52. ročník matematické olympiády na středních školách

Kategorie P

In: 52. ročník matematické olympiády na středních školách. Zpráva o řešení úloh ze soutěže konané ve školním roce 2002/2003. 44. mezinárodní matematická olympiáda. 15. mezinárodní olympiáda v informatice. (Czech). Praha: Jednota českých matematiků a fyziků, 2004. pp. 97–145.

Persistent URL: http://dml.cz/dmlcz/405061

Kategorie P

Texty úloh

P – I – 1

Čajovník

Pan Nyi byl dvorním pěstitelem čaje císaře Tiang-tonga. Byl to pěstitel skutečně vyhlášený a jeho čajové lístky putovaly nejen do blízkých šálků císaře Tianga, ale i do dalekých zemí za oceánem. Tajemství Nyiho skvělého čaje spočívalo především v pečlivosti, s jakou se o své čajové keře staral. Nyi byl tak pečlivý, že si o každém svém keři vedl záznamy. Psal si dokonce i to, kolik větviček vychází z kterého místa keře. Po smrti pana Nyiho byly záznamy rozkradeny a jeho nástupce pan Myi tak měl práci o mnoho těžší. Rozhodl se proto, že záznamy získá zpět. Problémem ale je, že mnoho různých podvodníků mu nabízí záznamy falešné. Ty naštěstí většinou obsahují nesmyslné počty větvení, a tak se dají snadno odhalit. Pana Myiho neustálé ověřování pravosti záznamů už unavuje, a proto vás požádal, abyste mu napsali program, který mu s ověřováním pomůže.

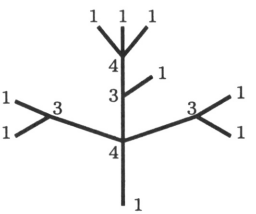
Váš program dostane na vstupu počet významných míst N na údajném čajovníku. Významným místem na čajovníku je buď místo, kde se čajovník větví, nebo místo, kde končí nějaká větev čajovníku. Protože žádné dvě větve čajovníku nemohou srůstat, nemohou vznikat „cykly“ z větví. Dále je na vstupu programu zadáno N kladných celých čísel $c_1, c_2, \ldots, c_N$, kde c_i určuje počet částí kmene, které vycházejí z i-tého významného místa. Na výstup program vypíše zprávu, zda může existovat čajovník, který bude mít takovéto počty větvení.

Formát vstupu: Vstupní textový soubor `caj.in` obsahuje dva řádky. Na prvním řádku je uvedeno jediné celé číslo N, $1 \leqq N \leqq 1\,000$. Druhý řádek obsahuje celá čísla $c_1, c_2, \ldots, c_N$ oddělená mezerami, $1 \leqq c_i \leqq N - 1$.

Formát výstupu: Výstupní textový soubor `caj.out` obsahuje jediný řádek tvořený buď slovem `EXISTUJE`, nebo slovem `NEEXISTUJE`.

Příklad 1 (obr.):

```
caj.in                          caj.out
    14                              EXISTUJE
    1 4 3 1 1 3 1 1 3 1 4 1 1 1
```



Příklad 2:

```
caj.in                          caj.out
    6                               NEEXISTUJE
    3 3 3 1 1 1
```

P – I – 2

Knihovna

Knihovnice Míla potřebuje objednat další skříň s poličkami do své knihovny; bohužel však sama neumí spočítat její optimální rozměry. Míla by ráda do nové skříně umístila 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 posloupnost N čísel v_i, $1 \leqq i \leqq N$, kde v_i je výška i-té knihy (uspořádáno podle rostoucích kódů). Pro zjednodušení můžete předpokládat, že všechny knihy mají stejnou tloušťku 1 cm. Váš program by měl ze zadaných údajů spočítat následující:

- Šířku skříně — označme ji s.
- Počet poliček ve skříni — označme ho p.
- Výšku w_i i-té poličky pro každé $1 \leqq i \leqq p$.
- Rozmístění knih do skříně se spočítanými parametry, které respektuje požadavky na pořadí knih zmíněné v zadání tohoto příkladu.

Navíc si knihovnice Míla přeje, aby skříň byla co nejužší a přitom aby se vešla do místnosti vysoké 250 cm. Rozmístění knih, které váš program nalezne, musí tedy ještě splňovat následující podmínky:

- Výška libovolné z knih umístěných do i-té poličky je nejvýše w_i.

▷ Součet tlouštěk knih umístěných do jedné poličky je nejvýše s cm, tj. tato polička obsahuje nejvýše s knih.

▷ Výška skříně, která je rovna $\sum_{i=1}^{p} w_i + (p+1) \cdot 1$ cm (předpokládáme, že šířka desek oddělujících poličky ve skříni je 1 cm), nesmí přesáhnout výšku místnosti 250 cm.

▷ s je nejmenší možné.

Příklad: Předpokládejme, že Míla chce do skříně umístit celkem 11 knih, jejichž výšky jsou v pořadí podle jejich kódů následující: 40 cm, 10 cm, 40 cm, 25 cm, 40 cm, 25 cm, 50 cm, 40 cm, 40 cm, 25 cm a 40 cm. Jedno z optimálních řešení by mohlo vypadat následovně: Skříň bude mít šířku pro 3 knihy a celkem 4 poličky s následujícími výškami: 40 cm, 40 cm, 50 cm a 40 cm. Výška skříně je v tomto případě 175 cm. Nalezení jednoho konkrétního možného umístění knih do skříně je snadné.

P – I – 3

Transformace

Jedna z metod zpracování textu používá následující transformační algoritmus:

Na vstupu mějme n-znakový řetězec $C = c_1 c_2 \ldots c_n$, jehož všechny znaky jsou navzájem různé. Řetězec $C' = c_{k+1} c_{k+2} \ldots c_n c_1 \ldots c_k$ nazýváme řetězcem C zrotovaným o k (tedy např. `eldat` je řetězec `datel` zrotovaný o 3). Vezměme si zadaný řetězec C a napišme si pod sebe C, C zrotovaný o 1, ..., C zrotovaný o $n-1$. Tím jsme získali tabulku n řetězců. Ty setřídíme v běžném lexikografickém pořadí (tzn. podle abecedy). Z výsledné tabulky si vybereme poslední sloupec S; dále si také zapamatujeme číslo řádku $\check{r}$, na němž se po setřídění nachází náš původní řetězec. Dvojice $(S, \check{r})$ je výsledek transformace zadaného vstupu. Jakkoli magicky to vypadá, tyto dva údaje stačí k rekonstrukci původního řetězce.

Příklad: Na vstupu máme slovo `datel`. Transformace probíhá takto:

```
datel              ateld
ateld              datel *
telda      ⟶      eldat
eldat              ldate
ldate              telda
```

Výsledkem tedy je slovo `dltea` a informace, že původně zadané slovo je na druhém řádku setříděné tabulky.

Soutěžní úloha. Program dostane na vstupu řetězec S délky n ($1 \leqq n \leqq 100$), jehož všechny znaky jsou navzájem různé (tj. je-li $S = s_1 s_2 \ldots s_n$, pak $s_i \neq s_j$ pro každá i, j, $i \neq j$), a číslo $\check{r}$ ($1 \leqq \check{r} \leqq n$). Úkolem je najít řetězec C takový, že dvojice $(S, \check{r})$ je výsledkem aplikace výše popsané transformace na řetězec C (máte zaručeno, že takový existuje).

Uvědomte si, že při použití v praxi se délky zpracovávaných vstupů pohybují řádově ve stovkách kilobytů; je tedy nevhodné, aby váš program měl kvadratické časové nebo paměťové nároky.

Formát vstupu: Na prvním řádku vstupního souboru `bw.in` se nachází řetězec S (řetězec neobsahuje mezery). Na druhém řádku je jedno celé číslo $\check{r}$.

Formát výstupu: Výstupní soubor `bw.out` je tvořen jedním řádkem obsahujícím řetězec C (jehož transformací je dvojice $(S, \check{r})$).

Příklad:

`bw.in`	`bw.out`
`dltea` `2`	`datel`

P – I – 4

Reverzibilní výpočty: Políčko pole

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: $5 \operatorname{xor} 15 = (0101)_2 \operatorname{xor} (1111)_2 = (1010)_2 = 10$. Obecně pro libovolná čísla x a y platí $x \operatorname{xor} y = y \operatorname{xor} x$, $x \operatorname{xor} x = 0$, $x \operatorname{xor} 0 = x$ a $(x \operatorname{xor} y) \operatorname{xor} z = x \operatorname{xor} (y \operatorname{xor} z)$. Podobně lze zavést operace *and* a *or*: $0 \operatorname{and} 0 = 0 \operatorname{and} 1 = 1 \operatorname{and} 0 = 0$, $1 \operatorname{and} 1 = 1$, $0 \operatorname{or} 0 = 0$, $0 \operatorname{or} 1 = 1 \operatorname{or} 0 = 1 \operatorname{or} 1 = 1$, 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, kterou příkaz modifikuje, (resp. prvek pole) 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 musejí mít po návratu z procedury stejnou hodnotu jako před jejím zavoláním) a jejich počet i typy musejí odpovídat deklaraci procedury.
- ▷ *Příkaz obrácení výpočtu:* `undo` *příkaz* — provede daný příkaz pozpátku podle následujících pravidel:

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

Soutěžní úloha. Napište *reverzibilní* proceduru `Najdi(var n:word; var X:array [1..n] of word; var co, kde:word)`. Tato procedura má za úkol v n-prvkovém poli X hledat hodnotu co, a pokud se tam tato hodnota vyskytuje, přičíst k proměnné *kde* pozici jejího výskytu, tedy i takové, že $X_i = co$. Navíc je známo, že pole X je uspořádáno vzestupně, tedy že pro každé $i < j$ platí $X_i < X_j$; proto také může být výskyt nejvýše jeden. Ve svém řešení se snažte dosáhnout co nejmenší časové i prostorové složitosti.

P – II – 1

Tajemný obraz

V Chile objevili archeologové tajemný obraz z předkolumbovské doby. Obraz vypadá jako několik bodů rozmístěných na pomyslné kružnici, přičemž každé dva body jsou spojeny rovnou čarou. Každá z čar je buď žlutá nebo červená. Archeologové dlouho hledali smysl této malby, ale žádný nemohli nalézt. Až amatérský archeolog a dobrodruh Erik von Kätzinken po dlouhém pátrání vyslovil hypotézu, že obrazec je poselstvím dávných Inků. Počet jednobarevných trojúhelníků s vrcholy na pomyslné kružnici prý udává počet dní od namalování obrazce, po nichž mají na Zemi opět přistát mimozemšťané. Protože obrazec je poměrně velký, rozhodl se Erik určit počet těchto jednobarevných trojúhelníků pomocí počítače.

Vaším úkolem je napsat program, který dostane na vstupu počet bodů nakreslených na obrazci N, $3 \leqq N$, a dále seznam dvojic čísel bodů (body si očíslujeme od jedné do N), které jsou propojeny červenou čarou (zbylé dvojice bodů jsou tedy propojeny žlutou). Na výstup program vypíše počet jednobarevných trojúhelníků v zadaném obrazci (tzn. takových

trojúhelníků, jejichž všechny tři vrcholy leží v bodech vyznačených na kružnici a jsou spojeny čarami téže barvy).

Příklad. V obrazci s pěti body a červenými čarami mezi body $(1,2)$, $(2,3)$, $(3,4)$ a $(2,4)$ jsou tři jednobarevné trojúhelníky (jeden červený a dva žluté). Červený trojúhelník má vrcholy v bodech $2,3,4$ a žluté trojúhelníky v bodech $1,3,5$ a $1,4,5$.

P – II – 2

Knihovna

Knihovnice Míla opět potřebuje objednat další skříň do své knihovny. Bohužel však zase sama neumí spočítat, jak by tato skříň měla být široká, a tak vás znovu poprosila o pomoc. Míla by ráda do nové skříně umístila celkem N knih, ale na rozdíl od úlohy P–I–2 z prvního kola je jí jedno, v jakém pořadí knihy do skříně umístí. Vstupem vašeho programu bude posloupnost N čísel v_i, $1 \leqq i \leqq N$, kde v_i je výška i-té knihy. Pro zjednodušení předpokládejme, že všechny knihy mají stejnou tloušťku — 1 cm.

