```
  i:=M;
  while A[i]=0 do i:=i-1;
  writeln('Maximální hmotnost: ',i);
  write('Předměty v množině:');
  while A[i]<>-1 do
    begin
      write(' ',A[i]);
      i:=i-hmotnost[A[i]];
    end;
  writeln;
end.
```

Náš druhý příklad bude z oblasti grafových algoritmů, tzv. Floyd-Warshallův algoritmus pro nalezení nejkratších cest mezi všemi vrcholy grafu. My se však pokusíme bez definice grafu jak v zadání, tak v řešení tohoto příkladu obejít.

Vstupem algoritmu je $N$ měst. Mezi některými dvojicemi měst vedou (obousměrné) silnice, jejichž délky jsou dány na vstupu. Předpokládáme, že silnice se jinde než ve městech nepotkávají (pokud se kříží, tak mimoúrovňově). Úkolem je spočítat nejkratší vzdálenosti mezi všemi dvojicemi měst, tj. délky nejkratší cest mezi všemi dvojicemi měst. Cestou rozumíme posloupnost měst spojených silnicemi a délkou cesty součet délek silnic, které spojují po sobě následující města.

Jsou sice známy i trošičku rychlejší způsoby řešící popsaný problém (umí se $O(N^2 \log N + N \cdot M)$), ale výhoda popisovaného algoritmu je v tom, že je velmi krátký a jednoduchý.

Na začátku jsou uloženy vzdálenosti mezi městy ve dvourozměrném poli $D$, tj. $D[i][j]$ je vzdálenost z města $i$ do města $j$. Pokud mezi městy $i$ a $j$ nevede žádná silnice, bude $D[i][j] = \infty$, v praxi tedy nějaké dostatečně velké číslo. V průběhu výpočtu si budeme na pozici $D[i][j]$ udržovat délku nejkratší dosud nalezené cesty.

Samotný algoritmus se skládá z $N$ fází. Na konci $k$-té fáze bude v $D[i][j]$ uložena délka nejkratší cesty mezi městy $i$ a $j$, která může procházet skrz libovolné z měst $1, \ldots, k$. V průběhu $k$-té fáze tedy stačí vyzkoušet, zda mezi městy $i$ a $j$ je kratší stávající cesta přes města $1, \ldots, k-1$, jejíž délka je uložena v $D[i][j]$, anebo nová cesta přes město $k$. Pokud nejkratší cesta prochází přes město $k$, pak její část do města $k$ je nejkratší cesta z $i$ do $k$ přes města $1, \ldots, k-1$ a její část z města $k$ je nejkratší cesta z $k$ do $j$ přes města $1, \ldots, k-1$. Délka takové cesty je tedy rovna $D[i][k] + D[k][j]$. Pokud je tedy součet $D[i][k] + D[k][j]$ menší než stávající hodnota $D[i][j]$, nahradíme hodnotu na pozici $D[i][j]$ tímto součtem.

Z popisu přímo plyne, že po $N$-té fázi je na pozici $D[i][j]$ uložena délka nejkratší cesty z města $i$ do města $j$. Každá z $N$ fází algoritmu vyžaduje čas $O(N^2)$, takže celková časová složitost bude $O(N^3)$. Paměťová složitost algoritmu je $O(N^2)$. Popišme si ještě, jak bychom postupovali, kdybychom kromě vzdáleností mezi městy chtěli nalézt i samotné nejkratší cesty. To lze jednoduše vyřešit například tak, že si budeme udržovat další pomocné pole $E[i][j]$, do kterého při změně hodnoty $D[i][j]$ uložíme nejvyšší číslo města na cestě z $i$ do $j$ délky $D[i][j]$ (při změně v $k$-té fázi je to číslo $k$). Máme-li pak vypsat nejkratší cestu z $i$ do $j$, vypíšeme nejprve cestu z $i$ do $E[i][j]$ a pak cestu z $E[i][j]$ do $j$. Tyto cesty nalezneme stejným (rekurzivním) postupem.

```
var N:word; { počet měst }
    D:array[1..N] of array[1..N] of longint;
      { délky silnic mezi městy, D[i][i]=0,
        místo neexistujících je "nekonečno" }
    i,j,k:word;
begin
  for k:=1 to N do
    for i:=1 to N do
      for j:=1 to N do
        if D[i][k]+D[k][j] < D[i][j] then
          D[i][j]:=D[i][k] + D[k][j];
end.
```

