P-III-1 Hračkářství

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 projité 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 projité 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 Knihovna

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 $\sum_{i=1}^{i_1} t_i \leq s$; je jasné, že i_1 je maximální možný počet knih, které lze umístit do první poličky. Poté nalezneme největší i_2 takové, že $\sum_{i=i_1+1}^{i_2} t_i \leq s$, 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, $1 \leq k \leq P$, ve funkci `existuje`. Za tímto účelem si nejprve vytvoříme pomocné pole, ve kterém budou uloženy součty tlouštěk prvních j knih pro $1 \leq j \leq N$. Při počítání hodnoty i_k metodou půlení intervalu vyhledáme v tomto pomocném poli největší číslo i' takové, že $\sum_{i=1}^{i'} t_i - \sum_{i=1}^{i_{k-1}} t_i \leq s$; 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 Reverzibilní výpočty: Sčítání

První řešení: Inspirujeme se tradičním algoritmem pro sčítání čísel „pod sebou“ a uvědomíme si, že není závislý na použité číselné soustavě (zvědavější povahy obětují 5 minut na důkaz indukcí). Pokud bychom uměli spočítat přenosy mezi řády, je samotné sečtení triviální: $A'_i = A_i \operatorname{xor} B_i \operatorname{xor} P_i$, 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 si čas strávený touto procedurou $T(l)$, bude platit $T(l) = 1 + 3 \cdot T(l/2)$: 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 $T(l) = 1+3\cdot(1+3\cdot T(l/4)) = 1+3+9\cdot T(l/4)$ a když budeme dosazovat dál, po k krocích dojdeme k $T(l) = 1+3+\ldots+3^{k-1}+3^k\cdot T(l/2^k)$. My ale víme, že $T(1) = 1$, takže pro $k = \log_2 l$ (naše hloubka rekurze) vyjde $T(l) = 1 + 3 + \ldots + 3^{\log l - 1} + 3^{\log_2 l}$. To je ovšem geometrická řada se součtem $(3^{k+1}-1)/2 = O(3^k) = O(3^{\log_2 l})$, což můžeme ještě zjednodušit: $3^{\log_2 l} = (2^{\log_2 3})^{\log_2 l} = 2^{\log_2 3 \cdot \log_2 l} = (2^{\log_2 l})^{\log_2 3} = l^{\log_2 3} \leq l^{1.59}$. 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 počítali zbytečně 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 sama sečte horní polovinu čísla a nakonec rekurzivně zavolá sebe sama 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 zařídit snadno 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 konstatní, nýbrž logaritmická, zatímco druhé řešení by se dalo snadno upravit tak, aby mělo stále logaritmickou složitost (stačí si uvědomit, že jej lze naprogramovat nerekurzivně, čímž se zbavíme prostoru na lokální proměnné). Ovšem časové složitosti zůstanou zachovány, takže druhé řešení bude pracovat v témže prostoru rychleji.

P-III-4 Poklad kapitána Flinta

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-úhelníka, 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-úhelníka, 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 projití všech vrcholů 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 přejití 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 mnohoúhelníka */

  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 Vážení

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 stromě 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 stromě).

Základem programu bude rekurzivní procedura `vytvor(prvni,posledni)`, která vytvoří jednosměrný spojový seznam popsaný v prvním odstavci. 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 polovičních 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 stromě hlavy (tj. prvního prvku) prvního ze seznamů a na některou z mincí ve stromě 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`. Pokud 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;
  }
```