Милан Вугделија

О УЛОЗИ МАТЕМАТИКЕ У НАСТАВИ ПРОГРАМИРАЊА

Међу ученицима средњих школа који се заинтересују за програмирање, математика се често схвата као досадна, апстрактна и неприменљива гимнастика ума, која нема много везе са писањем програма. Међу некима од ових ученика као да влада мишљење да се велики програмер постаје тако што се научи напамет шта који интерапт ради, или са ког порта се очитавају које информације о поједином уређају. Многе од њих више интересује како оставити неку врсту потписа на рачунару, тако да свима буде јасно ко је то урадио, а мало коме — како је то урадио или како уклонити „потпис“. Наравно да ови ђаци таквим знањем одскачу од осталих ђака у одељењу, а неретко изазивају дивљење или завист, што их чини важнима. Кад је овакво знање довољно да се буде важан, чему онда учење математике? Математика се теже савладава, јер захтева више напора ума, а и нема баш видљиве везе са одушевљењем које осетите када на рачунару урадите нешто (макар и деструктивно) што остали не умеју.

У овом тексту ће бити наведено неколико задатака, који се сви могу задавати у средњим школама. Сви ови задаци се применом математике могу решити на елегантнији, једноставнији и/или ефикаснији начин, тако да се за мање времена долази до решења које је по много чему супериорније од оног, које би се добило без потребног знања математике. Ученик који реши задатак на лошији начин, може бранити своје решење прилично упорно, јер оно заиста ради лепо на примерима који се могу ручно проверити, а ако на мало већим примерима ради споро, треба набавити бржи рачунар. Оно што обично није јасно, је да разлике могу бити толике, да лошије решење за нешто веће улазне податке може бити потпуно неупотребљиво.

Примена математике у програмирању даје прилику ученицима који се определе за овакав начин рада, да се и они похвале знањем, које је у програмирању бар исто толико важно као и оно поменуто, па да тако и они буду важни. Програми ових ученика ће моћи много тога што не могу програми њихових другова, који можда познају више техничких детаља о рачунару, више разних програмских окружења и библиотека (што су без сумње корисна знања), али слабије знају или слабије користе математику. При томе неће бити додатног посла и главобоље за људе који по школама одржавају рачунаре. Напротив, нека оригинална решења могу бити врло корисна у пракси.

Циљ овог текста је да код ученика заинтересованих за програмирање развије и подржи уверење, да се кроз примену математике могу истаћи управо у

програмирању, и то често на један здравији начин, него што је на пример, онемогућавање нормалног функционисања рачунара.

Принцип укључења-искључења

Средње вредности

Дата је тродимензиона матрица A димензија $N \times N \times N$. Исписати елементе и индексе свих елемената $a[i,j,k]$ матрице A, који су једнаки аритметичкој средини свих оних елемената $a[p,q,r]$ за које је $p \neq i$, $q \neq j$, $r \neq k$.

Задатак се лако може решити, тако што се за сваки елемент матрице A израчуна одговарајућа средња вредност, и упореди се са текућим елементом. Такво решење би захтевало три угњеждена циклуса за избор елемента који се проверава, и још три у дубину за рачунање средње вредности, што је укупно 6 циклуса. За N реда величине 100 такав алгоритам је тешко употребљив, јер је потребно извршити број операција реда величине 10^{12}, а то би вероватно трајало предуго.

Јасно је да се велики број сабирања у претходном решењу понавља. До потребне средње вредности можемо доћи и на други начин, а то је да од једном израчунатог збира свих елемената одузмемо оне којима се бар један индекс поклапа са одговарајућим индексом текућег елемента. Таквих елемената има знатно мање, тако да би за сабирање била довољна два циклуса уместо три. Међутим, да би се рачунање свело на минимум, можемо још неке потребне суме израчунати унапред, и тиме избећи било какве додатне циклусе. Тада ће се потребне средње вредности израчунавати једном наредбом. У ту сврху, потребне су нам:

- суме „слојева“ или „равни“ у сва три правца, односно:
  - за дато i – сума $a[i,j,k]$ за све j,k,
  - за дато j – сума $a[i,j,k]$ за све i,k,
  - за дато k – сума $a[i,j,k]$ за све i,j;
- затим, суме врста и колона, тј. „правих“ у сва три правца, односно:
  - за дато j,k – сума $a[i,j,k]$ за све i,
  - за дато k,i – сума $a[i,j,k]$ за све j,
  - за дато i,j – сума $a[i,j,k]$ за све k;
- и коначно сума свих елемената.