Váš program by měl ze zadaných údajů spočítat následující:

- Šířku skříně — označme ji s.
- Počet poliček ve skříni — označme ho p.
- Výšku w_i i-té poličky pro každé $1 \leqq i \leqq p$.
- Rozmístění knih do skříně se spočítanými parametry.

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

- Výška libovolné z knih umístěných do i-té poličky je nejvýše w_i.
- Součet tlouštěk knih umístěných do jedné poličky je nejvýše s cm, tj. tato polička obsahuje nejvýše s knih.
- Výška skříně, která je rovna $\sum_{i=1}^{p} w_i + (p+1) \cdot 1$ cm (předpokládáme, že šířka desek oddělujících poličky ve skříni je 1 cm), nesmí přesáhnout výšku místnosti 250 cm.
- s je nejmenší možné.

Příklad. Předpokládejme, že Míla chce do skříně umístit celkem 14 knih, z nichž devět má výšku 50 cm a pět má výšku 40 cm. Jedno z optimálních řešení by mohlo vypadat následovně: Skříň bude mít šířku pro 3 knihy a celkem 5 poliček — tři z nich budou mít výšku 50 cm a dvě poličky výšku 40 cm. Nalézt jedno konkrétní možné rozmístění knih do skříně je snadné.

P – II – 3

Transformace

Jedna z metod zpracování textu používá následující transformační algoritmus: Na vstupu mějme n-znakový řetězec $C = c_1 c_2 \ldots c_n$. Řetězec $C' = c_{k+1} c_{k+2} \ldots c_n c_1 \ldots c_k$ nazýváme řetězcem C zrotovaným o k (tedy např. `akaabr` je řetězec `abraka` zrotovaný o 3). Vezměme si zadaný řetězec C a napišme si pod sebe C, C zrotovaný o 1, ..., C zrotovaný o $n-1$. Tím jsme získali tabulku n řetězců. Ty setřídíme v běžném lexikografickém pořadí (tzn. podle abecedy). Z výsledné tabulky si vybereme poslední sloupec S; dále si také zapamatujeme číslo řádku $\check{r}$, na němž se po setřídění nachází náš původní řetězec (je-li těchto řádků více, libovolný z nich). Dvojice $(S, \check{r})$ je výsledek transformace zadaného vstupu. Jakkoli magicky to vypadá, tyto dva údaje stačí k rekonstrukci původního řetězce.

Příklad. Na vstupu máme slovo `abraka`. Transformace probíhá takto:

```
abraka              aabrak
brakaa              abraka *
rakaab      ⟶       akaabr
akaabr              brakaa
kaabra              kaabra
aabrak              rakaab
```

Výsledkem tedy je slovo `karaab` a informace, že původně zadané slovo je na druhém řádku setříděné tabulky.

Soutěžní úloha. Program dostane na vstupu řetězec S délky n ($1 \leqq n \leqq 10\,000$) a číslo $\check{r}$ ($1 \leqq \check{r} \leqq n$). Úkolem je najít řetězec C takový, že dvojice $(S, \check{r})$ je výsledkem aplikace výše popsané transformace na řetězec C (máte zaručeno, že takový existuje).

Poznámka. Budete-li psát program v Pascalu, můžete předpokládat, že do stringu se řetězec této délky vejde.

Uvědomte si, že při použití v praxi se délky zpracovávaných vstupů pohybují řádově ve stovkách kilobytů; je tedy nevhodné, aby váš program měl kvadratické časové nebo paměťové nároky.

Formát vstupu: Na prvním řádku vstupního souboru `bw.in` se nachází řetězec S (řetězec neobsahuje mezery). Na druhém řádku je jedno celé číslo $\check{r}$.

Formát výstupu: Výstupní soubor `bw.out` je tvořen jedním řádkem, obsahujícím řetězec C (jehož transformací je dvojice $(S, \check{r})$).

Příklad: Vstupní soubor `bw.in`:

```
karaab
2
```

Výstupní soubor `bw.out`:

```
abraka
```

P – II – 4

Reverzibilní výpočty: Ouřad

(Definice reverzibilních výpočtů je obsažena v textu úlohy P–I–4.)

Oblastní Ouřad sídlí v městě $\mathcal{M}$ v n budovách. Aby si ouředníci sídlící na různých místech mohli rychle posílat všechny ty dopisy, přípisy, zápisy, dobropisy, vrubopisy, tiskopisy a vůbec všelijaké spisy s podpisy, vybudovali si potrubní poštu — systém rour spojujících některé dvojice budov. Těmito rourami se pomocí stlačeného vzduchu posílají zásilky. Aby nedocházelo ke kolizím, je každá roura využívána jenom jedním směrem. Nejsou si ale jisti, jestli již postavili všechna potřebná potrubí, a tak je zajímá, jak zjistit, zda mezi nějakými dvěma zadanými místy je možné dopravit zásilku, a to buďto přímo, nebo s přeložením zásilky v nějaké mezistanici, případně mezistanicích.

Napište *reverzibilní* proceduru `Zkoumej(var n:word; var A:array [1..n] of array [1..n] of bit; var x,y,d:word)`, která dostane jako vstup počet budov n, matici A popisující v prvku $A[i][j]$, zda vede (`1`) či nevede (`0`) roura z i-té do j-té budovy, a čísla budov x a y. Poté do proměnné d přičte, s jakým nejmenším počtem přeložení je možno přepravit zásilku z budovy x do budovy y, případně přičte číslo větší než n, pokud to možné není. Snažte se dosáhnout co nejmenší *prostorové* složitosti výpočtu při zachování polynomiální časové složitosti.

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 N odměněných dětí 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, $1 \leqq i \leqq N$, kde t_i je tloušťka i-té knihy. Můžete předpokládat, že tloušťka t_i i-té knihy je celé číslo z rozmezí od 1 do 50. 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.
- ▷ 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, 20, 7, 6, 2 a 4. Minimální možná šířka skříně v tomto případě je 20 — 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 obsažena v textu úlohy P–I–4.)

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 své procedury.

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-úhelníku. Celou mapu pak rozřezal na mnoho částí, přičemž každý řez vedl přímo mezi dvěma vrcholy mnohoúhelníku 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, $3 \leqq N \leqq 30\,000$, $0 \leqq M \leqq 30\,000$ — 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`:	Výstupní soubor `poklad.out`:

`5 2`
`1 3`
`3 5`
	`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 000krát.

Vstup: Vstupní soubor `vahy.in` obsahuje jediný řádek s jediným číslem N, $1 \leqq N \leqq 10\,000$, 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`:

```
4
```

Průběh komunikace:

volání `porovnej(2,4)` vrací `-1`
volání `porovnej(1,2)` vrací `1`
volání `porovnej(3,4)` vrací `-1`
volání `porovnej(1,3)` vrací `0`

Výstupní soubor `vahy.out`:

```
2
1 3
4
```

P – I – 1

Pro řešení úlohy si vypůjčíme terminologii z teorie grafů. Významná místa na čajovníku budeme nazývat *vrcholy*, části kmene čajovníku mezi dvěma významnými místy pak *hrany*. Vrcholy spolu s hranami si pojmenujeme *strom* (pokud bychom chtěli být opravdu přesní, měli bychom jej nazvat *grafem*. My ale víme, že náš čajovník nemá žádný cyklus a graf s touto vlastností se nazývá strom). Počet hran, které vedou z nějakého vrcholu v (tedy vlastně počet částí kmene, které vedou z významného místa v), nazveme *stupněm* vrcholu v.

Nejdříve si všimneme, že každý strom s alespoň dvěma vrcholy má alespoň jeden vrchol stupně jedna (takovýto vrchol se nazývá *list*). Tento vrchol můžeme snadno nalézt následovně. Začneme strom prohledávat v libovolném vrcholu. Pokud ještě nejsme v listu, přejdeme do libovolného sousedního (tzn. připojeného hranou) vrcholu, ve kterém jsme dosud nebyli. Jelikož ve stromu nejsou cykly, musí takovýto vrchol vždy existovat. Protože vrcholů je konečný počet, musíme jednou skončit — a to můžeme pouze v listu.

Nyní ukážeme, že součet stupňů všech vrcholů v libovolném stromu je $2N-2$ (kde N je počet vrcholů). Naopak platí, že máme-li N kladných celých čísel se součtem $2N-2$, pak existuje strom s N vrcholy majícími tyto stupně. Z toho už je jasné, že stačí zjistit, zda je součet čísel na vstupu roven $2N-2$, a podle výsledku vypsat patřičnou zprávu.

První tvrzení dokážeme indukcí podle počtu vrcholů. Strom o dvou vrcholech obsahuje jedinou hranu. Součet stupňů vrcholů je tedy $1+1=2$ a naše tvrzení platí. Pokud má strom více vrcholů, víme z předchozího pozorování, že má list. Když tento list odebereme (tedy zrušíme vrchol a hranu, která ho připojuje ke zbytku stromu), získáme zřejmě opět strom. Pro něj z indukčního předpokladu platí, že součet stupňů je $2\cdot(N-1)-2=2N-4$. Protože v původním stromu měl jeden vrchol stupeň o jedna vyšší (ten, ke kterému byl připojen list) a byl v něm navíc list, je součet stupňů v původním stromu $2N-4+2=2N-2$. Tím je první tvrzení dokázáno.

Druhé tvrzení dokážeme indukcí dle počtu členů posloupnosti: Nechť máme posloupnost dvou kladných celých čísel, jejichž součet je $2\cdot 2-2=2$. Tato čísla tedy mohou být pouze dvě jedničky. Pro ně jsou zřejmě odpovídajícím stromem dva vrcholy spojené hranou. Pokud má posloupnost

více než dvě čísla, musí zřejmě obsahovat alespoň jednu jedničku (jinak by součet N čísel byl alespoň $2N$ a ne $2N-2$). Analogicky musí také obsahovat alespoň jedno číslo větší než jedna. Když z posloupnosti vypustíme jednu jedničku a jedno z čísel větších než jedna snížíme o jedna, získáme posloupnost čísel o jedna kratší se součtem $2N-2-2 = 2 \cdot (N-1) - 2$. Dle indukčního předpokladu tedy existuje strom na $N-1$ vrcholech s příslušnými stupni vrcholů. Když do stromu přidáme jeden list a připojíme ho hranou k vrcholu, který odpovídá číslu, jež jsme zmenšovali o jedničku, získáme přesně strom pro naši původní posloupnost. Tím je dokázáno i druhé tvrzení.

Časová složitost algoritmu je $O(N)$, paměťová $O(1)$.

```
program Caj;
var
  N, i, Suma, Casti : Integer;
  vstup, vystup : Text;
begin
  Assign(vstup, 'caj.in');
  Assign(vystup, 'caj.out');
  Reset(vstup);
  Rewrite(vystup);
  Suma := 0;
  ReadLn(vstup, N);
  for i := 1 to N do begin
    Read(vstup, Casti);
    Suma := Suma + Casti;
  end;
  if Suma = 2*(N-1) then
    WriteLn(vystup, 'EXISTUJE')
  else
    WriteLn(vystup, 'NEEXISTUJE');
  Close(vstup);
  Close(vystup);
end.
```

P – I – 2

Předvedeme si dvě možná řešení této úlohy. Obě jsou založena na metodě zvané dynamické programování: Úloha se nejprve vyřeší pro podúlohu velikosti 1. Tohoto řešení se použije pro nalezení řešení podúlohy velikosti 2. Takto nalezených řešení se použije pro vyřešení podúlohy velikosti 3 atd. V našem případě bude velikost podúlohy určená počtem knih, které chceme do skříně umístit.