Na rozmyšlenou:

- Jak byste algoritmus modifikovali, kdyby silnice byly jednosměrné?
- Nastavit $\infty$ na `maxint` je sice lákavé, ale špatně, protože $\infty + \infty$ by pak mohlo přetéci. Pomůže `maxint div 2`.
- Hodnoty v poli si sice přepisujeme pod rukama, takže by se nám mohly poplést hodnoty z předchozí fáze s těmi z fáze současné. Ale zachrání nás to, že čísla, o která jde, vyjdou v obou fázích stejně.
- Popis algoritmu vysloveně svádí k „rejpnutí“: „Jak víme, že spojením dvou cest, které provádíme, vznikne zase cesta (tj. že se na ní nemohou nějaké vrcholy opakovat)?“ Inu, to samozřejmě nevíme, ale všimněte si, že kdykoliv by to cesta nebyla, tak si ji nevybereme, protože původní cesta bez vrcholu $k$ bude vždy kratší nebo alespoň stejně dlouhá ... tedy alespoň pokud se v naší zemi nevyskytuje cyklus záporné délky. (Což, pokud bychom chtěli být přesní, musíme přidat do předpokladů našeho algoritmu.)
- Pozor na pořadí cyklů – program vysloveně svádí k tomu, abychom psali cyklus pro `k` jako vnitřní ... jenže pak samozřejmě nebude fungovat.

Svá řešení nám zasílejte do 18. října 2004 buďto elektronicky nebo na adresu:

**Korespondenční seminář z programování**
**KSVI MFF UK**
**Malostranské náměstí 25**
**118 00 Praha 1**

**Tips & Tricks: Z letáků KSP si můžete složit knížečku**



**Zadání první série sedmnáctého ročníku KSP**

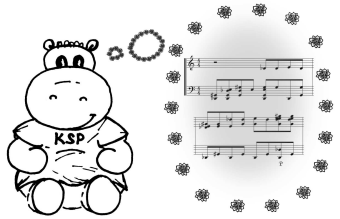
## 17-1-1 Výdělek bratří Součků — 10 bodů

Bratři Součkovi, Ain a Kábel, potomci známého velikého Suka, byli odmala talentovaní hudebníci. Jejich vzájemný vztah bohužel byl, jak jejich jména kážou, dosti špatný. I v dospělém věku si oba konkurovali jako hudební kritici.

Při vzácné příležitosti vystoupení známého zpěváka Miguela J. X. Sona byli oba bratři Součkovi firmou Granny najati, aby se pokusili odposlechnout J. X. Sonův největší hit. Oba bratři – každý sám – koncert navštívili a když se do Sonova hitu zaposlouchali, zjistili, že se neustále opakuje. Tak si oba poznamenali jeho začátek až do chvíle, kdy si byli jisti, že celý hit je jen opakování jimi zaznamenaného začátku.

Při odevzdání svých záznamů ale zjistili, že jsou různě dlouhé! Oba však ale trvají na tom, že zaznamenali skladbu správně, a obviňují toho druhého. Vedoucí firmy Granny, paní Babičková, si však myslí, že ačkoliv jsou jejich zápisy různě dlouhé, mohly by představovat jedinou skladbu. A Vás poprosila, jestli byste jejich zápisy mohli porovnat.

Na vstupu dostanete Ainův i Kábelův záznam. Každý se skládá z délky a pak z jednotlivých not, které budeme pro jednoduchost zapisovat přirozenými čísly. Úkolem Vašeho programu je říci, zda posloupnost, která vznikne jako nekonečné opakování Ainova zápisu je *stejná* jako ta, která vznikne jako nekonečné opakování zápisu od Kábela.