Ради памћења свих ових сума, погодно је опсег индекса $1\,..\,N$ по свакој димензији проширити на $0\,..\,N$. Елемент коме је неки од индекса једнак 0, користи се као сума свих полазних елемената матрице (оних са ненултим индексима), којима се одговарајући индекси поклапају са ненултим индексима тог елемента. Тако на пример, $a[0,j,k]$ једнак је суми „праве“ која се добија за све i и фиксиране j,k, а $a[0,j,0]$ је сума „равни“ свих $a[i,j,k]$ са датим j.

Остаје да видимо како се израчунава збир елемената $a[p,q,r]$, за које је $p \neq i$, $q \neq j$, $r \neq k$. Тај збир је могуће израчунати користећи познати принцип укључења-искључења. Најпре од суме свих елемената одузмемо суме $a[i,0,0]$ $a[0,j,0]$ $a[0,0,k]$, равни у којима се налазе елементи са бар једним истим индексом као код $a[i,j,k]$. Међутим, тада смо елементе са два поклопљена индекса одузели 2 пута, па их сада треба додати, тј. треба додати суме $a[i,j,0]$ $a[0,i,k]$ $a[i,0,k]$. При томе је сам елемент $a[i,j,k]$ додат $1-3+3=1$ пута, па га треба још једном одузети. Тако долазимо до тражене суме. Једнакост која се у програму проверава, добија се када сам елемент $a[i,j,k]$ и средњу вредност помножимо са $(N-1)^3$, јер толико има сабирака у суми. Тиме смо избегли потребу за дељењем и употребом реалних бројева.

```
var
  a:  array[0..30, 0..30, 0..30] of integer;
  i, j, k, n, n3:  integer;
begin
  write('n='); readln(n); n3:=(n-1)*(n-1)*(n-1);
  for i:=0 to n do
    for j:=0 to n do
      for k:=0 to n do
        if(i*j*k=0) then a[i,j,k]:=0
        else
        begin
          write('a[', i,',', j,',', k,']=');
          readln(a[i, j, k]);
          a[0,0,0]:=a[0,0,0]+a[i,j,k];
          a[i,0,0]:=a[i,0,0]+a[i,j,k];
          a[0,j,0]:=a[0,j,0]+a[i,j,k];
          a[0,0,k]:=a[0,0,k]+a[i,j,k];
          a[i,j,0]:=a[i,j,0]+a[i,j,k];
          a[0,j,k]:=a[0,j,k]+a[i,j,k];
          a[i,0,k]:=a[i,0,k]+a[i,j,k];
        end;
  for i:=1 to n do
    for j:=1 to n do
      for k:=1 to n do
        if (n3*a[i,j,k]=a[0,0,0]-(a[i,0,0]+a[0,j,0]+a[0,0,k])
              +a[i,j,0]+a[0,j,k]+a[i,0,k] -a[i,j,k]) then
          writeln('a[', i,',', j,',', k,']=', a[i,j,k]);
end.
```

Делимичне суме

На основу дате матрице A димензија $N \times N$, треба формирати матрицу B истих димензија, такву да је $b[i,j]$ једнак суми оних $a[p,q]$, за које је $i \leqslant p$, $i+j \leqslant p+q$.

Поново имамо једно једноставно и неефикасно решење: да за свако i, j, прођемо кроз потребан део матрице A, и израчунамо $b[i, j]$ са два помоћна циклуса:

```
var
  a,b:  array[1..100, 1..100] of integer;
  n, i, j, p, q, qlast:  integer;
begin
  write('n='); readln(n);
  for i:=1 to n do
    for j:=1 to n do
    begin
      write('a[', i,',', j,']=');
      readln(a[i, j]);
      b[i,j]:=0;
      for p:=1 to i do
      begin
        if n<(i+j-p) then
          qlast:=n
        else
          qlast := i+j-p;
        for q:=1 to qlast do
          b[i,j]:=b[i,j]+a[p,q];
      end;
    end;
  for i:=1 to n do
  begin
    for j:=1 to n do write(b[i,j]:8);
    writeln;
  end;
end.
```