První řešení je založeno na vytvoření dvojrozměrného pole A o rozměrech $N \times V$, kde N je celkový počet knih, které máme do skříně umístit, a V je maximální výška skříně; V je v našem případě rovno 250 podle

zadání úlohy. Hodnota $A[i,j], 0 \leqq i \leqq N, 1 \leqq j \leqq V$, udává minimální možnou šířku skříně výšky j, do které lze umístit prvních i knih. Pokud do skříně výšky j prvních i knih nelze umístit, tj. některá z těchto knih je vyšší než $j - 2\,\text{cm}$, pak je hodnota $A[i,j]$ rovna nějaké speciální hodnotě, např. -1. Popíšeme si, jak lze v čase $O(N)$ spočítat hodnotu $A[i_0, j_0]$, máme-li již spočítány hodnoty $A[i,j]$ pro $i < i_0$. Pokud je $i_0 = 0$, pak zřejmě $A[i_0, j_0] = 0\,\text{cm}$. Pokud existuje i, $1 \leqq i \leqq i_0$, takové, že výška v_i i-té knihy je větší než $j_0 - 2\,\text{cm}$, pak prvních i knih nelze do skříně výšky j_0 umístit a $A[i_0, j_0]$ bude rovno -1. Ve zbylých případech určíme hodnotu $A[i_0, j_0]$ následovně: Pro $0 \leqq i < i_0$ zkusíme umístit na poslední poličku skříně $(i+1)$-ní až i_0-tou knihu a prvních i knih dáme na předcházející poličky; výška poslední poličky by tedy musela být alespoň $v = \max_{i+1 \leqq k \leqq i_0} v_k$ a můžeme předpokládat, že je právě v. Šířka této poličky musí být alespoň $i_0 - i$. Pokud $A[i, j_0 - v - 1]$ je rovno -1, pak nelze vytvořit skříň výšky j_0, která by obsahovala prvních i_0 knih a na poslední poličce by z nich měla posledních $i_0 - i$. V opačném případě je nejmenší šířka skříně výšky j_0, která obsahuje prvních i_0 knih a na poslední poličce má z nich umístěno posledních $i_0 - i$, rovna $\max\{A[i, j_0 - v - 1], i_0 - i\}$. Nejmenší z těchto výrazů pro $0 \leqq i < i_0$ bude roven hledané hodnotě $A[i_0, j_0]$. Výše popsaný výpočet lze provést v čase $O(N)$, budeme-li postupovat od $i = i_0 - 1$ k $i = 0$; v takovém případě lze $v = \max_{i+1 \leqq k \leqq i_0} v_k$ spočítat z v pro hodnotu i o 1 větší v konstantním čase. Hodnota pole $A[N, 250]$ je hledanou minimální možnou šířkou skříně. Pokud chceme zároveň nalézt i rozmístění knih do skříně a výšky jednotlivých poliček, zavedeme si ještě pomocné pole $B[i,j]$, $0 \leqq i \leqq N$, $1 \leqq j \leqq V$, do jehož položky $B[i_0, j_0]$ si při výpočtu hodnoty $A[i_0, j_0]$ uložíme to i, pro které je šířka skříně minimální při výšce j_0. Z hodnoty $B[N, 250]$ určíme počet knih, které jsou v optimálním řešení na poslední poličce; tato hodnota nám umožní spočítat výšku skříně bez poslední poličky a počet knih v těchto poličkách. Z příslušné hodnoty v poli B určíme počet knih na předposlední poličce a takto postupujme, dokud nedosáhneme první poličky. Vzhledem k velikosti pole A jsme si právě popsali algoritmus, jehož časová složitost je $O(VN^2)$ a paměťová složitost $O(VN)$.

Nyní si popíšeme druhé možné řešení. Nejprve si ukážeme, jak lze v čase $O(N^2)$ rozhodnout, zda lze knihy umístit do skříně šířky s a výšky V. K tomu si vytvoříme pomocné pole $A[i]$, $0 \leqq i \leqq N$, které udává minimální výšku skříně šířky s, do které lze umístit prvních i knih. Pokud $A[N] > V$, pak knihy nelze umístit do skříně šířky s a výšky V; v opačném případě je lze do skříně s těmito rozměry umís-

tit. K určení hodnot v poli A opět použijeme dynamické programování. Hodnota $A[0]$ je 1 cm, což je speciální případ obecného vztahu „součet výšek poliček + (počet poliček + 1) × 1 cm“ pro výšku skříně. Popíšeme, jak lze určit hodnotu $A[i_0]$, pokud známe hodnoty $A[i]$ pro $0 \leqq i < i_0$. Zvolme $i_0 - s \leqq i < i_0$; na poslední poličku chceme umístit v takovémto případě posledních $i_0 - i$ knih (proto podmínka $i_0 - s \leqq i$). Výška skříně je pak rovna $A[i] + 1 + \max_{i<k\leqq i_0} v_k$; nejmenší z těchto výrazů pro i, $i_0 - s \leqq i < i_0$, je hledaná hodnota $A[i_0]$. Hodnotu $A[i_0]$ lze spočítat v čase $O(N)$, pokud budeme postupovat od $i = i_0 - 1$ k $i = i_0 - s$ (potom lze $\max_{i<k\leqq i_0} v_k$ spočítat z hodnoty pro $i + 1$ v konstantním čase). Popsaná procedura v čase $O(N^2)$ s pamětí $O(N)$ rozhodne, zda lze zadaných N knih umístit do skříně šířky s a výšky V. Zbývá popsat, jak lze tuto proceduru použít pro vyřešení původní úlohy. Nejprve zkontrolujeme, že výška všech knih je nejvýše $V - 2\,\text{cm} = 248\,\text{cm}$, a tedy že knihy lze vůbec umístit do nějaké skříně výšky V. K určení minimální šířky s_0 skříně použijeme metodu zvanou půlení intervalu. Budeme si udržovat dvě proměnné $s_1 \leqq s_2$, které nám budou ohraničovat možný interval, ve kterém je hledaná šířka s_0, tj. $s_1 \leqq s_0 \leqq s_2$. Nejprve položíme $s_1 = 1$ a $s_2 = N$. V každém kroku zvolíme $s = \lfloor \frac{1}{2}(s_1 + s_2) \rfloor$ a pomocí výše popsané procedury zkontrolujeme, zda lze našich N knih umístit do skříně výšky V a šířky s. Pokud knihy lze do takové skříně umístit, položíme $s_2 = s$; v opačném případě položíme $s_1 = s + 1$. Celý postup opakujeme, dokud se hodnoty s_1 a s_2 liší, tj. dokud nenalezneme hledanou hodnotu s_0. Všimněte si, že v každém kroku se rozdíl $s_2 - s_1$ zmenší alespoň o 1 (kdybychom při volbě s použili horní celou část místo dolní celé části, nebylo by toto tvrzení pravdivé) a tento rozdíl se zmenší zhruba na polovinu. Tedy po $O(\log N)$ krocích nalezneme hledanou optimální šířku skříně s_0. Výšky poliček a rozmístění knih lze nalézt podobně jako v předcházejícím algoritmu zavedením pomocného pole B, do kterého si budeme ukládat počet knih na poslední poličce v optimálním řešení. Celková časová složitost právě popsaného algoritmu je tedy $O(N^2 \log N)$ a paměťová složitost je $O(N)$.

Zbývá vyřešit otázku, který ze dvou popsaných algoritmů je lepší. Odpověď je, že ani jeden není lepší. Vzhledem k zadání úlohy, kde V je omezeno, je časová složitost prvního algoritmu sice $O(N^2)$ a paměťová pouze $O(N)$, ale multiplikativní konstanta skrytá ve „velkém O“ je lineární s V; na druhou stranu paměťová složitost druhého algoritmu je pouze $O(N)$, kde multiplikativní konstanta je nezávislá na výšce. Dle výše popsaného postupu dokonce druhý algoritmus pracuje s poli, která jsou

250krát menší. Stejně v časové složitosti člen $\log N$ bude menší než člen V vyskytující se v časové složitosti prvního algoritmu. První algoritmus je tedy pro omezenou výšku V asymptoticky lepší, ale ve skutečnosti bude lepší než druhý popsaný algoritmus až pro velmi velké hodnoty N. Lze tedy říci, že druhý algoritmus je použitelnější.

```
program p_1_2;
{ Řešení úlohy P-I-2 verze 1 }
const MAXN=100;
      VYSKA_MISTNOSTI=250;
var vyska: array[1..MAXN] of word;  { výšky knih }
    n: word;      { počet knih }
    A: array[0..MAXN,1..VYSKA_MISTNOSTI] of integer;
                  { pole minimálních šířek skříně }
    B: array[0..MAXN,1..VYSKA_MISTNOSTI] of word;
                  { počty knih na poslední poličce v optimálním řešení }
function max(a,b:longint):longint;
  begin
    if a<b then max:=b else max:=a
  end;
procedure vypis(n: word; v: word);
  var i,k:word;
  begin
    if n=0 then
      begin
        writeln('Výška skříně: ',VYSKA_MISTNOSTI-v+1,' cm');
        exit;
      end;
    k:=0;
    for i:=n-B[n,v]+1 to n do k:=max(k,vyska[i]);
    vypis(n-B[n,v],v-k-1);
    writeln;
    writeln('Výška poličky: ',k,' cm');
    write('Knihy na poličce:');
    for i:=n-B[n,v]+1 to n do write(' ',i,'(',vyska[i],' cm)');
    writeln;
  end;
var i,j,k: word;
    maxvyska: word;
begin
  readln(n);
  for i:=1 to n do read(vyska[i]);
  for i:=1 to n do
    if vyska[i]>VYSKA_MISTNOSTI-2 then
      begin
        writeln('Pro zadané rozměry knih neexistuje knihovna!');
        halt;
      end;
  for j:=1 to VYSKA_MISTNOSTI do A[0,j]:=0;
  for i:=1 to n do
    for j:=1 to VYSKA_MISTNOSTI do
      begin
        maxvyska:=vyska[i];
        A[i,j]:=-1;
```

```
        for k:=1 to i do
          begin
            maxvyska:=max(maxvyska,vyska[i-k+1]);
            if maxvyska+2>j then break;
            if A[i-k,j-maxvyska-1]=-1 then continue;
            if (A[i,j]=-1) or (A[i,j]>max(A[i-k,j-maxvyska-1],k)) then
              begin
                A[i,j]:=max(A[i-k,j-maxvyska-1],k);
                B[i,j]:=k;
              end;
          end;
      end;
      end;
  if A[n,VYSKA_MISTNOSTI]=-1 then
    begin
      writeln('Pro zadané knihy nelze knihovnu navrhnout.');
      halt;
    end;
  writeln('Optimální šířka skříně je ',A[n,VYSKA_MISTNOSTI],' cm.');
  vypis(n,VYSKA_MISTNOSTI);
end.

program p_1_2;
{ Řešení úlohy P-I-2 verze 2 }
const MAXN=1000;
      VYSKA_MISTNOSTI=250;
var vyska: array[1..MAXN] of word; { výšky knih }
    n: word;                       { počet knih }
function lze_skrin(s: word; v: word; vypisovat: boolean):boolean;
  var A: array[0..MAXN] of word;   { pole s minimálními výškami knihoven }
      B: array[1..MAXN] of word;   { pole s počty knih na poličkách }
      maxvyska: word;
      i, j: word;
  begin
    A[0]:=1;
    for i:=1 to n do
      begin
        maxvyska:=vyska[i];
        A[i]:=A[i-1]+maxvyska+1;
        B[i]:=1;
        j:=i-1;
        while (j>0) and (i-j<s) do
          begin
            if maxvyska<vyska[j] then maxvyska:=vyska[j];
            if A[i]>A[j-1]+maxvyska+1 then
              begin
                A[i]:=A[j-1]+maxvyska+1;
                B[i]:=i-j+1;
              end;
            dec(j);
          end
      end;
    lze_skrin:= A[n] <= v;
    if not(vypisovat) then exit;
    i:=n;
    writeln('Výška skříně: ',A[n],' cm.');
```

```
    writeln('Výšky poliček ve skříni a jejich naplnění knihami ',
            'od spodu knihovny:');
    while i>0 do
      begin
        writeln;
        writeln('Výška poličky: ',A[i]-A[i-B[i]]-1,' cm');
        write('Knihy v poličce:');
        for j:=i-B[i]+1 to i do
          write(' ',j,'(',vyska[j],' cm)');
        writeln;
        i:=i-B[i];
      end
  end;
var i:word;
    s1,s2:word;
begin
  readln(n);
  for i:=1 to n do read(vyska[i]);
  for i:=1 to n do
    if vyska[i]>VYSKA_MISTNOSTI-2 then
      begin
        writeln('Pro zadané rozměry knih neexistuje knihovna!');
        halt;
      end;
  s1:=1; s2:=n;
  while s1<s2 do
    if lze_skrin((s1+s2) div 2, VYSKA_MISTNOSTI, false) then
      s2:=(s1+s2) div 2
    else
      s1:=(s1+s2) div 2+1;
  writeln('Optimální šířka skříně je ',s1,' cm.');
  lze_skrin(s1,VYSKA_MISTNOSTI,true);
end.
```

P – I – 3

Máme zadán poslední sloupec setříděné tabulky. Základní myšlenka celého řešení spočívá v tom, že tímto je dán i první sloupec — stačí setřídit písmena posledního sloupce podle abecedy. Nyní využijeme toho, že jednotlivé řádky tabulky vznikly rotací nějakého řetězce; tedy je-li na některém řádku v prvním sloupci písmeno x a v posledním sloupci písmeno y, znamená to, že v původním řetězci bylo písmeno x za písmenem y (bráno cyklicky — tedy za posledním písmenem následuje první). Vzhledem k tomu, že každé písmeno se v řetězci vyskytuje nejvýše jednou, n řádků tabulky nám již určuje pořadí písmen v původním řetězci až na rotaci. Navíc máme zadáno, na kterém řádku se vyskytuje námi hledané slovo; tím máme určeno jeho první písmeno.

