět se na vás obrátila, abyste jí pomohli spočítat optimální rozměry nové skříně. Nová skříň má mít P poliček a Míla by do ní ráda umístila celkem N knih. Každá kniha má přiřazen jednoznačný číselný kód a tyto kódy určují pořadí knih ve skříni. Kniha s menším kódem se má nacházet na stejné nebo výše umístěné poličce než kniha s větším kódem; na každé poličce mají být knihy s menšími kódy umístěny vlevo od knih s většími kódy. Vstupem vašeho programu bude celé číslo P a posloupnost N čísel t_i, $1 \leq i \leq N$, kde t_i je tloušťka i-té knihy. Můžete předpokládat, že tloušťka t_i i-té knihy je celé číslo z rozmezí od 1 mm do 50 mm. Výška každé knihy je taková, že ji lze bez problémů umístit do libovolné z plánovaných P poliček. Váš program by měl ze zadaných údajů spočítat následující:

- Šířku skříně – označme ji s.
- Rozmístění knih do skříně se spočítanými parametry.

Rozmístění knih, které váš program nalezne, musí z pochopitelných důvodů splňovat následující:

- Součet tlouštěk knih umístěných do jedné poličky je nejvýše s mm.
- Šířka skříně s je nejmenší možná.

Příklad: Předpokládejme, že nová skříň má mít 3 poličky a má být do ní umístěno celkem 6 knih s následujícími tloušťkami (seřazeny vzestupně podle svých kódů): 15 mm, 20 mm, 7 mm, 6 mm, 2 mm a 4 mm. Minimální možná šířka skříně v tomto případě je 20 mm – na první poličku se dá pouze první kniha, na druhou pouze druhá kniha a zbylé knihy se umístí na poslední třetí poličku.

P-III-3 Reverzibilní výpočty: Sčítání

(Definice reverzibilních výpočtů je stejná jako v domácím i krajském kole, jen jsme některá důležitá pravidla zvýraznili. Naleznete ji ve studijním textu za touto úlohou.)

Napište *reverzibilní* proceduru `Add(var n:word; var A,B:array [0..n-1] of bit)` sloužící ke sčítání dvou n-bitových čísel zapsaných ve dvojkové soustavě (bit číslo 0 odpovídá řádu jednotek). Tato procedura přičte číslo uložené v poli B k číslu uloženému v poli A. Vstup dostane vždy takový, aby nedošlo k přetečení, tedy součet bude vždy také n-bitový. Snažte se dosáhnout co nejmenší *prostorové* složitosti vaší procedury.

Studijní text:

Při hledání nových úspornějších polovodičových technologií se zjistilo, že nejvíce energie se spotřebovává při mazání informací, tudíž že optimální jsou ty výpočty, při nichž se žádné informace neztrácejí. Takovým výpočtům se říká *reverzibilní*, protože díky této vlastnosti mohou probíhat oběma směry – dokáží nejen spočítat ze vstupu výstup, ale také z výstupu jednoznačně určit vstup. Vydejme se proto i my do tohoto zvláštního symetrického světa a prozkoumejme, jak se programuje „ekologicky“.

Začněme tím nejjednodušším, co se v klasických programovacích jazycích vyskytuje, a to je přiřazovací příkaz. Nic takového si bohužel dovolit nemůžeme, ztratili bychom totiž původní obsah proměnné, do níž se přiřazuje. Místo toho zavedeme několik příkazů modifikujících proměnnou vratně:

- *proměnná* `+=` *hodnota* – přičte hodnotu k proměnné.
- *proměnná* `-=` *hodnota* – odečte hodnotu od proměnné.
- *proměnná* `^=` *hodnota* – přixoruje hodnotu k proměnné. (*xor* je bitová operace, která má pro jednobitová čísla výsledek 1 právě tehdy, když jsou oba vstupy různé: $0 \text{ xor } 0 = 1 \text{ xor } 1 = 0$, $0 \text{ xor } 1 = 1 \text{ xor } 0 = 1$. Vícebitová čísla se xorují po bitech – i-tý bit prvního čísla s i-tým bitem druhého dají i-tý bit výsledku: $5 \text{ xor } 15 = (0101)_2 \text{ xor } (1111)_2 = (1010)_2 = 10$. Obecně pro libovolná čísla x a y platí $x \text{ xor } y = y \text{ xor } x$, $x \text{ xor } x = 0$, $x \text{ xor } 0 = x$ a $(x \text{ xor } y) \text{ xor } z = x \text{ xor } (y \text{ xor } z)$. Podobně lze zavést operace *and* a *or*: $0 \text{ and } 0 = 0 \text{ and } 1 = 1 \text{ and } 0 = 0$, $1 \text{ and } 1 = 1$, $0 \text{ or } 0 = 0$, $0 \text{ or } 1 = 1 \text{ or } 0 = 1 \text{ or } 1 = 1$, ale ty nejsou reverzibilní, takže pro nás nebudou tak důležité.)
- *proměnná* `=:=` *proměnná* – prohodí obsah dvou proměnných.