*Příklad:* Pokud je Ainův záznam $1, 2, 1, 2, 1, 2$ a Kábelův $1, 2, 1, 2$, zaznamenali oba bratři skladbu stejně. Pokud by Ain zaznamenal $1, 2, 1, 2, 3, 2$, nebylo by tomu tak.

## 17-1-2 Bůhdhova odměna — 10 bodů

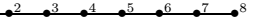
Když známý Tigamský kupec Semtodaj Čornulaj Apadaj, věrný reprezentant svého národa, prodal další kus „své“ Tigamské plošiny, rozhodl se Bůhdha, že už se na to nemůže dál koukat. Ovšem jeho hlasité „Budiž černočerná tma!“ se minulo účinkem a vrátilo ozvěnou. (Přeci jenom Bůhdha nemůže být všemocný; kdyby mohl, dokázal by vytvořit neřešitelný problém, který by nevyřešil ani on sám – ale pak by nebyl všemocný. *QED.*)

A tak si usmyslel, že Tigamany alespoň odmění – obmění jejich jazyky. A to tak, aby si žádní obyvatelé dvou sousedních vesnic nerozuměli. Sousední vesnice jsou takové, mezi kterými vede (samozřejmě horská) pěšina. A protože jsme v horách, žádné dvě pěšiny se nekříží.

Ubohý Bůhdha ale dokázal vymyslet jen 6 odlišných jazyků. Zklamán dosavadními neúspěchy se raději obrátil na Vás, abyste zjistili, zda je možné jeho ďa... božský plán provést.

Máte napsat program, který dostane na vstupu popis Tigamské říše – počet vesnic $N$ a dále $M$ pěšin, každá z nich spojuje právě dvě vesnice. Každá pěšina je obousměrná a navíc platí, že žádné dvě pěšiny se mimo vesnice nekříží (ani nadjezdem, natož tunelem). Úkolem programu je zjistit, zda je možno přiřadit každé vesnici jeden z šesti jazyků tak, aby si žádní obyvatelé sousedních vesnic nerozuměli. Pokud to jde, má vypsat jedno takové přiřazení.

*Příklad:* Pro následující situaci




stačí Bůhdhovi dokonce jen dva jazyky – rozdá je střídavě.

## 17-1-3 Chmatákův lup — 10 bodů

Cecil Hromdotruhlice, Mistr Antibankovních Technologií Álias Kraďas byl zářným potomkem svého otce. Zdědil po něm vše dobré, co měl a co se tak za nehet vešlo, ale také všechno špatné. Včetně svého povolání. A ne ledasjakého povolání. Cecil je totiž profesionální antibankovní činitel – to znamená, že bohatým bere a chudým koneckonců taky. Sice už nezbyl nikdo, komu by mohl dávat, než on sám, ale s touto nepříjemností se už všichni Hromdotruhlíkové dávno smířili.

Jednoho dne se Cecil vydal na prohlídku jedné obzvláště bohaté banky v přestrojení za hygienika telefonních sluchátek. Uvnitř ke svému Hromovému překvapení zjistil, že není schopen všechny cenné věci odnést! Chtěl by ale dostát své antibankovní cti a obrat banku o co nejvíc peněz.

Cecil dokáže unést nanejvýš (spíše nanejtíž) $N$ kg lupu. V bance je $P$ cenných věcí a u každé odhadl Cecil její hmotnost na $m_i$ celých kg a cenu na $c_i$ zlaťáků. Cena, na rozdíl od váhy, může být i desetinné číslo.

Zkuste napsat program, který po zadání popsaných údajů poradí Cecilovi, jaké předměty vzít, aby je ještě unesl a přitom jejich celková cena byla *největší možná*.

*Příklad:* Pokud dokáže Cecil unést $N = 8$ kg a v bance jsou

| $i$ | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| $m_i$ | 5 | 4 | 3 | 2 |
| $c_i$ | 12.5 | 10 | 6 | 7.5 |

tyto $P = 4$ cennosti, je pro Cecila nejlepší odnést věci 1 a 4. Pokud by ale byla jeho nosnost o kilogram větší ($N = 9$), bylo by nejlepší odnést předměty 2, 3 a 4.