Sestrojit na základě této myšlenky algoritmus je již jednoduché. Písmena zadaného řetězce si setřídíme podle abecedy (vzhledem k tomu, že hodnoty jsou z omezeného rozsahu, nabízí se přihrádkové třídění) a vyrobíme si tabulku, v níž bude každému písmenu přiřazeno jemu odpovídající následující písmeno. Pak začneme od písmene, o němž víme, že je první, a postupujeme od něj po následnících, přičemž rovnou vypisujeme výsledek, dokud se k tomuto písmeni nevrátíme.

Časová i paměťová složitost algoritmu jsou zjevně lineární v délce zadaného řetězce.

```
program bw;
const MAX = 100;
type slovo = array[1..MAX] of char;
var prvni_sloupec, posledni_sloupec : slovo;
    radek : integer;
    delka : integer;
    buckets : array[char] of boolean;
    naslednik : array[char] of char;
    s : string;
    i, l : integer;
    ch : char;

begin
  readln (s);                                  { načtení zadání }
  delka := length (s);
  for i := 1 to delka do
    posledni_sloupec[i] := s[i];
  readln (radek);

  for ch := #0 to #255 do                      { bucket sort }
    buckets[ch]:=false;
  for i := 1 to delka do
    buckets[posledni_sloupec[i]]:=true;
  l := 0;
  for ch := #0 to #255 do
    if buckets[ch] then
      begin
        inc (l);
        prvni_sloupec[l] := ch;
      end;

  for i:=1 to delka do                         { určení následníků }
    naslednik[posledni_sloupec[i]] := prvni_sloupec[i];

  ch := prvni_sloupec[radek];                  { výpis }
  for i := 1 to delka do
    begin
      write (ch);
      ch := naslednik[ch];
    end;
  writeln;
end.
```

Políčit na pozici příslušného prvku prohledáváním pole políčko po políčku přinese požadovanou proceduru, přesto poněkud pomalou. Pokračujme proto, přátelé, v přemýšlení:

Sestrojíme reverzibilní verzi binárního vyhledávání, místo tradičního zápisu pomocí cyklu `while` ovšem použijeme rekurzi. Zavedeme si podproceduru `Hledej(var l,p:word)`, která bude vyhledávat hodnotu *co* v úseku $X_l, X_{l+1}, \ldots, X_p$ a výsledek přičte k proměnné *kde*. Zařídí to tak, že si nejdříve spočte pozici prostředního prvku X_m zadaného úseku (pokud má úsek sudou délku, zaokrouhlíme libovolným směrem) a podle jeho hodnoty zjistí, ve které polovině úseku má hledání pokračovat: pokud $X_m < co$, pak od $m+1$ do r, je-li $X_m > co$, tak od l do $m-1$. Na tento úsek pak zavoláme tutéž proceduru rekurzivně, ale nesmíme zapomenout, až se vrátí, m ještě odpočítat. Nastane-li kdykoliv při porovnávání rovnost, právě jsme hodnotu *co* našli a po zvýšení *kde* o m se z procedury vracíme. Dospějeme-li v rekurzi k úseku nulové délky ($r < l$), vracíme se s prázdnou, tedy aniž bychom *kde* jakkoliv měnili.

Podle tohoto algoritmu již snadno vytvoříme program, pro odpočítávání v něm použijeme příkaz `wrap`:

```
procedure Najdi(var n:word; var X:array [1..n] of word; var co,kde:word);
   var one:word;
   procedure Hledej(var l,r:word);
   var m:word;
   begin
      if l<=r then                          { úsek není prázdný }
         wrap m += (l+r) div 2              { spočteme střed }
         on
            if X[m]=co then                 { našli jsme }
               kde += m
            else if X[m]<co then begin      { postoupíme do pravého úseku }
               wrap m += 1
               on Hledej(m,r)
               end
            else begin                      { do levého }
               wrap m -= 1
               on Hledej(l,m)
               end
   end;
begin
   wrap one += 1
   on Hledej(one,n)
end;
```

Postoupíme-li v rekurzi o úroveň hlouběji, zmenší se prohledávaný úsek minimálně o polovinu, takže po nejvýše $\lceil \log_2 n \rceil$ rekurzivních vo-

láních buďto hledanou hodnotu odhalíme, nebo dospějeme k úseku nulové délky, kde rekurze rovněž končí. Časová složitost tedy činí $O(\log n)$ a paměťová taktéž (pro každou úroveň rekurze spotřebujeme konstantní množství paměti).

P – II – 1

Řešení úlohy nám velmi usnadní následující trik. Nebudeme určovat počet jednobarevných trojúhelníků, ale počet dvoubarevných trojúhelníků. Požadovaný výsledek pak snadno získáme tak, že od počtu všech trojúhelníků s vrcholy v bodech obrazce (těch je $\frac{1}{6}N(N-1)(N-2)$) odečteme počet dvoubarevných.

Počet dvoubarevných trojúhelníků určíme následovně: Uvažujme nějaký bod v, z něhož vede k_v červených hran a $N-1-k_v$ hran žlutých. Celkem je tento bod součástí $\frac{1}{2}(N-1)(N-2)$ trojúhelníků (počet způsobů, jak zvolit zbylé dva vrcholy trojúhelníku) a z nich je $k_v(N-1-k_v)$ zaručeně dvoubarevných — to jsou ty, u nichž jsme zvolili jeden vrchol připojený červenou a druhý žlutou čarou. Když sečteme tyto počty zaručeně dvoubarevných trojúhelníků přes všechny vrcholy, dostaneme dvojnásobek počtu dvoubarevných trojúhelníků (každý z dvoubarevných trojúhelníků jsme totiž započítali právě u dvou bodů).

Z výše uvedeného rozboru je algoritmus již zřejmý. Nejdříve si u každého bodu spočítáme červené čáry. Potom podle uvedeného postupu určíme počet všech dvoubarevných trojúhelníků a následně již snadno dopočítáme i počet jednobarevných trojúhelníků. Algoritmus má časovou složitost $O(M+N)$, kde M je počet červených čar, a paměťovou složitost $O(N)$.

```
program obraz;
const
  MAXN = 100;
var
  Cervene : Array[1..MAXN] of Integer;  {Počet červených čar z bodů}
  N : Integer;                          {Počet bodů}

{Načtení vstupu do pole}
procedure Nacti;
var
  i : Integer;
  A, B : Integer;   {Konce načítané spojnice}
begin
  Write('Pocet bodu: ');
  ReadLn(N);
  for i := 1 to N do
    Cervene[i] := 0;
```

```
  while true do begin
    Write('Pocatek cervene cary: ');
    ReadLn(A);
    if A = 0 then {Konec?}
      break;
    Write('Konec cervene cary: ');
    ReadLn(B);
    Inc(Cervene[A]);
    Inc(Cervene[B]);
  end;
end;

{Spočte jednobarevné trojúhelníky}
function Spocti : Integer;
var
  i : Integer;
  Dvoj : Integer;   {Počet dvoubarevných trojúhelníků}
begin
  Dvoj := 0;
  for i := 1 to N do
    Dvoj := Dvoj + Cervene[i]*(N-1-Cervene[i]);
  Dvoj := Dvoj div 2;
  Spocti := N*(N-1)*(N-2) div 6 - Dvoj;
end;

begin
  Nacti;
  WriteLn('Pocet jednobarevnych trojuhelniku: ', Spocti);
end.
```

P – II – 2

Nejprve učiňme následující pozorování: Nechť s_0 je šířka optimální skříně a nechť má tato skříň p poliček. Potom existují výšky $w_1 \geqq \ldots \geqq w_p$ poliček a rozmístění knih do skříně se šířkou s_0 a poličkami výšky $w_1, \ldots, w_p$ takové, že výšky knih v této skříni v pořadí seshora dolů a v každé poličce zleva doprava tvoří nerostoucí posloupnost (první polička je ta nejvýše umístěná).

První část pozorování, o existenci výšek $w_1 \geqq \ldots \geqq w_p$, je jednoduchá — pokud výšky poliček ve skříni seshora dolů netvoří nerostoucí posloupnost, stačí poličky (i s jejich obsahem) ve skříni přeuspořádat. Nyní dokážeme, že existuje rozmístění knih ve skříni takové, že výšky knih tvoří nerostoucí posloupnost. Bez újmy na obecnosti můžeme předpokládat, že $v_1 \geqq \ldots \geqq v_N$. Uvažme rozmístění knih do skříně takové, že první polička obsahuje s_0 nejvyšších knih, druhá s_0 nejvyšších knih mezi zbylými knihami atd. a v každé z poliček výšky knih tvoří nerostoucí posloupnost. Tvrdíme, že výška nejvyšší knihy v i-té poličce je nejvýše w_i,

tj. $v_{(i-1)s_0+1} \leqq w_i$. Pokud tomu tak není, pak $((i-1)s_0+1)$-tá kniha musí být v optimálním řešení na jedné z prvních $i-1$ poliček, ale pak některá z $(i-1)s_0$ nejvyšších knih (řekněme ta s výškou v_k, $1 \leqq k \leqq (i-1)s_0$) není v optimálním řešení na jedné z prvních $i-1$ poliček — je tedy na j-té poličce, $j \geqq i$. Potom ale $w_j \leqq v_k$, a tedy $w_i \leqq v_{(i-1)s_0+1}$, což je požadovaná nerovnost.

Všimněme si, že jsme v předchozím odstavci vlastně dokázali, že ve výše popsaném optimálním řešení jsou všechny poličky až na tu poslední plné, tj. obsahují přesně s_0 knih. Základem našeho programu bude funkce `existuje(s:integer)`, která pro danou šířku s rozhodne, zda existuje knihovna maximální výšky 250 cm a šířky s, do které lze umístit všechny knihy. Optimální hodnotu s_0 nalezneme pak metodou půlení intervalu, kterou lze nalézt v popisu řešení úlohy P–I–2 domácího kola. Samotná funkce zvolí za výšku i-té poličky výšku $v_{(i-1)s_0+1}$, což je výška nejvyšší knihy, kterou uložíme do i-té poličky v řešení popsaném v minulém odstavci. Naše funkce z výšek jednotlivých poliček snadno spočte výšku celé knihovny a ověří, zda je nejvýše 250 cm.