Abychom se vyhnuli problémům s přetečením (co by pak byla inverzní operace?), dohodněme se, že budeme počítat pouze s nezápornými celými čísly v rozsahu $0 \ldots maxword$ (takovým číslům budeme říkat *přirozená*) a všechny operace budou vydávat výsledky modulo $maxword + 1$, tedy opět přirozené číslo. Příkaz `+=` provedený pozpátku je pak totéž, co `-=` a opačně; příkazy `^=` a `=:=` jsou inverzní samy k sobě.

Co všechno ale může být *hodnota*? Jistě libovolná konstanta nebo proměnná (ovšem různá od té, do které přiřazujeme, jinak bychom mohli napsat třeba `a -= a`, což určitě reverzibilní není). Také bychom měli povolit nějaké další aritmetické operace – ty samy nemusí být reverzibilní; důležité je, aby se jejich výsledek zpracoval reverzibilně. Každý složitější výraz pak už můžeme přepsat na výrazy s jedinou operací, například `x ^= (a*b)+(c*d)` rozepíšeme takto:

```
t1 += a*b;
t2 += c*d;
x  ^= t1+t2;
t2 -= c*d;
t1 -= a*b;
```

Zde `t1` a `t2` jsou pomocné proměnné, které jsou na počátku výpočtu nulové a po dopočítání výrazu se opět k nulovým hodnotám vrátí, takže je můžeme používat pro všechny výrazy v celém programu. Podobně se vypořádáme s každým výrazem – nejdříve si spočítáme všechny mezivýsledky do pomocných proměnných, pak hlavní výsledek použijeme, načež mezivýsledky opět „odpočítáme“. Takže můžeme používat i složité výrazy a spolehnout se na překladač, že je sám rozepíše.

Trik s odpočítáváním mezivýsledků a spouštěním částí programu pozadu je, zdá se, velice šikovný, tak si rovnou nadefinujeme, že `undo` *příkaz* znamená spustit *příkaz* pozpátku a `wrap` *příkaz*$_1$ `on` *příkaz*$_2$ provede nejdříve *příkaz*$_1$, pak *příkaz*$_2$ a nakonec `undo` *příkaz*$_1$ pro odpočítání mezivýsledků. Náš příklad s výrazem pak snadno zapíšeme takto:

```
wrap begin
   t1 += a*b;
```

```
    t2 += c*d
    end
on x ^= t1+t2
```

Podmíněné příkazy `if`–`then`–`else` můžeme používat bez obav, pokud zaručíme, že po provedení podmíněného příkazu dopadne podmínka úplně stejně jako předtím (třeba proto, že žádná z proměnných, které v ní vystupují, není v podmíněné části programu měněna). Pak totiž i při provádění výpočtu pozpátku rozpoznáme, kterou z větví se výpočet má vydat.

S cykly je situace svízelnější, protože tam si s neměnícími se podmínkami nevystačíme (to by každý cyklus buďto neproběhl nikdy nebo by se opakoval do nekonečna). Dalo by se to, pravda, zachránit tím, že by každý cyklus měl jednu podmínku, která by fungovala současně jako vstupní i výstupní – sami si rozmyslete, jak by takové cykly vypadaly. My si ale pro naše účely vystačíme s cykly `for`, ty určitě reverzibilní jsou, pokud řídící proměnnou cyklu ani její meze žádný příkaz uvnitř cyklu nemodifikuje, a to se koneckonců nesmí ani v mnoha jiných programovacích jazycích. Navíc abychom nemuseli řešit, co se v řídící proměnné musí vyskytovat před začátkem cyklu a co po jeho konci, domluvíme se, že příkaz `for` si tuto proměnnou sám vytvoří a na konci ji zase zruší.