## 17-1-4 Paloučkova výhra — 10 bodů

Ludvík Palouček, známý to milovník přírody, byl svým přítelem Pepou Běhavým vyzván k běžeckému závodu, který se má odehrát v Běhavého rodném městě. Ludvík se závodu nebojí, protože jeho přítel dostal jméno spíš po svých zažívacích potížích než kvůli rychlým nohám, ale nechce se mu trávit mnoho času jinde než na svém paloučku: „A jak dlouho to bude, Pepo, trvat?“ „Ale, stačí jedno kolečko,“ odpověděl mu vítězství chtivý kamarád.

Ludvík se této odpovědi chytl a rozhodl se naplánovat trasu závodu sám. Závod má začínat a končit na jednom místě (Pepa chtěl kolečko) a přitom má být co nejkratší, aby mohl být Ludvík co nejdřív doma. Když ale uviděl mapu města, zhrozil se a raději Vás požádal o pomoc.

Na vstupu dostanete popis Běhavého města: $N$, což je počet křižovatek, a dále $M$ ulic. Každá ulice je obousměrná, má

nějakou délku a spojuje dvě křižovatky. Ačkoliv se ulice mimo křižovatky nekříží, ve městě může být mnoho nadjezdů a tunelů.

Vaším úkolem je zjistit, zda ve městě existuje nějaký *okruh*, a pokud ano, máte najít a vypsat libovolný *nejkratší* z nich i s jeho délkou. Okruh je posloupnost alespoň dvou neopakujících se ulic, přičemž po sobě následující ulice okruhu začínají a končí na stejné křižovatce – včetně první a poslední ulice okruhu. Délkou okruhu rozumíme součet délek všech jeho ulic.

*Příklad:* Pokud je v městě $N = 5$ křižovatek a ulice

| odkud | kam | délka |
|---|---|---|
| 1 | 2 | 2 |
| 2 | 3 | 3 |
| 1 | 3 | 9 |
| 3 | 4 | 1 |
| 4 | 5 | 3 |
| 1 | 5 | 2 |

tak nejkratší je okruh $1 \to 2 \to 3 \to 4 \to 5 \to 1$ délky 11. Všimněte si, že $1 \to 2 \to 1$ není okruh, protože je skládá z jediné opakující se ulice (a to se nesmí).

## 17-1-5 Jazykozpytcův poklad — 10 bodů

Co mají společného překladače programovacích jazyků, vyhledávání v textu, komprese dat nebo třeba také rozdělování slov? Na první pohled nepříliš, ale teoretickým informatikům se přesto podařilo najít teorii, která shrnuje základní věci z těchto oblastí (a mnohých jiných) a říká o nich mnoho zajímavého. Je to teorie automatů a formálních jazyků a právě té jsme se rozhodli věnovat náš letošní seriál.

Začneme nejprve názvoslovím:

- *Abeceda* je libovolná konečná množina znaků.
- *Slovo* $\alpha$ nad abecedou $A$ je uspořádaná konečná posloupnost znaků abecedy $A$. Prázdné slovo značíme $\lambda$. Množinu všech možných slov nad abecedou $A$ značíme $A^*$.
- *Jazyk* $L$ nad abecedou $A$ je nějaká podmnožina (klidně nekonečná) množiny $A^*$. Nenechte se zmást názvem jazyk, nemáme tím na mysli nějaký specifický programovací či dokonce přirozený jazyk (i když i tyto do naší definice spadají), jedná se zkrátka o nějakou množinu slov.
- Jsou-li $\alpha$ a $\beta$ dvě slova, pak zápisem $\alpha\beta$ rozumíme jejich zřetězení za sebe.
- Zápisem $\alpha^i$ rozumíme $i$-násobné opakování slova $\alpha$ (tj. třeba $(ab)^2 = abab$).

*Příklad:* nad abecedou $\{a, b, c\}$ lze vybudovat třeba jazyky $\{baba, abba, bac\}$ (ten je konečný) či $\{a^i b^i;\ \forall i \in \mathbf{N}\}$ (ten je nekonečný a patří do něj třeba slova $ab$ či $aaabbb$, nepatří tam $abb$ ani $bbbaaa$).