Nyní odhadněme časové a paměťové nároky výše popsaného programu. Nejprve potřebujeme setřídit N čísel, což lze učinit užitím některého ze standardních algoritmů v čase $O(N \log N)$. Časová složitost funkce `existuje` je $O(N/s)$, neboť je v ní potřeba sečíst $\lceil N/s \rceil$ čísel. Odtud již plyne, že časové nároky celého našeho algoritmu jsou majorizovány funkcí $O(N \log N)$. Pokud si uvědomíme, že $s = N/2$ při prvním volání funkce `existuje`, $s = N/4$ při druhém, atd., pak lze časové nároky algoritmu bez úvodního setřídění výšek knih dokonce odhadnout funkcí $O(N)$. Paměťové nároky algoritmu lze odhadnout funkcí $O(N)$, neboť potřebujeme pole velikosti N na uložení výšek jednotlivých knih.

```
program knihovna;
const MAXN=100;
      VYSKA_MISTNOSTI=250;
var vyska: array[1..MAXN] of word;  { výšky knih }
    n: word;                        { počet knih }
procedure utrid_vysky(i1,i2:word);  { quicksort }
  var pivot: word;
      w: word;
      j1, j2: word;
  begin
    if i1>=i2 then exit;
    pivot:=vyska[(i1+i2) div 2];
    j1:=i1; j2:=i2;
    while (j1<j2) do
      begin
        while (vyska[j1]>pivot) do inc(j1);
```

```
        while (vyska[j2]<pivot) do dec(j2);
        w:=vyska[j1]; vyska[j1]:=vyska[j2]; vyska[j2]:=w;
        inc(j1); dec(j2);
      end;
    utrid_vysky(i1,j2);
    utrid_vysky(j1,i2);
  end;
function existuje(s:word):boolean;
  var v:word;
      i:word;
  begin
    v:=1; i:=1;
    repeat
      v:=v+vyska[i]+1;
      i:=i+s;
    until i>n;
    existuje:=v<=VYSKA_MISTNOSTI
  end;
var i:word;
    s1,s2:word;
    v:word;
begin
  readln(n);
  for i:=1 to n do read(vyska[i]);
  utrid_vysky(1,n);
  if vyska[1]>VYSKA_MISTNOSTI-2 then
    begin
      writeln('Pro zadané rozměry knih neexistuje knihovna!');
      halt;
    end;
  s1:=1; s2:=n;
  while s1<s2 do
    if existuje((s1+s2) div 2) then
      s2:=(s1+s2) div 2
    else
      s1:=(s1+s2) div 2+1;
  writeln('Optimální šířka skříně je ',s1,' cm.');
  writeln('Počet poliček ve skříni: ',(n+s1-1) div s1);
  i:=1; v:=1;
  while (i<=n) do
    begin
      v:=v+vyska[i]+1;
      writeln('Výška poličky: ',vyska[i],' cm');
      write('Výšky knih na poličce:');
      repeat
        if (i>n) then break;
        write(' ',vyska[i],' cm');
        inc(i);
      until (i mod s1)=1;
      writeln;
    end;
  writeln('Výška skříně: ',v,' cm');
end.
```

P – II – 3

Použijeme myšlenku podobnou té z řešení úlohy P–I–3 domácího kola; problém ovšem je, že když se nám nyní mohou písmena opakovat, následníci nemusí být jednoznačně určeni. Provedeme následující úvahu:

Máme dán poslední sloupec, jeho setříděním dostaneme první sloupec. Dále máme dánu pozici slova, které bylo zakódováno, v setříděné tabulce, tedy známe jeho první písmeno; nechť je to x. Toto písmeno se nám může v prvním sloupci vyskytovat vícekrát, na pozicích odpovídajících slovům xv_1, xv_2, ..., xv_k, kde $xv_1 \leqq xv_2 \leqq \ldots \leqq xv_k$. Z toho ovšem plyne také $v_1x \leqq v_2x \leqq \ldots \leqq v_kx$, a tedy je-li xv_j zakódované slovo, wx j-té (v abecedním pořadí) slovo končící na x, musí platit $w = v_j$. Nyní můžeme celý postup opakovat (pozice, na níž je první písmeno zbytku zakódovaného slova, je ta, na níž je v posledním sloupci j-té písmeno x).

Algoritmus je již pouze přímočarým přepisem této myšlenky. Implementace tohoto algoritmu je poměrně jednoduchá; místo komplikované práce s dvojicemi (písmeno, pozice) je výhodnější si písmena v posledním sloupci očíslovat (písmenu přiřadíme jeho index v posledním sloupci) a po setřídění (přihrádkovým tříděním, abychom dosáhli lineární časové složitosti) pracovat pouze s těmito indexy.

Časová i paměťová složitost algoritmu jsou opět lineární.

```
program transformace;
const MAX = 10000;
var prvni_sloupec : array[1 .. MAX] of integer;
    posledni_sloupec : string;
    radek, delka, i, l : integer;
    buckets: array[char] of integer;
    ch : char;
begin
  {nacteni a ocislovani}
  readln (posledni_sloupec);
  readln (radek);
  delka := length (posledni_sloupec);
  for ch := #0 to #255 do buckets[ch] := 0;
  for i := 1 to delka do
    inc (buckets[posledni_sloupec[i]]);

  {setrideni}
  l := 1;
  for ch := #0 to #255 do
    begin
      i := l;
      inc (l, buckets[ch]);
      buckets[ch] := i;
    end;
  for i := 1 to delka do
    begin
```

```
      ch := posledni_sloupec[i];
      l := buckets[ch];
      inc (buckets[ch]);
      prvni_sloupec[l] := i;
    end;

  {vypis}
  for i:=1 to delka do
    begin
      write (posledni_sloupec[prvni_sloupec[radek]]);
      radek := prvni_sloupec[radek];
    end;
  writeln;
end.
```

P – II – 4

Podobnost úlohy s počítáním vzdálenosti vrcholů (tj. délky nejkratší cesty mezi nimi) v orientovaném grafu jistě není náhodná, držme se proto i my grafové analogie: Jednotlivé budovy Ouřadu jsou pro nás vrcholy, potrubí mezi nimi orientovanými hranami grafu a A není ničím jiným než maticí sousednosti grafu. Nabízí se použít prohledávání grafu do šířky, ovšem musíme je náležitě upravit, aby bylo reverzibilní.

Vrcholy grafu si rozdělíme do *vrstev*: i-tá vrstva W_i bude obsahovat právě ty vrcholy, jejichž vzdálenost od vrcholu x je rovna i. Vrstev je proto nejvýše n a můžeme je snadno zkonstruovat indukcí: do W_0 padne vrchol x a žádný další; když máme sestrojeny vrstvy W_0 až W_{i-1}, tak do W_i patří právě ty vrcholy w, do kterých vede hrana z nějakého vrcholu $v \in W_{i-1}$ (tedy existuje cesta délky i z x do w) a $w \notin W_j$ pro $j < i$ (neexistuje žádná kratší cesta).

To je bezpochyby reverzibilní postup — při konstrukci vrstvy nijak neměníme vrstvy už spočítané; nakonec najdeme číslo vrstvy, do které padl vrchol y, to vydáme jako výsledek a všechny informace o vrstvách opět odpočítáme. Tak dostaneme řešení s časovou složitostí $O(n^3)$ a prostorovou složitostí $O(n^2)$. Všimněme si ještě dvou drobností:

1. Ačkoliv vrstev může být až n a v každé z nich až $n-1$ vrcholů, lze je uložit efektivněji, protože ve všech vrstvách dohromady je nejvýše n vrcholů. Stačí je všechny naskládat za sebe do jednoho pole (říkejme mu třeba V) a nechat druhé pole S ukazovat, kde v poli V která vrstva začíná. Vrcholy ve vrstvě W_i tedy budou uloženy v prvcích V_{S_i} až $V_{S_{i+1}-1}$.
2. Reverzibilita programu není příliš nakloněna značkování vrcholů. Když si totiž budeme v nějakém poli pro každý vrchol pamatovat,

zda jsme v něm již byli, a případně jej pak označkujeme, řekněme takto:

```
if UžJsemTamByl[i]=0 then begin
   { objevil jsem nový vrchol a někam si ho zapíšu }
   UžJsemTamByl[i] += 1;
   end;
```

dostaneme se do sporu s reverzibilitou podmínek: po ukončení příkazu `if` nepoznáme, zda byla podmínka splněna či nikoliv, protože `UžJsemTamByl[i]` bude vždycky jednička. To přesně náš jazyk zakazuje. Naštěstí nás zachrání jednoduchý trik: pokud dokážeme zajistit, abychom v rámci jedné vrstvy na každý vrchol narazili nejvýše jednou, stačí si u každého vrcholu zapamatovat (k tomu budeme používat pole L), ve které vrstvě byl objeven, a pokud dosud objeven nebyl, tak nějaké dostatečně velké číslo `inf`. Test se změní na

```
if L[i] >= TatoVrstva then begin
   { objevil jsem nový vrchol a někam si ho zapíšu }
   L[i] -= inf - TatoVrstva;
   end;
```

a to už je korektní: platnost podmínky v této vrstvě se totiž přenastavením `L[i]` nezmění, ale v dalších vrstvách již správně poznáme, že vrchol byl zpracován.

Zde je program využívající oba popsané triky:

```
procedure Zkoumej(var n:word; var A:array [1..n] of array [1..n] of bit;
          var x,y,d:word);
var inf,cnt:word;
var L,V,S:array [0..n] of word;
begin
   wrap begin
      inf += n+1;                          { "nekonečná vzdálenost" }
      for var i = 1 to n do                { L[i] = inf }
         L[i] += inf;
      V[0] += x;                           { nultá vrstva: vrchol x... }
      L[x] -= inf;                         { ...ve vzdálenosti 0... }
      S[1] += 1;                           { ...a žádný další }
      for var i = 1 to n-1 do begin        { hledáme další vrstvy }
         S[i+1] += S[i];                   { zatím prázdná }
         for var w = 1 to n do
            if L[w] >= i then              { nezařazený vrchol }
               wrap
                  for var j = S[i-1] to S[i]-1 do
                                           { vede do něj hrana z vrstvy i-1? }
                     if A[V[j]][w]=1 then
                        cnt += 1
               on if cnt>0 then begin { ano => přidat do i-té vrstvy }
```

```
                    V[S[i+1]] += w;
                    S[i+1] += 1;
                    L[w] -= inf-i     { L[w] >= i stále platí }
                    end
        end
      end
   on d += L[y]                       { vrátíme výsledek }
end;
```

Zbývá ještě dodat, že prostorová složitost procedury je lineární a časová kvadratická (inicializace je lineární, vše mimo cyklu řízeného proměnnou j kvadratické a vnitřek zbylého cyklu se provede pro každý vrchol j právě n-krát, takže je dohromady také kvadratický).

Poznámka. Pokud bychom se vzdali polynomiální časové složitosti, existovala by prostorově ještě efektivnější řešení. Jedno z nich je založeno na následující úvaze: hledám-li cestu délky l z x do y, pak je buďto $l < 2$ (tehdy je úloha triviální), nebo cesta musí mít nějaký střední vrchol ve vzdálenosti $\lfloor \frac{1}{2} l \rfloor$. Vyzkouším proto postupně všechny vrcholy a pro každý z nich si rekurzivním zavoláním téže funkce pro obě poloviny cesty a poloviční l ověřím, zda existuje příslušná polovina cesty. Hloubka rekurze je maximálně $\lceil \log_2 l \rceil = O(\log n)$, dosáhneme tedy prostorové složitosti $O(\log n)$ za cenu drastického zpomalení na $n^{O(\log n)}$.

P – III – 1

Při řešení úlohy si nejdříve uvědomíme, že ze zadaných čísel hraček můžeme snadno odvodit, které dítě chce hračku po kterém dítěti. Situaci si představíme jako orientovaný graf, kde vrcholy odpovídají dětem a od vrcholu i vede hrana k vrcholu j, pokud dítě i chce hračku po dítěti j. Protože dítě je ochotno vyměnit hračku pouze tehdy, když dostane tu svou vytouženou, mohou si děti vyměňovat hračky pouze po cyklech — aby se dítě i_1 vzdalo své hračky, musí dostat hračku od i_2, to od i_3 a tak dále, až nějaké dítě dostane hračku od i_1. Chceme tedy nalézt v grafu množinu disjunktních kružnic (pro snazší vyjadřování budeme nadále považovat za kružnici i vrchol se smyčkou), které dohromady obsahují co nejvíce vrcholů. Hledání těchto kružnic je usnadněno tím, že každé dvě kružnice v našem grafu jsou disjunktní — kdyby nějaké dvě kružnice měly společný vrchol, musely by se v nějakém místě také od sebe oddělovat. Z příslušného vrcholu by tedy musely vést dvě hrany, což ovšem v našem grafu není možné.