Као и у претходном задатку, и овде имамо понављање истих сабирања. Многа од тих сабирања постају сувишна, ако искористимо претходно израчунате збирове. Употребићемо поново принцип укључења-искључења. Размотимо најпре случај када елемент $a[i, j]$ није ивични. Збирови $b[i, j-1]$ и $b[i-1, j+1]$ (означени на слици) садрже све сабирке из збира $b[i, j$, осим самог елемента $a[i, j]$. Међутим, сабравши ова два збира, елементе који се налазе у њиховом пресеку рачунамо два пута, па треба одузети збир $b[i-1, j]$, који представља тај пресек. Разрадом детаља, добијају се све потребне формуле (претпоставља се да је $N > 1$):

$$
\begin{aligned}
b[1,1] &= a[1,1],\\
b[1,j] &= b[1,j-1] + a[1,j] \quad (j > 1),\\
b[i,1] &= b[i-1,2] + a[i,1] \quad (1 < i \leqslant N),\\
b[i,j] &= b[i,j-1] - b[i-1,j] + b[i-1,j+1] + a[i,1] \quad (1 < i \leqslant N,\ 1 < j < N),\\
b[i,N] &= b[i,N-1] + a[i,N] \quad (1 < i \leqslant N).
\end{aligned}
$$

Ове формуле не само да дају могућност уштеде времена (двоструки циклус уместо четвороструког), него и простора, јер се сада све операције могу извести у истој матрици, ако се елементи b израчунавају ред по ред, слева на десно. На основу свега реченог, програм се лако може написати.

Похлепан алгоритам

У следећем примеру илуструје се доказивање коректности и примена такозваног похлепног (енгл. greedy) алгоритма:

Највећа зарада на пословима са роком

У јединици времена може да се обави један посао. Дато је N послова, који имају рокове: i-ти посао треба обавити најкаснije у тренутку (временској јединици) d_i, $1 \leqslant i \leqslant N$, d_i је природан број. Зарада на i-том послу је R_i новчаних јединица ($1 \leqslant i \leqslant N$) ако је посао обављен на време, иначе је 0. Наћи један редослед обављања послова тако да се максимизира зарада.

Напомена: обично постоји извесна зарада при сваком послу, па и при оном који касни. Да не бисмо губили на општости, зараду можемо раздвојити на фиксни и променљиви део. Фиксни део зараде је онај који се свакако остварује (минимална или гарантована зарада), а променљиви део добијамо када зараду на послу обављеном на време умањимо за вредност фиксне зараде. Након тога се можемо сконцентрисати на максимизацију променљивог дела зараде.

Проблем је формулисан нешто апстрактније, међутим лако се долази до разних интерпретација. На пример, у случају фотокопирнице *временска јединица* може бити 2 сата, у случају микропроцесора 1 микросекунда, а у случају грађевинске фирме месец дана. Исто тако, посао може бити копирање књиге, извршавање дела програма, или градња објекта. Реална ситуација је нешто идеализована тиме што се сматра да сви послови у датом примеру једнако трају, и да се обављају један по један без застоја.

Потребно је начинити оптималан избор послова који ће бити обављени на време. Прецизније, за сваку јединицу времена треба изабрати по један посао, тако да сваки изабрани посао буде обављен на време, а да збир зарада на изабраним пословима буде што већи.

Нека је дат један редослед неких послова, у коме се сви изабрани послови обављају на време. Исти послови се могу обавити и у редоследу у коме величине d_i расту. Заиста, нека је у полазном редоследу посао p био обављен у тренутку t_p, а посао q у тренутку t_q, где је $t_p < t_q$ и $d_p \geqslant d_q$. Другим речима, прво је обављен посао p, који може више (или исто) да чека. Оба посла су обављена на време, односно важи $t_p \leqslant d_p$, $t_q \leqslant d_q$. На основу ових релација следе два закључка. Прво, $t_p < tq \leqslant d_q$, тј. посао q се може обавити у тренутку t_p јер је и у евентуално каснијем тренутку t_q био обављен на време. Друго, $t_q \leqslant d_q \leqslant d_p$, што значи да се посао p може обавити у време t_q, јер је у то време био успешно обављен посао q, који може мање (или исто) да чека. Према томе, ако неки скуп послова може да се обави на време, онда редослед обављања увек може да се измени тако да се испоштује принцип „најпре најхитнији“. Користећи контра-

позицију претхдног закључка, долазимо до јасније формулације: ако неки скуп послова не може да се обави на време поштујући принцип „најпре најхитнији“, онда не може да се обави ни у једном редоследу.

Ради једноставности изражавања уведимо следеће дефиниције: скуп послова ћемо звати *изводљивим*, ако се сви послови из тог скупа могу обавити на време, у противном, скуп је *неизводљив*. За изводљив скуп послова кажемо да је *недопуњив*, ако придруживањем скупу било којег од преосталих послова скуп постаје неизводљив. Сада можемо формулисати тврђење: сви изводљиви недопуњиви скупови имају исти број елемената. Докажимо ово тврђење!