U každého jazyka lze studovat například tyto dvě věci: jak daný jazyk *rozpoznávat* (rozhodnout o zadaném slovu, zda patří do jazyka) a jak *generovat* všechna slova daného jazyka. K prvnímu úkolu slouží „stroje“ čili *automaty*, s jejichž nejběžnějšími typy se v seriálu seznámíme. To druhé mají na starost *gramatiky*. Gramatika je formální popis pravidel, pomocí kterých se vytvářejí všechna slova daného jazyka. Původně je vymyslel lingvista pan Chomsky pro popis přirozených jazyků – z hodin českého jazyka jistě znáte větné rozbory, tj. pravidla typu [věta] → [podmětná část][přísudková část], kde podmětná a přísudková část se opět rozpadají na podčásti, atd. Gramatika se tedy skládá ze sady přepisovacích pravidel $\alpha \to \beta$, kde na obou stranách vystupují slova sestávající se jednak z pomocných symbolů (těm se říká *neterminální*) a jednak ze symbolů *terminálních* (po domácku *terminálů*), které už se dále neexpandují (čili už se na ně dále nepoužívají přepisovací pravidla). Terminály se vlastně dají chápat jako jednotlivé znaky použité abecedy.

Formální definice je následující: Gramatikou nazveme čtveřici $(V_N, V_T, S, P)$, kde:

- $V_N$ je konečná množina neterminálních symbolů,
- $V_T$ je konečná množina terminálních symbolů,
- $S \in V_N$ je počáteční neterminální symbol,
- $P$ je konečný systém přepisovacích pravidel $\alpha \to \beta$, kde $\alpha, \beta \in (V_N \cup V_T)^*$ a $\alpha$ obsahuje alespoň jeden neterminální symbol. Dvě pravidla $\alpha \to \beta$ a $\alpha \to \gamma$ obvykle zkráceně zapisujeme jako $\alpha \to \beta \mid \gamma$.

Gramatika vezme počáteční symbol a začne ho expandovat (nahrazovat) podle některého z uvedených pravidel. Typicky bývá několik možností, jak expandovat, tehdy můžeme použít libovolné vhodné pravidlo. Expanze končí, když z expandovaného řetězce vymizí všechny neterminální symboly. Všechna možná slova, která pomocí jedné gramatiky $G$ můžeme různými posloupnostmi expanzí dostat, tvoří *jazyk gramatiky,* ten budeme značit $L(G)$.

Jako příklad si uvedeme gramatiku, která popisuje jazyk všech aritmetických výrazů s čísly 1 a 2 používajících operace + a ∗ a závorky. Použijeme neterminální symboly $V_N = \{V, T, F\}$, terminální symboly $V_T = \{1, 2, +, *, (, )\}$, počáteční symbol je $V$ a pravidla:

$$V \to T + V \mid T$$
$$T \to F * T \mid F$$
$$F \to (V) \mid 1 \mid 2.$$

Například výraz $1+2*2$ je generován posloupností přepisů $V \to T+V \to F+V \to 1+V \to 1+T \to 1+F*T \to 1+F*F \to 1+2*F \to 1+2*2$. Slovo 22++1 zjevně pomocí sady našich pravidel nevytvoříme.

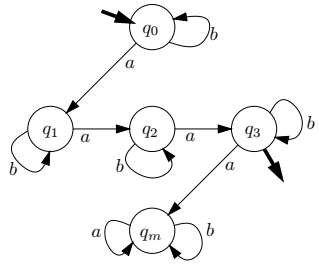
V prvním dílu seriálu se seznámíme s nejjednodušší rodinou jazyků – *regulárními* jazyky. Regulární jazyk je takový jazyk, ke kterému existuje *konečný automat,* který ho rozpoznává.