Příkaz `goto` pro jistotu zakážeme úplně.

Procedury mohou také fungovat reverzibilně, ale musíme se vyhnout kopírování parametrů a výsledků; budeme proto vše vždy předávat odkazem (pascalské `var`). Lokální proměnné budou při spuštění procedury vždy nulové a procedura sama je musí, než skončí, opět do tohoto stavu vrátit. Rekurze je bez problémů.

Nyní již máme vše potřebné, abychom si vybudovali reverzibilní programovací jazyk. Ten náš bude vzdáleným příbuzným Pascalu. Vypadá takto:

Datové typy: K dispozici máme typy `word` (celá čísla bez znaménka), `bit` (jednobitové číslo, tedy `0` nebo `1`; používá se rovněž pro pravdivostní hodnoty) a pole `array [`x`..`$y`$`] of` *typ* ($x$ a $y$ udávají meze indexů a jsou to buďto čísla nebo výrazy, jejichž hodnota se po dobu existence pole nezmění – to si proti Pascalu dovolíme navíc). Prvky polí mohou být také pole, čímž získáme pole vícerozměrná. Svůj vlastní typ si můžete zavést deklarací `type` *identifikátor* `=` *typ*`;`

Identifikátory slouží k pojmenovávání typů, proměnných a procedur a jsou to libovolné řetězce písmen, číslic a znaků '`_`', které nezačínají číslicí a které se neshodují s některým z klíčových slov jazyka (zde sázena `courierem`). Malá a velká písmena se nerozlišují.

Procedury se deklarují konstrukcí

> `procedure` *identifikátor* `(` *parametry* `);`
> *deklarace* lokálních typů, proměnných a procedur
> `begin`
> *příkazy* oddělené středníky
> `end;`

Zde *parametry* mají syntaxi `var` *jméno*`:`*typ*, kde *jméno* je identifikátor, jímž se lze na předaný parametr uvnitř procedury odkazovat. Pokud má procedura parametrů více, oddělují se středníky, jsou-li stejného typu, lze zkracovat, např.: `procedure X(var m,n:integer; var Z:array [1..n] of bit);`. Všechny deklarované objekty (parametry, typy, proměnné i procedury) existují pouze během volání této procedury, každá procedura vidí „své“ lokální proměnné a navíc lokální proměnné všech procedur, uvnitř kterých je deklarována (zastiňování se řídí stejnými pravidly jako v Pascalu nebo C).

Proměnné jsou pojmenovány identifikátory, musí se vytvořit deklarací `var` *identifikátor* `:` *typ*`;`. Při vstupu do procedury, v níž jsou deklarovány, mají nulovou hodnotu (v případě pole ji mají všechny jeho prvky) a než proměnná na konci procedury zanikne, musí být opět nulová. Deklaraci více proměnných téhož typu lze zkrátit, např. `var` i_1`,` i_2`, ...,` i_n `:` *typ*`;`.

Výrazy mohou obsahovat:

- *konstanty* (přirozená čísla a `maxword` reprezentující maximální dostupné číslo),
- *proměnné*,
- *prvky polí* (*pole*`[`*výraz*`]`),
- *číselné operace* (vstupem i výstupem jsou přirozená čísla) `+`, `-`, `*`, `div` (celá část podílu), `mod` (zbytek po dělení), `and`, `or`, `xor` (bitové operace viz definice o pár odstavců výše) a `not` (prohození nulových a jedničkových bitů), výsledky jsou automaticky modulo *maxword* + 1.

- *relační operace* (vstupem jsou dvě čísla, výstupem bitová hodnota `1`, když relace platí, `0` pokud nikoliv) `<`, `>`, `=`, `<=`, `>=` a `<>`,
- *závorky* (pokud nezávorkujeme, operátory mají své obvyklé priority).