A nyní jak budeme kružnice hledat: Začneme v libovolném vrcholu (třeba prvním) a půjdeme po hranách (z každého vrcholu vede právě

jedna hrana, takže postup je jednoznačný), dokud se nevrátíme do nějakého vrcholu, ve kterém jsme už byli (to poznáme snadno, když si budeme označovat navštívené vrcholy). Tím jsme v grafu nalezli nějakou kružnici, tu můžeme vypsat a její vrcholy označit za vyřešené. Vrcholy, které jsme prošli předtím, než jsme se dostali na kružnici, pro změnu zaručeně na žádné kružnici neleží (jinak by z nějakého vrcholu musely vést alespoň dvě hrany). Proto se těmito vrcholy už nikdy nemusíme zabývat a můžeme je rovněž označit jako vyřešené. Nyní vezmeme další dosud nevyřešený vrchol a opět se z něj vydáme hledat kružnici. Pokud narazíme na nějaký již vyřešený vrchol, hledání ukončíme a prošlé vrcholy označíme jako vyřešené — nemohou totiž zřejmě ležet na žádné kružnici. Když už nezbude žádný nevyřešený vrchol, máme nalezeny všechny kružnice a výpočet ukončíme. Jediným nedořešeným problémem zůstává, jak rychle hledat dosud nevyřešené vrcholy. To můžeme snadno dělat tak, že při hledání dalšího nevyřešeného vrcholu začneme hledat od naposledy nalezeného vrcholu (před ním jistě žádné nevyřešené již nejsou). Díky tomu s hledáním vrcholů strávíme dohromady čas $O(N)$, a protože na nalezení kružnic potřebujeme dohromady též $O(N)$ (každou hranou projdeme nejvýše jednou), je celková časová složitost $O(N)$. Paměťová složitost je také $O(N)$.

```
/* Hračkářství */
#include <stdio.h>

#define MAXD 100    /* Maximální počet dětí */

int N;              /* Počet dětí */
int Ma[MAXD];       /* Číslo hračky, kterou příslušné dítě má */
int Vlastni[MAXD];  /* Dítě, které vlastní příslušnou hračku */
int Chce[MAXD];     /* Dítě, jehož hračku příslušné dítě chce */
int Hotovo[MAXD];   /* Už jsme dítě řešili? */

/* Načte vstup */
void nacti(void)
{
  int i;

  scanf("%d", &N);
  for (i = 0; i < N; i++) {
    printf("Dite %d: ", i+1);
    scanf("%d %d", &Ma[i], &Chce[i]);
    Ma[i]--; Chce[i]--;
    Vlastni[Ma[i]] = i;
  }
  /* Převedeme odkazy na hračky na odkazy na děti */
  for (i = 0; i < N; i++)
    Chce[i] = Vlastni[Chce[i]];
}
```

```
/* Projde děti a zjistí největší spokojenou skupinu */
void res(int act)
{
  int start = act;

  /* Projde děti a najde cyklus */
  while (!Hotovo[act]) {
    Hotovo[act] = 1;
    act = Chce[act];
  }
  /* Vypíše cyklus */
  while (Hotovo[act] != 2) {
    Hotovo[act] = 2;
    printf(" %d", act+1);
    act = Chce[act];
  }
  /* Ještě označíme zbylé prošlé vrcholy */
  act = start;
  while (Hotovo[act] != 2) {
    Hotovo[act] = 2;
    act = Chce[act];
  }
}

int main(void)
{
  int i;

  nacti();

  printf("Spokojene deti:");
  for (i = 0; i < N; i++)
    if (!Hotovo[i])      /* Zatím jsme dítě neřešili? */
      res(i);
  printf("\n");
  return 0;
}
```

P – III – 2

Základem našeho řešení bude funkce `existuje(s:integer)`, která pro zadanou šířku s rozhodne, zda existuje knihovna s P poličkami, do které lze umístit všech N knih. Označme T součet tlouštěk knih, tj. $T = t_1 + \ldots + t_N$. Potom minimální šířka knihovny s P poličkami pro dané knihy je alespoň T/P. Na druhou stranu, určitě existuje knihovna šířky $T/P + t_{\max}$, kde $t_{\max}$ je maximální tloušťka knihy: Knihy rozmístíme na poličky tak, že každých prvních k poliček obsahuje nejmenší možný počet knih takový, aby součet tlouštěk knih na těchto poličkách byl alespoň kT/P. Snadno nahlédneme, že šířka každé poličky je nejvýše $T/P + t_{\max}$, a tedy existuje knihovna takové šířky. Optimální šířku skříně

pak nalezneme vyzkoušením všech hodnot mezi T/P a $T/P + t_{\max}$ jako možné šířky skříně. Takových hodnot je ale konstantně mnoho kvůli omezení na tloušťku knihy ze zadání úlohy.

Samotná funkce `existuje` bude fungovat následovně: Pro zadané s nalezne největší i_1 takové, že $\sum_{i=1}^{i_1} t_i \leqq s$; je jasné, že i_1 je maximální možný počet knih, které lze umístit do první poličky. Poté nalezneme největší i_2 takové, že $\sum_{i=i_1+1}^{i_2} t_i \leqq s$, tedy největší možný počet knih i_2, které lze umístit do prvních dvou poliček, atd. Pokud se nám podaří umístit všechny knihy, tj. $i_P = N$, pak existuje knihovna šířky s, do které lze všechny knihy uložit; v opačném případě taková knihovna zjevně neexistuje.

Zbývá domyslet, jak rychle hledat čísla i_k, $1 \leqq k \leqq P$, ve funkci `existuje`. Za tímto účelem si nejprve vytvoříme pomocné pole, ve kterém budou uloženy součty tlouštěk prvních j knih pro $1 \leqq j \leqq N$. Při počítání hodnoty i_k metodou půlení intervalu vyhledáme v tomto pomocném poli největší číslo i' takové, že $\sum_{i=1}^{i'} t_i - \sum_{i=1}^{i_{k-1}} t_i \leqq s$; zřejmě i' je hledaná hodnota i_k.

Nyní odhadněme časovou a paměťovou složitost našeho algoritmu. Funkce `existuje` provede P vyhledávání v poli velikosti N, tj. doba jejího běhu je majorizována funkcí $O(P \log N)$. Celková doba běhu našeho programu je tedy $O(N + P \log N)$; čas $O(N)$ spotřebujeme kromě načtení dat také na vytvoření pomocného pole popsaného v minulém odstavci. Pokud by platilo, že $P \log N > N$, lze výše popsanou funkci `existuje` nahradit jednodušší funkcí pracující v čase $O(N)$, která místo P binárních vyhledávání projde pole sekvenčně. Časová složitost našeho programu je tedy majorizována funkcí $O(N)$. Paměťová složitost je $O(N)$ — pole velikosti N je potřeba na uložení tlouštěk jednotlivých knih a stejná je i velikost pomocného pole.

```
program knihovna;
const MAXN=1000;
var tloustka: array[1..MAXN] of word;   { tloušťky knih }
    soucet: array[0..MAXN] of word;     { součty tlouštěk knih }
    n: word;                            { počet knih }
    p: word;                            { počet poliček }
function vyhledej(s: word): word;
  var i1,i2: word;
  begin
    i1:=0; i2:=n;
    while i1<i2 do
```

```
      if soucet[(i1+i2+1) div 2]>s then
        i2:=(i1+i2+1) div 2-1
      else
        i1:=(i1+i2+1) div 2;
    vyhledej:=i1
  end;
function existuje(sirka: word):boolean;
  var i,j: word;
  begin
    i:=0;
    for j:=1 to p do i:=vyhledej(soucet[i]+sirka);
    existuje:=i=n;
  end;
var i: word;
    s1, s2: word;
    tmax: word;
begin
  readln(n,p);
  tmax:=0;
  for i:=1 to n do
    begin
      read(tloustka[i]);
      if tmax<tloustka[i] then tmax:=tloustka[i]
    end;
  soucet[0]:=0;
  for i:=1 to n do soucet[i]:=soucet[i-1]+tloustka[i];
  s1:=soucet[n] div p;
  s2:=soucet[n] div p+tmax;
  while s1<s2 do
    if existuje((s1+s2) div 2) then
      s2:=(s1+s2) div 2
    else
      s1:=(s1+s2) div 2+1;
  writeln('Optimální šířka skříně: ',s1,' mm');
  i:=1;
  while i<=n do
    begin
      write('Knihy na poličce:');
      s2:=0;
      while (i<=n) and (s2+tloustka[i]<=s1) do
        begin
          write(' ',i,'(',tloustka[i],' mm)');
          s2:=s2+tloustka[i];
          inc(i);
        end;
      writeln;
    end
end.
```

P – III – 3

První řešení. Inspirujeme se tradičním algoritmem pro sčítání čísel „pod sebou" a uvědomíme si, že není závislý na použité číselné soustavě (zvěda-

vější povahy obětují 5 minut na důkaz indukcí). Pokud bychom uměli spočítat přenosy mezi řády, je samotné sečtení triviální: $A'_i = A_i \text{xor} B_i \text{xor} P_i$, kde P_i je přenos z $(i-1)$-ního do i-tého řádu (*xor* funguje úplně stejně jako sčítání dvou bitů modulo 2). Pokud jsme ochotni obětovat paměť na všechna P_i, můžeme je spočítat postupně: $P_0 = 0$; pro $i > 0$ je $P_i = 1$, když buďto A_{i-1} a B_{i-1} jsou současně jedničky, nebo když alespoň jedno z nich je jednička a P_{i-1} je jednička. Z toho okamžitě dostáváme program s lineární časovou i prostorovou složitostí:

```
procedure Add(var n:word; var A,B:array [0..n-1] of bit);
var P:array [0..n] of bit;
begin
   wrap
      for var i = 0 to n-1 do
         P[i+1] ^= (A[i] and B[i]) or ((A[i] or B[i]) and P[i])
   on
      for var i = 0 to n-1 do
         A[i] ^= B[i] xor P[i]
end;
```

My se ovšem s lineárním množstvím paměti nespokojíme a zkusíme být při výpočtu přenosů šetrnější. Celý problém je v tom, že na spočtení P_i potřebujeme P_{i-1}, a to musí být dostupné i v okamžiku, kdy budeme P_i odpočítávat (pokusy o odpočítávání P_i pomocí P_{i+1} selhávají na tom, že když už jsme si jednoho ze sčítanců přepsali výsledkem, nelze určit, zda jednička z výsledku vznikla z jedničky v přepsaném sčítanci nebo z nuly a přenosu z nižšího řádu). Takže si musíme P_{i-1} celou dobu pamatovat a prostorová složitost prostě musí být vždy alespoň lineární a naše první řešení je optimální... a nebo přeci jen ne? Nešlo by na P_{i-1} zapomenout, a až budeme chtít P_i odpočítat, tak si P_{i-1} spočítat znovu? To by fungovalo, ale musíme to provést šikovně, abychom rekurzivním voláním výpočtů předchozích P_j nespotřebovali více paměti, než jsme ušetřili. Tak získáme

Druhé řešení. Sestrojíme si proceduru `Prenos(i,l,in,out)`, která pro nějaký úsek čísel A a B (konkrétně od i-tého řádu do $(i+l-1)$-ního) za předpokladu, že přenos do našeho úseku z nižších řádů $P_i = in$, spočítá přenos P_{i+l} do vyšších řádů a přixoruje jej k proměnné *out*. Pokud je úsek jednoprvkový, udělá to již dobře známým způsobem z našeho prvního řešení (v konstantní paměti). Větší úsek si rozdělí na poloviny, rekurzivně si spočítá přenos *mid* z nižší poloviny do vyšší, pak rekurzivně spočítá přenos z vyšší poloviny „ven" a nakonec *mid* třetím rekurzivním zavoláním odpočítá. To se dá jako obvykle snadno zapsat pomocí příkazu `wrap`:

```
procedure Prenos(var i,l:word; var in,out:bit);
var l1,l2,j:word;
var mid:bit;
begin
   if l=1 then                { jednobitový přenos }
      out ^= (A[i] and B[i]) or ((A[i] or B[i]) and in)
   else wrap begin
      l1 += l div 2;          { l1=délka dolní poloviny }
      l2 += l-l1;             { l2=délka horní poloviny }
      j += i+l1;              { j=začátek horní poloviny }
      Prenos(i,l1,in,mid)     { přenos přes dolní polovinu }
      end
   on Prenos(j,l2,mid,out)    { přenos přes horní polovinu }
end;
```