Co že to ten konečný automat (též zkratkou KA) vlastně je? Matematici mají rádi nejrůznější uspořádané $k$-tice, formálně si proto konečný automat zavedeme jako pětici $(Q, A, \delta, q_0, F)$, kde:

- $Q$ je konečná množina stavů stroje,
- $A$ abeceda, nad kterou stroj pracuje,
- $\delta : Q \times A \to Q$ je tzv. *přechodová funkce*, která ke každé kombinaci stavu a načteného znaku určuje nový stav, do kterého automat přejde,
- $q_0 \in Q$ je počáteční stav,
- $F \subseteq Q$ je množina koncových (přijímajících) stavů.

A nyní lidsky: konečný automat je stroj, který dostane na vstupu nějaké slovo a má se o něm rozhodnout, zda ho přijme či nikoliv. Automat se může nacházet v konečné a předem dané množině stavů $Q$, na začátku dejme tomu ve stavu $q_0$. V každém kroku své činnosti načte jeden znak ze vstupu a podle tohoto znaku se rozhodne, do jakého stavu přejde. To je dáno přechodovou funkcí, která k aktuálnímu stavu $q$ a znaku $a$ vrátí nový stav $q'$, tedy $\delta(q, a) = q'$. Pokud po přečtení všech znaků slova automat skončil v některém z přijímacích stavů z množiny $F$, říkáme, že slovo bylo přijato, jinak bylo odmítnuto. Všechna slova, která daný automat $A$ přijímá, tzv. *jazyk automatu,* značíme $L(A)$.

*Příklad:* automat nad abecedou $\{a, b\}$, přijímající všechna slova s právě třemi výskyty znaku $a$ a libovolným počtem výskytů znaku $b$. Automaty je nejpřehlednější zapisovat obrázkem:




Automat má 5 stavů, stav $q_0$ je počáteční, stav $q_3$ je jediný přijímací. Stav $q_i$ nám vlastně značí, že doposud jsme načetli $i$ znaků $a$, stav $q_m$ je záchytný a znamená, že $a$-ček už jsme přečetli moc.

V následujících dílech seriálu si představíme více jazykových rodin, ukážeme si jak jejich příslušné rozpoznávající stroje (tzv. akceptory), tak také odpovídající typy gramatik. Například regulárním jazykům odpovídají gramatiky obsahující pouze pravidla ve tvaru $X \to \alpha Y$, $X \to \alpha$, kde $X, Y \in V_N$ a $\alpha \in V_T^*$.

Ale nyní již

**Soutěžní úlohy:**

1. Uvažme abecedu $A = \{0, 1\}$. Slovo nad touto abecedou bude kódovat číslo zapsané v dvojkové soustavě, s obvyklou konvencí, tj. nejvýznamnější bit nalevo, nejméně významný napravo. Sestrojte konečný automat nad $A$ rozpoznávající všechna čísla dělitelná třemi a nedělitelná dvojkou (tj. jeho jazykem budou všechna slova kódující číslo dělitelné 3 a nedělitelné 2). *[5 bodů]*

2. Sestrojte gramatiku se stejným jazykem jako v první úloze – tj. generující právě čísla v binárním zápisu, která jsou dělitelná třemi a nejsou dělitelná dvojkou. *[5 bodů]*

Kromě zkonstruovaného automatu a gramatiky by měl být součástí řešení i stručný slovní popis toho, proč daný automat resp. gramatika dělá to co má, případně důkaz, že hledáme marně a to, co chceme, neexistuje.

## Recepty z programátorské kuchařky


I v následujícím ročníku KSP vám kromě úloh budeme servírovat recepty z programátorské kuchařky. V první kuchařce nového ročníku si povíme něco o jedné z nejpoužívanějších programátorských technik, tzv. *dynamickém programování.* Dynamickým programováním rozumíme takový postup, kdy vyřešíme zadanou úlohu nejprve pro zadání menší velikosti a pak nalezená řešení zkombinujeme dohromady, abychom získali řešení původní úlohy. Techniku dynamického programování si předvedeme na dvou (učebnicových) příkladech.

Prvních z našich dvou příkladů je úloha známá jako *problém batohu.* Je dáno $N$ předmětů o hmotnostech $m_1, \ldots, m_N$ a dále je dáno číslo $M$ (nosnost batohu). Úkolem je vybrat některé z předmětů tak, aby součet jejich hmotností byl co největší, ale zároveň nepřekročil $M$. My si popíšeme algoritmus, který tento problém řeší, s časovou složitostí $O(MN)$.