Příkazy existují tyto:

- *Blok:* `begin` *příkazy* oddělené středníky `end` – způsobí vykonání všech příkazů, které obsahuje, v daném pořadí.
- *Modifikační příkazy: proměnná* `+=` *výraz* – způsobí vyhodnocení výrazu a přičtení jeho výsledku k dané proměnné (může to být rovněž prvek pole indexovaný nějakým výrazem). **Proměnná (resp. prvek pole), kterou příkaz modifikuje, se již nesmí nikde jinde v témže příkazu vyskytnout.** Analogicky příkazy `-=` a `^=`.
- *Prohazovací příkaz: proměnná* `=:=` *proměnná* – prohodí obsah dvou proměnných stejného typu. Pokud se jedná o prvky polí, nesmí se ve výrazech určujících indexy používat žádné z těchto polí.
- *Podmíněný příkaz:* `if` *podmínka* `then` $příkaz_1$ `else` $příkaz_2$ – vyhodnotí se *podmínka*, což je výraz s bitovým výsledkem, a pokud je roven jedné, vykoná se první z příkazů, jinak druhý. **Platnost podmínky musí zůstat po vykonání příkazu nezměněna.** Část `else` je možno vypustit, v případech typu `if x then if y then a else b` se pak `else` vztahuje vždy k nejbližšímu předchozímu ještě neukončenému příkazu `if`.
- *Příkaz cyklu:* `for var` *identifikátor* `=` d `to` h `do` *příkaz* – založí novou proměnnou daného jména a daný příkaz vykonává pro tuto proměnnou nabývající postupně hodnot $d, d+1, \ldots, h$, načež proměnnou opět zruší. Meze d a h jsou celočíselné výrazy, pokud $d > h$, příkaz se neprovede ani jednou. **Příkaz musí zachovávat hodnotu řídící proměnné, jakož i mezí cyklu (to znamená, že je může modifikovat, ale na konci jednoho průchodu cyklem musí mít obojí opět původní hodnotu).** Též je možno použít h `downto` d, tehdy cyklus běží pozpátku, tj. $h, h-1, \ldots, d$.
- *Volání procedury: procedura*`(`$parametr_1$`, ... ,` $parametr_n$`)` – zavolá proceduru se zadanými parametry, což mohou být buďto proměnné nebo indexovaná pole (výrazy v indexech ovšem musí mít po návratu z procedury stejnou hodnotu jako před jejím zavoláním) a jejich počet i typy musí odpovídat deklaraci procedury.
- *Příkaz obrácení výpočtu:* `undo` *příkaz* – provede daný příkaz pozpátku podle následujících pravidel:

`undo begin` p_1 `;` ... `;` p_n `end`	$\longrightarrow$	`begin undo` p_n `;` ... `;` `undo` p_1 `end`
`undo` x `+=` y	$\longrightarrow$	x `-=` y
`undo` x `-=` y	$\longrightarrow$	x `+=` y
`undo` x `^=` y	$\longrightarrow$	x `^=` y
`undo` x `=:=` y	$\longrightarrow$	x `=:=` y
`undo if` x `then` y `else` z	$\longrightarrow$	`if` x `then undo` y `else undo` z
`undo for` x `=` d `to` h `do` p	$\longrightarrow$	`for` x `=` h `downto` d `do undo` p
`undo` $P(x_1, \ldots, x_n)$	$\longrightarrow$	`undo` těla procedury (`begin` ... `end`)
`undo undo` p	$\longrightarrow$	p

  Konstrukce `begin` p `; undo` p `end` tedy nevykoná nic, ač může počítat poměrně dlouho.
- *Příkaz lokálního výpočtu:* `wrap` $příkaz_1$ `on` $příkaz_2$ je zkratkou za konstrukci `begin` $příkaz_1$ `;` $příkaz_2$ `; undo` $příkaz_1$ `end`.

Hlavní program nebudeme zavádět. Abychom se vyhnuli problémům se vstupy a výstupy, budeme vše vždy programovat jako procedury. Ty jako své parametry dostanou jak proměnné, které obsahují vstupní data, tak proměnné, které mají být předepsaným způsobem zmodifikovány podle výsledku.