Jelikož při každém rekurzivním volání klesne l minimálně na polovinu, je hloubka rekurze nejvýše $\lceil \log_2 l \rceil$, takže procedura `Prenos` dosahuje prostorové složitosti $O(\log l)$. Se složitostí časovou je to trochu obtížnější: Označíme-li čas strávený touto procedurou $T(l)$, bude platit $T(l) = 1 + 3T(l/2)$: procedura vykoná nějakou konstantní práci (jelikož nás složitost zajímá jen asymptoticky, můžeme předpokládat, že jednotkovou), načež třikrát zavolá sama sebe na vstup poloviční délky. Dosadíme-li tento vztah do sebe sama, dostaneme $T(l) = 1 + 3(1 + 3T(l/4)) = 1 + 3 + 9T(l/4)$ a když budeme dosazovat dál, po k krocích dojdeme k $T(l) = 1 + 3 + \ldots + 3^{k-1} + 3^k T(l/2^k)$. My ale víme, že $T(1) = 1$, takže pro $k = \log_2 l$ (naše hloubka rekurze) vyjde $T(l) = 1 + 3 + \ldots + 3^{\log_2 l - 1} + 3^{\log_2 l}$. To je ovšem geometrická řada se součtem $\frac{1}{2}(3^{k+1} - 1) = O(3^k) = O(3^{\log_2 l})$, což můžeme ještě zjednodušit: $3^{\log_2 l} = (2^{\log_2 3})^{\log_2 l} = 2^{\log_2 3 \log_2 l} = (2^{\log_2 l})^{\log_2 3} = l^{\log_2 3} \leqq l^{1,59}$. Z toho plyne, že časová složitost celé procedury je $T(l) = O(l^{1,59})$.

Teď bychom mohli rekurzivní výpočet přenosů zapojit do naší původní sčítací procedury (musíme ovšem sčítat pozadu, abychom si nepřepsali hodnoty, ze kterých budeme přenosy ještě potřebovat) a získat tak sčítání s logaritmickou prostorovou složitostí v čase $O(n \cdot n^{1,59}) = O(n^{2,59})$, ale neuděláme to, protože si všimneme, že každý z blokových přenosů bychom zbytečně počítali mnohokrát.

Místo toho zkonstruujeme podobnou rekurzivní proceduru, která bude provádět současně sčítání a počítání přenosu. Nazveme ji `Secti` a bude mít úplně stejné parametry jako procedura `Prenos`. Nejdříve si zavolá proceduru `Prenos` pro výpočet přenosu z dolní poloviny bloku (ten opět přixoruje k proměnné *mid*), pak rekurzivním zavoláním sebe samé sečte horní polovinu čísla a nakonec rekurzivně zavolá sebe samu

pro dolní polovinu čísla, čímž ji jednak sečte a jednak odpočítá přenos *mid*. Triviální případ sčítání jednobitových čísel opět vyřešíme klasicky.

```
procedure Secti(var i,l:word; var in,out:bit);
var l1,l2,j:word;
var mid:bit;
begin
   if l=1 then begin          { jednobitové sčítání }
      out ^= (A[i] and B[i]) or ((A[i] or B[i]) and in);
      A[i] ^= B[i] xor in
      end
   else wrap begin
      l1 += l div 2;          { opět počítáme, kde jsou poloviny }
      l2 += l-l1;
      j += i+l1
      end
   on begin
      Prenos(i,l1,in,mid);    { přenos přes dolní polovinu }
      Secti(j,l2,mid,out);    { sečteme horní polovinu }
      Secti(i,l1,in,mid)      { sečteme dolní a odpočteme přenos }
      end
end;
```

Časová i prostorová složitost naší sčítací procedury bude stejná jako u procedury `Prenos`, protože až na ošetřování triviálních případů, které je konstantní, vypadají obě procedury úplně stejně. Sčítáme tedy v prostoru $O(\log n)$ a čase $O(n^{1{,}59})$. Program vypadá takto:

```
procedure Add(var n:word; var A,B:array [0..n-1] of bit);
{ Zde jsou vloženy procedury Prenos a Secti }
var zero:word;
var in,out:bit;
begin
   Secti(zero,n,in,out);     { víme, že out vyjde nulový }
end;
```

Třetí řešení. A nešlo by to ještě lépe? Zkusme vyřešit jednodušší problém: jak k danému číslu přičíst jedničku, tedy nalézt maximální souvislý úsek jedniček na nejnižších řádech, tyto jedničky změnit na nuly a bezprostředně předcházející nulu změnit na jedničku. Jinak řečeno změnit ty číslice, za kterými již nenásleduje žádná nula. To se ovšem dá snadno zařídit následujícím trikem: Nejdříve postupujeme od nejnižšího řádu k nejvyššímu a za každou nulu si do počítadla přičteme jedničku, a pak projdeme pole ještě jednou v opačném směru, počítadlo za každou nulu o jedničku snižujeme, a jakmile dospěje do nuly, začneme všechny bity, přes které přejdeme, negovat:

```
procedure AddOne(var n:word; var A:array [0..n-1] of bit);
var i,c:word;
begin
   for var i=0 to n-1 do
      if A[i]=0 then c += 1;
```

```
   for var i=n-1 downto 0 do begin
      if A[i]=0 then c -= 1;
      if c=0 then A[i] ^= 1;
      end;
end;
```

Dokážeme to tedy v lineárním čase a konstantním prostoru. Jenže když umíme přičíst jedničku, dokážeme přičíst i libovolnou mocninu dvojky — stačí začít u jiného než nejnižšího řádu, a tím pádem také libovolné jiné číslo, protože ho můžeme rozložit na mocniny dvojky a každou přičíst zvlášť:

```
procedure Add(var n:word; var A,B:array [0..n-1] of bit);
var i,j,c:word;
begin
   for var i=0 to n-1 do
      if B[i]=1 then begin
         for var j=i to n-1 do
            if A[j]=0 then c += 1;
         for var j=n-1 downto i do begin
            if A[j]=0 then c -= 1;
            if c=0 then A[j] ^= 1;
            end;
         end;
end;
```

Tak dosáhneme časové složitosti $O(N^2)$ při prostorové složitosti $O(1)$.

Poznámka na závěr. Řešení v konstantním prostoru těží z toho, že jsme v našem výpočetním modelu nadefinovali prostorovou složitost poněkud nedbale a neměříme ji v bitech, nýbrž ve wordech. Kdybychom počítali opravdu precizně, nebyla by prostorová složitost třetího řešení konstantní, nýbrž logaritmická, zatímco druhé řešení by se dalo snadno upravit tak, aby mělo stále logaritmickou složitost (stačí si uvědomit, že je lze naprogramovat nerekurzivně, čímž se zbavíme závislosti prostoru na lokální proměnné). Ovšem časové složitosti zůstanou zachovány, takže druhé řešení bude pracovat v témže prostoru rychleji.

P – III – 4

Úloha, převedená do podoby v matematice běžnější, zní: Je dán konvexní N-úhelník a M jeho neprotínajících se tětiv dělících N-úhelník na díly. Nalezněte maximální počet dílů, z nichž žádné dva nemají společnou stranu.

Uvažujme následující graf G. Vrcholy grafu budou odpovídat jednotlivým dílům N-úhelníku, přičemž dva vrcholy budou spojeny hranou, pokud jim odpovídající díly mají společnou stranu. Graf G zřejmě bude

souvislý a navíc nebude obsahovat žádný cyklus. Uvnitř cyklu by totiž ležela alespoň jedna stěna grafu G. Té musí odpovídat nějaký průsečík v nakresleném N-úhelníku. Tento průsečík však rozhodně nemůže ležet na okraji N-úhelníku, a máme tak spor s tím, že žádné dvě tětivy se neprotínají.

Souvislý graf bez cyklů je strom a naše úloha se tím zjednodušuje na nalezení maximální nezávislé množiny vrcholů (tj. takové množiny vrcholů, že žádné dva vrcholy z této množiny nejsou spojeny hranou) ve stromu. Maximální nezávislou množinu můžeme určit prohledáváním do hloubky. Na počátku si označíme všechny vrcholy jako přijatelné do nezávislé množiny. Začneme v libovolném vrcholu prohledávat strom. Když se vracíme z nějakého vrcholu, který je označen jako přijatelný, přidáme ho do nezávislé množiny a jeho otce odznačíme. Když takto projdeme celý graf, máme vybranou maximální nezávislou množinu. Nezávislost vybrané množiny je zřejmá. Proč ale bude vybraná množina maximální? Označme si vybranou nezávislou množinu A a dále si vezměme maximální nezávislou množinu B, která se od naší vybrané množiny liší v nejméně vrcholech. Nyní se podívejme na takový vrchol v, ve kterém se A a B liší a který je nejvzdálenější od vrcholu, ve kterém začalo prohledávání do hloubky. Případ, kdy v je v B a ne v A, nastat nemůže, protože když jsme nějaký vrchol v nevzali do A, tak pouze proto, že byl sousedem nějakého vrcholu u pod ním zařazeného do A. Protože v je nejvzdálenější vrchol, ve kterém se A a B liší, musí být u obsažen i v B, a tedy B také nemůže obsahovat v. Může tedy nastat pouze situace, že v je obsažen v A a není obsažen v B. Pokud ale v přidáme do B a z B vyřadíme otce v (pokud v ní byl), bude B stále maximální nezávislá množina a přitom se bude lišit v méně vrcholech, což je spor s výběrem B. Vybraná nezávislá množina A musí být proto skutečně maximální.

Zkonstruovat výše popsaný graf a na něm pak provést prohledání do hloubky je zbytečně pracné. My budeme graf prohledávat bez jeho explicitní konstrukce. Nejdříve si tětivy zorientujeme tak, aby každá tětiva začínala ve vrcholu s nižším číslem a přidáme pomocnou tětivu začínající v prvním a končící v posledním vrcholu. Tětivy si pomocí přihrádkového třídění setřídíme vzestupně podle jejich počátku, tětivy začínající ve stejném vrcholu pak sestupně podle jejich konce. Nyní postupně procházíme vrcholy N-úhelníku v pořadí od vrcholu s číslem jedna po vrchol s číslem N. Při procházení si udržujeme zásobník s tětivami, od nichž jsme viděli začátek, ale ne konec. U každé tětivy na zásobníku si navíc pamatujeme, zda je přijatelná. Vždy, když začneme zpracovávat nový vrchol,

nejdříve ze zásobníku odebereme tětivy, které v tomto vrcholu končí. Pokud je odebíraná tětiva označena jako přijatelná, zvětšíme velikost nezávislé množiny a odznačíme tětivu pod ní v zásobníku. Po odebrání všech končících tětiv přidáme na zásobník všechny tětivy začínající v daném vrcholu a označíme je jako přijatelné. Pak pokračujeme do dalšího vrcholu N-úhelníku. Výpočet skončíme po průchodu všemi vrcholy N-úhelníku.