Náš algoritmus bude používat pomocné pole $A[0 \ldots M]$ a jeho činnost bude rozdělena do $N$ kroků. Na konci $k$-tého kroku budou nenulové hodnoty v poli $A$ právě na těch pozicích, které odpovídají součtu hmotností předmětů z nějaké podmnožiny prvních $k$ předmětů. Před prvním krokem (po nultém kroku), jsou všechny hodnoty $A[i]$ pro $i > 0$ nulové a $A[0]$ má nějakou nenulovou hodnotu, řekněme $-1$. Všimněme si, že kroky algoritmu odpovídají podúlohám, které řešíme: nejdříve vyřešíme podúlohu tvořenou jen prvním předmětem, pak prvními dvěma předměty, prvními třemi předměty, atd.

Popišme si nyní $k$-tý krok algoritmu. Pole $A$ budeme procházet od konce, tj. od $i = M$ po $i = m_k$. Pokud je hodnota $A[i]$ stále nulová, ale hodnota $A[i - m_k]$ je nenulová, změníme hodnotu uloženou v $A[i]$ na $k$. Rozmysleme si, že po provedení $k$-tého kroku odpovídají nenulové hodnoty v poli $A$ hmotnostem podmnožin z prvních $k$ předmětů, pokud před jeho provedením nenulové hodnoty odpovídaly hmotnostem podmnožin z prvních $k-1$ předmětů. Pokud je hodnota $A[i]$ nenulová, pak buď byla nenulová před $k$-tým krokem (a v tom případě odpovídá hmotnosti nějaké podmnožiny prvních $k-1$ předmětů) anebo se stala nenulovou v $k$-tém kroku. Potom ale existuje podmnožina prvních $k-1$ předmětů, jejíž hmotnost je $i - m_k$, ke které stačí přidat $k$-tý předmět, abychom našli podmnožinu hmotnosti přesně $i$. Naopak, pokud lze vytvořit podmnožinu $I$ hmotnosti $m$, pak $I$ je buď tvořena jen prvními $k-1$ předměty a tedy hodnota $A[m]$ je nenulová již před $k$-tým krokem, anebo $k \in I$. Potom ale hodnota $A[m - m_k]$ je nenulová před $k$-tým krokem (hmotnost podmnožiny $I \setminus \{k\}$ je $m - m_k$) a hodnota $A[m]$ se stane nenulovou v $k$-tém kroku.

Po provedení všech $N$ kroků odpovídají nenulové hodnoty $A[i]$ přesně hmotnostem podmnožin ze všech předmětů, co máme k dispozici. Speciálně největší index $i_0$ takový, že hodnota $A[i_0]$ je nenulová, odpovídá hmotnosti nejtěžší podmnožiny předmětů, která nepřekročí hmotnost $M$. Nalézt jednu množinu této hmotnosti také není obtížné: v $A[i_0]$ je uloženo číslo jednoho z předmětů nějaké takové podmnožiny, v $A[i_0 - m_{A[i_0]}]$ číslo dalšího předmětu, atd. Samotný kód našeho algoritmu lze nalézt níže.

Časová složitost algoritmu je $O(NM)$, neboť se skládá z $N$ kroků, z nichž každý vyžaduje čas $O(M)$. Paměťová složitost činí $O(N + M)$, což představuje paměť potřebnou pro uložení pomocného pole $A$ a hmotností daných předmětů.

```
var N: word; {   počet předmětů   }
    M: word; { hmotnostní omezení }
    hmotnost: array[1..N] of word;
              { hmotnosti daných předmětů }
    A: array[0..M] of integer;
    i, k: word;
begin
  A[0]:=-1;
  for i:=1 to M do A[i]:=0;
  for k:=1 to N do
    for i:=M downto hmotnost[k] do
      if (A[i-hmotnost[k]]<>0) and (A[i]=0) then
        A[i]:=k;
```