Časová a prostorová složitost se definuje podobně jako v klasickém programování: časovou složitostí výpočtu je počet vykonaných příkazů modifikujících proměnné, ať již proběhly kterýmkoliv směrem. Množství paměti využité programem v nějakém okamžiku výpočtu spočítáme jako součet velikostí všech lokálních proměnných (typy `bit` a `word` mají jednotkovou velikost, pole má velikost rovnou součtu velikostí svých prvků) a parametrů (ty se všechny počítají jako jednotka, ať už jsou kteréhokoliv typu, protože jsou předávány odkazem) všech právě zavolaných procedur + jednotka navíc za každou takovou proceduru. Prostorovou složitostí programu nazveme pak maximum z využitého množství paměti přes celou dobu běhu programu. (Pozor, jelikož program je pro nás vždy procedurou, jeho vstupy a výstupy se do prostorové složitosti započítávají pouze jednotkově, i když to mohou být velká pole.)

Zbývá maličkost: cokoliv uzavřeného do složených závorek `{` a `}` je *komentářem*, který je počítačem zcela ignorován, jako kdyby na jeho místě byla mezera. Komentář nesmí uvnitř obsahovat složené závorky.

Příklad 1: Procedura pro prohození obsahu dvou proměnných (která ukazuje, že `=:=` se dá snadno odvodit pomocí ostatních operací). Časová i prostorová složitost jsou konstantní, tedy $O(1)$.

```
procedure Prohod(var x,y:word);
begin                  { x = X, y = Y  (X,Y jsou pův. hodnoty) }
   x ^= y;             { x = X xor Y, y = Y }
   y ^= x;             { x = X xor Y, y = Y xor (X xor Y) = X }
   x ^= y              { x = (X xor Y) xor X = Y, y = X }
end;
```

Příklad 2: Procedura pro výpočet maxima ze zadaných n čísel. Je dáno pole X celých čísel a proměnná max, k níž máme spočtené maximum přičíst. To dokážeme takto: Nejprve si předpočítáme do $M[i]$ maximum z čísel $X[1], \ldots, X[i]$, pak přičteme $M[n]$ k max a nakonec $M[i]$ opět vyprázdníme, což snadno zapíšeme pomocí příkazu `wrap`. Časová i prostorová složitost jsou $O(n)$, čili lineární.

```
procedure Maximum(var n:word; var X:array [1..n] of word; var max:word);
var M:array [0..n] of word;
begin
   wrap
      for var i=1 to n do
         if X[i]>M[i-1] then
            M[i] += X[i]
         else
            M[i] += M[i-1]
   on max += M[n];
end;
```

P-III-4 Poklad kapitána Flinta

Program: `poklad.pas` / `poklad.c` / `poklad.cpp`
Vstup: `poklad.in`
Výstup: `poklad.out`

Kapitán Flint si při svých pirátských výpravách přišel k docela pěkné hromádce zlaťáků. Pirátské výpravy jsou však dosti nejisté a štěstěna vrtkavá, a proto kapitán zakopal část svého jmění na pustém ostrově a cestu k pokladu zakreslil na ovčí kůži ve tvaru konvexního N-úhelníka. Celou mapu pak rozřezal na mnoho částí, přičemž každý řez vedl přímo mezi dvěma vrcholy mnohoúhelníka a žádné dva řezy se neprotínaly. Aby si pojistil věrnost posádky svého škuneru, rozhodl se Flint některé části darovat nejzdatnějším pirátům. Protože by se ale námořníci mohli snadno dohodnout, mapu sestavit a poklad vykopat, chce mezi ně kapitán rozdělit části mapy tak, aby žádní dva námořníci neměli sousední díly (tedy takové, které mají společnou hranu). Přitom chce mezi námořníky rozdělit co nejvíce částí mapy. Dokázali byste napsat program, který pomůže kapitánovi vyřešit jeho problém?

Vstup: Na prvním řádku vstupního souboru `poklad.in` dostane program dvě celá čísla N a M oddělená mezerou, $3 \leq N \leq 30\,000$, $0 \leq M \leq 30\,000$ – počet vrcholů mapy a počet řezů. Následuje M řádků popisujících jednotlivé řezy. Každý z těchto řádků obsahuje dvě čísla A a B oddělená mezerou – čísla vrcholů, mezi kterými vede řez (vrcholy číslujeme od jedné do N).

Výstup: Výstupní textový soubor `poklad.out` bude obsahovat jediné číslo udávající maximální počet částí mapy, které lze mezi námořníky rozdělit tak, aby žádní dva námořníci neměli sousední díly mapy.