Uvedený algoritmus přesně odpovídá dříve popsanému prohledávání do hloubky. Každá tětiva totiž jednoznačně koresponduje s hranou v grafu, která spojuje vrcholy odpovídající dílům odděleným tětivou. Uložení pomocné tětivy $(1, N)$ na zásobník odpovídá vstupu do vrcholu, ze kterého začínáme prohledávání. Uložení další tětivy na zásobník odpovídá přechodu po odpovídající hraně dolů (směrem od vrcholu, ve kterém začalo prohledávání), vybrání tětivy ze zásobníku, pak návratu zpět po hraně. Při prohledávání do hloubky jsme si označovali vrcholy, které lze přidat do nezávislé množiny. V upraveném algoritmu místo vrcholu značíme tu hranu, po které jsme do vrcholu poprvé vstoupili. Algoritmus má časovou i paměťovou složitost $O(N)$ (tětiv nikdy nemůže být více než $N - 3$).

```
#include <stdio.h>
#include <stdlib.h>

#define MAXV 30000  /* Maximální počet vrcholů */
#define MAXR 30000  /* Maximální počet řezů */

/* Struktura pro jeden řez */
struct rez {
  int a, b;
};

int rezu, vrcholu;  /* Počet řezů a vrcholů */
struct rez r[MAXR]; /* Jednotlivé řezy */
int vpoc[MAXV];     /* Počty řezů začínajících v jednotlivých vrcholech */

/* Načte vstup */
void nacti(void)
{
  int pom, i;
  FILE *vstup;

  if (!(vstup = fopen("poklad.in", "r")))
    exit(1);
  fscanf(vstup, "%d %d", &vrcholu, &rezu);
  for (i = 0; i < rezu; i++) {
    fscanf(vstup, "%d %d", &r[i].a, &r[i].b);
    r[i].a--; r[i].b--;
    if (r[i].a > r[i].b) {
      pom = r[i].a;
```

```
      r[i].a = r[i].b;
      r[i].b = pom;
    }
  }
  fclose(vstup);
  /* Přidáme ještě fiktivní řez mezi prvním a posledním vrcholem */
  r[rezu].a = 0;
  r[rezu].b = vrcholu-1;
  rezu++;
}

/* Setřídí řezy podle počátku a konce */
void setrid(void)
{
  struct rez r1[MAXR];  /* Jednotlivé přeskládané řezy */
  int vrchind[MAXV];    /* Index, kde začínají řezy z/do daného vrcholu */
  int vrchpoc[MAXV];    /* Počty řezů z/do daného vrcholu */
  int i;

  /* První průchod třídění */
  for (i = 0; i < vrcholu; i++)
    vrchpoc[i] = 0;
  /* Spočteme počty řezů do jednotlivých vrcholů */
  for (i = 0; i < rezu; i++)
    vrchpoc[r[i].b]++;
  vrchind[0] = 0;
  for (i = 1; i < vrcholu; i++)
    vrchind[i] = vrchind[i-1] + vrchpoc[i-1];
  /* Přerovnáme řezy podle cílového vrcholu */
  for (i = 0; i < rezu; i++)
    r1[vrchind[r[i].b]++] = r[i];

  /* Druhý průchod třídění */
  for (i = 0; i < vrcholu; i++)
    vrchpoc[i] = 0;
  for (i = 0; i < rezu; i++)
    vrchpoc[r1[i].a]++;
  vrchind[0] = 0;
  for (i = 1; i < vrcholu; i++)
    vrchind[i] = vrchind[i-1] + vrchpoc[i-1];
  /* Přerovnáme řezy podle zdrojového vrcholu
     (bereme je sestupně podle cílového vrcholu) */
  for (i = rezu-1; i >= 0; i--)
    r[vrchind[r1[i].a]++] = r1[i];
}

/* Spočte, kolik částí mapy může kapitán rozdat */
int spocti(void)
{
  int casti = 0;      /* Počet částí */
  int zasvrch = 0;    /* Vrchol zásobníku */
  int zas[MAXR];      /* Zásobník na zpracovávané řezy */
  int zasuzit[MAXR]; /* Značka, že příslušná část mapy může být rozdána */
  int actvrch = 0, actrez = 0;  /* Aktuální vrchol a řez mnohoúhelníku */
```

```
  while (actvrch < vrcholu) {
    while (zasvrch && r[zas[zasvrch-1]].b == actvrch) {
      if (zasuzit[zasvrch-1]) {
         /* Část oddělená tímto řezem může být použita? */
        casti++;
        if (zasvrch > 1)
          zasuzit[zasvrch-2] = 0;
      }
      zasvrch--;
    }
    while (actrez < rezu && r[actrez].a == actvrch) {
      zas[zasvrch] = actrez++;
      zasuzit[zasvrch++] = 1;
    }
    actvrch++;
  }
  return casti;
}

int main(void)
{
  FILE *vystup;

  nacti();
  setrid();

  if (!(vystup = fopen("poklad.out", "w")))
    exit(1);
  fprintf(vystup, "%d\n", spocti());
  fclose(vystup);
  return 0;
}
```

P – III – 5

Použijeme upravený třídící algoritmus mergesort. V programu si budeme vytvářet jednosměrné spojové seznamy, jež budou mít svým jednotlivým prvkům přiřazeny mince. Váhy mincí budou od počátku ke konci seznamu tvořit rostoucí posloupnost. Mince stejné hmotnosti budou přiřazeny témuž prvku seznamu. Tento seznam budeme realizovat tak, že každý jeho prvek bude obsahovat ukazatel na následující prvek seznamu a na strom, který obsahuje mince (téže hmotnosti) přiřazené tomuto prvku. Samotný strom bude binární strom, ve kterém má každý prvek žádného nebo dva syny. Čísla mincí ve stromu budou uchovávána v jeho listech a každý uzel tohoto stromu bude obsahovat číslo některé mince ze svého podstromu (v naší implementaci to bude nejmenší číslo mince ve stromu).

Základem bude rekurzivní procedura `vytvor(prvni,posledni)`, která vytvoří jednosměrný spojový seznam popsaný v prvním odstav-

ci. Tento seznam bude obsahovat všechny mince s čísly od `prvni` do `posledni`. Pokud jsou čísla `prvni` a `posledni` shodná, procedura vytvoří jednoprvkový seznam. Jeho jediný prvek bude ukazovat na strom tvořený jedním uzlem, který bude obsahovat číslo `prvni` = `posledni`. Pokud jsou čísla `prvni` a `posledni` různá, procedura nejdříve rozdělí interval tvořený čísly od `prvni` do `posledni` na dva intervaly poloviční délek a na každý z nich se rekurzivně zavolá. Takto získáme dva lineární spojové seznamy s vlastnostmi popsanými v prvním odstavci. Z nich naše procedura vytvoří jeden.

Výsledný seznam budeme vytvářet od začátku, a to následujícím způsobem: Na některou z mincí ve stromu hlavy (tj. prvního prvku) prvního ze seznamů a na některou z mincí ve stromu hlavy druhého seznamu zavoláme funkci `porovnej`. Pokud je mince hlavy prvního seznamu lehčí, odpojíme hlavu od prvního seznamu a připojíme ji na konec výsledného seznamu; poté pokračujeme porovnáním hlav nově vzniklé dvojice seznamů. Pokud je naopak mince hlavy druhého seznamu lehčí, připojíme na konec výsledného seznamu hlavu druhého seznamu a pokračujeme s druhým seznamem bez jeho původní hlavy. Zbývá případ, kdy mince obou hlav mají stejnou hmotnost. V tomto případě připojíme na konec výsledného seznamu prvek, který ukazuje na strom, jehož levý podstrom je strom hlavy prvního seznamu a pravý podstrom je strom hlavy druhého seznamu; od obou seznamů následně odpojíme jejich hlavy. Takto pokračujeme, dokud jeden nebo oba z našich dvou seznamů nejsou prázdné. Pokud je jeden z nich neprázdný, nezapomeneme ho připojit na konec výsledného seznamu.

Vytvořit celý program je nyní již snadné: Nejprve ze souboru `vahy.in` načteme počet mincí N. Poté zavoláme proceduru `porovnej` s parametry `prvni` = 1 a `posledni` = N. Nakonec vypíšeme čísla v listech stromů (zleva doprava) ve výsledném seznamu; každý strom vypíšeme na samostatný řádek souboru `vahy.out`, a to v pořadí, v jakém stromy odpovídají prvkům seznamu. Čísla na každém řádku jsou setříděna (z konstrukce stromů patřícím prvkům seznamu) a hmotnosti mincí v pořadí dle řádků jsou rostoucí (dle vlastností vytvářeného seznamu). Zbývá si rozmyslet, že náš algoritmus neprovádí zbytečné volání funkce `porovnej`, a určit jeho časovou složitost.

Nejprve dokážeme indukcí dle délky intervalu určeného parametry při volání procedury `vytvor`, že náš program neprovádí zbytečné volání funkce `porovnej`. Procedura `vytvor` volá funkci `porovnej` pouze na dvojice prvků z intervalu specifikovaného parametry procedury `vytvor`. Po-

kud je tento interval jednoprvkový, dokazované tvrzení platí z triviálních důvodů. V opačném případě se nejprve vytvoří dva seznamy rekurzivním voláním procedury `vytvor` a ty se následně sloučí. Při slučování dvou seznamů je funkce `porovnej` volána pouze na dvojice mincí z různých seznamů (tedy výsledek takového volání není určen výsledky volání funkce `porovnej` při rekurzi). Vzhledem k tomu, že porovnáváme z každého seznamu minci s nejmenší vahou (a mince s menšími vahami jsme zařadili již do výsledného seznamu), nemůže být vztah hmotností mincí z dotazované dvojice určen předchozími dotazy. Můžeme tedy uzavřít, že žádné volání funkce `porovnej` není zbytečné.

Hloubka rekurzivního volání procedury `vytvor` je $O(\log N)$ (N je počet mincí), neboť při každém volání se délka intervalu specifikovaného parametry funkce zmenší na polovinu. Na sloučení dvou seznamů je třeba čas úměrný délce výsledného seznamu. Protože na každé úrovni volání se libovolná mince vyskytuje právě v jednom seznamu, je čas strávený algoritmem během procedur `vytvor` na jedné úrovni rekurze lineární, tj. $O(N)$. Celková časová složitost je tedy $O(N \log N)$. Libovolná mince se vyskytuje při běhu programu vždy právě v jednom seznamu, a tedy paměťová složitost programu je $O(N)$.

```
#include <stdio.h>
#include <stdlib.h>
#include "vahy_lib.h"

struct tuzel {
  int prvek;
  struct tuzel *levy, *pravy;
  };
struct tseznam {
  struct tuzel *strom;
  struct tseznam *dalsi;
  };
struct tseznam *vytvor(int prvni, int posledni) {
  struct tseznam *vysledek, *seznam1, *seznam2, **ocas, *pomocna;
  if (prvni==posledni) {
    vysledek=malloc(sizeof(struct tseznam));
    vysledek->dalsi=NULL;
    vysledek->strom=malloc(sizeof(struct tuzel));
    vysledek->strom->prvek=prvni;
    vysledek->strom->levy=vysledek->strom->pravy=NULL;
    return vysledek;
    }
  seznam1=vytvor(prvni,(prvni+posledni)/2);
  seznam2=vytvor((prvni+posledni)/2+1,posledni);
  ocas=&vysledek;
  while (seznam1&&seznam2) {
    switch (porovnej(seznam1->strom->prvek,seznam2->strom->prvek)) {
    case 0:
```

```
      (*ocas)=malloc(sizeof(struct tseznam));
      (*ocas)->strom=malloc(sizeof(struct tuzel));
      (*ocas)->strom->prvek=seznam1->strom->prvek;
      (*ocas)->strom->levy=seznam1->strom;
      (*ocas)->strom->pravy=seznam2->strom;
      pomocna=seznam1; seznam1=seznam1->dalsi; free(pomocna);
      pomocna=seznam2; seznam2=seznam2->dalsi; free(pomocna);
      break;
    case 1:
      *ocas=seznam1; seznam1=seznam1->dalsi;
      break;
    case -1:
      *ocas=seznam2; seznam2=seznam2->dalsi;
      break;
      }
    ocas=&((*ocas)->dalsi);
    }
  *ocas=seznam1?seznam1:(seznam2?seznam2:NULL);
  return vysledek;
  }
void vypis_strom(FILE *soubor, struct tuzel *uzel) {
  if (uzel->levy) {
    vypis_strom(soubor,uzel->levy);
    vypis_strom(soubor,uzel->pravy);
    }
  else
    fprintf(soubor,"%d ",uzel->prvek);
  }
void vypis_seznam(FILE *soubor, struct tseznam *seznam) {
  while (seznam) {
    vypis_strom(soubor, seznam->strom);
    fprintf(soubor,"\n");
    seznam=seznam->dalsi;
    }
  }

int main(void) {
  FILE *soubor;
  int N;
  struct tseznam *seznam;
  soubor=fopen("vahy.in","r");
  fscanf(soubor,"%d",&N);
  fclose(soubor);
  seznam=vytvor(1,N);
  soubor=fopen("vahy.out","w");
  vypis_seznam(soubor,seznam);
  fclose(soubor);
  return 0;
  }
```