Příklad:

Vstupní soubor `poklad.in`:
```
5 2
1 3
3 5
```

Výstupní soubor `poklad.out`:
```
2
```

P-III-5 Vážení

Program: `vahy.pas` / `vahy.c` / `vahy.cpp`
Vstup: `vahy.in`
Výstup: `vahy.out`

Mudrc Tlučhuba se přihlásil do konkurzu na královského rádce v jednom nejmenovaném království. Vzápětí však byl zaskočen podmínkami tohoto konkurzu: Jako test svých schopností obdrží N mincí, z nichž některé mají různou a některé stejnou hmotnost. Jeho úkolem bude tyto mince rozdělit do skupin tvořených mincemi stejné hmotnosti a tyto skupiny pak seřadit vzestupně podle hmotností mincí tak, aby v první skupině byly nejlehčí mince a v poslední skupině byly nejtěžší mince. Bude mít k dispozici dvouramenné váhy, na jejichž misky smí v jednom okamžiku položit po jedné minci.

Mudrc Tlučhuba vás požádal o pomoc při plnění tohoto úkolu. Chtěl by, abyste vytvořili program, jenž mu pomůže při rozhodování, které mince zvážit, jak mince rozdělit do skupin a jak vytvořené skupiny uspořádat. Pro účely programu si mince očíslujeme od 1 do N. Samotné vážení bude ve vašem programu zastoupeno funkcí `porovnej`.

Mudrc Tlučhuba musí úkol splnit v časovém limitu, který mu byl stanoven (tedy v něm musí úkol splnit i váš program). Kromě toho musí provést všechna *nezbytná* vážení, tj. nesmí existovat dvě či více možných řešení konzistentních s odpověďmi funkce `porovnej`, jinak by byl mudrc upálen jako čarodějník (jak jinak by mohl vědět, které uspořádání je správné?). Na druhou stranu nesmí být provedeno žádné *zbytečné* vážení, tj. takové, jehož výsledek by již (přímo či nepřímo) vyplýval z předchozích odpovědí funkce `porovnej`.

Popis funkce `porovnej`*:*

Funkce `porovnej` je definována v knihovně `vahy_lib`. Váš program musí obsahovat následující řádek, aby mohl používat funkci `porovnej`:

Pascal: `uses vahy_lib;`

C/C++: `#include "vahy_lib.h"`

Funkce `porovnej` je deklarována takto:

Pascal: `function porovnej(a,b: longint): integer;`

C/C++: `int porovnej(int, int);`

Tato funkce očekává jako vstupní parametry čísla dvou mincí. Vrátí hodnotu -1, pokud mince odpovídající prvnímu parametru je lehčí než mince odpovídající druhému parametru, $+1$, pokud je tomu naopak, a 0, pokud obě mince mají stejnou hmotnost.

Nezapomeňte, že váš program **nesmí** funkci `porovnej` volat zbytečně, tj. výsledek žádného volání funkce `porovnej` nesmí vyplývat (tedy být jednoznačně určen) z předchozích volání této funkce. Např. pokud jsme voláním funkce `porovnej` zjistili, že mince s číslem 1 je lehčí než mince s číslem 2 a že mince s číslem 2 je lehčí než mince s číslem 3, nelze již funkci `porovnej` zavolat s parametry 1 a 3. Kromě toho váš program může funkci `porovnej` zavolat nejvýše 250 000-krát.

Vstup: Vstupní soubor `vahy.in` obsahuje jediný řádek s jediným číslem N, $1 \leq N \leq 10\,000$, které udává počet mincí.

Výstup: Výstupní soubor `vahy.out` musí obsahovat K řádků, kde K je počet různých hmotností mincí. Na každém řádku budou uvedena čísla mincí téže hmotnosti v rostoucím pořadí. Hmotnosti mincí jednotlivých řádků tvoří rovněž rostoucí posloupnost, tzn. první řádek obsahuje všechny nejlehčí mince a poslední řádek všechny nejtěžší mince.

Příklad:

Vstupní soubor `vahy.in`:	Průběh komunikace:	Výstupní soubor `vahy.out`:
`4`	Volání `porovnej(2,4)` vrací `-1`.	`2`
	Volání `porovnej(1,2)` vrací `1`.	`1 3`
	Volání `porovnej(3,4)` vrací `-1`.	`4`
	Volání `porovnej(1,3)` vrací `0`.	