Нека су A и B два изводљива недопуњива скупа послова, и нека су послови $a_k \in A$, $k = 1, M$ и $b_k \in B$, $k = 1, N$, поређани по хитности. Претпоставимо да A има мање елемената од B, тј. $M < N$. Јасно је да за свако k важи $d(a_k) \geqslant k$. Нека је k најмањи број за који важи $(\forall j)\, k < j \leqslant M \implies d(a_j) > j$. То значи да је k-ти посао последњи који се не може одлагати, или ако се сви послови могу одлагати, тада је $k = 0$. Да не бисмо издвајали случај $k = 0$, можемо сматрати да постоји фиктивни посао a_0, који се обавља у тренутку 0 (и носи зараду 0). Послова $b_{k+1}, b_{k+2}, \ldots, b_N$, има више него $a_{k+1}, a_{k+2}, \ldots, a_M$, јер је $N > M$ (могуће је да ових других и нема, ако је $k = M$, али тај случај не захтева засебно разматрање). Због тога међу пословима $b_{k+1}, b_{k+2}, \ldots, b_N$, постоји бар један који није међу $a_{k+1}, a_{k+2}, \ldots, a_M$, нека је то b_X. Како је $d(a_k) = k < X \leqslant d(b_X)$, то је посао a_k хитнији од b_X, па b_X није ни међу првих k послова у A (они су још хитнији него a_k), дакле уопште није у скупу A. Уметнимо посао b_X у скуп A, тако да су послови и даље поређани по хитности. Тиме се у скупу A одлажу за једну јединицу времена само неки од послова након k-тог, а важи $(\forall j)\, k < j \leqslant M \implies d(a_j) > j$, па се овим уметањем неће нарушити изводљивост послова у A. Ово је у контрадикцији са претпоставком да је A недопуњив скуп, што доказује да не може бити $M < N$. На исти начин није могуће ни $M > N$, па је $M = N$.

Доказали смо да се при сваком изводљивом и недопуњивом избору послова обавезно обавља исти број послова, означимо га са S. То значи да је сваки изводљив скуп са мање од S послова увек допуњив. Због тога је могуће у план укључити најпре посао који доноси највећу зараду, затим први следећи посао по величини зараде, који се може обавити заједно са првим послом (не нарушава изводљивост), итд. Прецизније, потребно је извршити следеће кораке:

- ☐ Уредити послове тако да зараде при њиховом успешном обављању опадају.
- ☐ Уврстити у план ангажовања први посао.
- ☐ За сваки следећи посао X редом,
  - ☐ Привремено уметнути посао X у низ A изабраних послова, тако да у низу A изабраних послова крајњи рокови расту.
  - ☐ Ако у новом низу A послова сваки посао може да се обави на време, тј. $(\forall k)\, d_k \leqslant k$, привремено додати посао X остаје у списку изабраних, иначе се брише из списка.

Докажимо да наведени поступак даје оптималан избор послова, означен са A. Нека је неки други избор послова B оптималан. Поређајмо послове у скуповима A и B по величини профита у опадајући редослед. Нека је прво неслагање елемената низова на позицији i $(0 < i \leqslant r)$, тј. нека се првих $i-1$ послова у ова два скупа поклапа, и нека $a_i \in A$ и $b_i \in B$ представљају први пар различитих елемената ових скупова. Доказаћемо најпре да посао b_i није вреднији од a_i. Претпоставимо да је b_i вреднији од a_i. За $i = 1$ то је немогуће, јер је a_1 највреднији посао. Са друге стране, за $i > 1$ би b_i (према начину бирања посла a_i), ако је вреднији, чинио са првих $i-1$ послова скупа A неизводљив скуп послова. Како исти послови постоје и у скупу B, следило би да је B неизводљив скуп. Дакле, b_i није вреднији од a_i. Додајмо сада посао a_i у скуп B. Ако скуп B постаје неизводљив, биће потребно уклонити један посао из скупа B, али то не мора бити ниједан од првих $i-1$ послова, јер је a_i са њима сагласан. Одавде следи да уклоњени посао нема већу вредност него a_i, па се овом заменом укупан профит не смањује. Након ове замене, уочимо нови индекс i са истом особином, и поновимо поступак. Долазимо до закључка да скуп послова A даје профит већи или једнак као у B, одакле следи да је и скуп A оптималан.

Математичко разматрање које изнето у овом задатку, даје један ефикасан начин за решавање класе проблема, који је познат као похлепан алгоритам (greedy algorithm). Основна идеја је јасна: у сваком тренутку чинити избор који је локално оптималан, односно који тренутно највише увећава величину која се максимизира (најмање увећава, ако се ради о минимизацији), а не нарушава услове које треба да испуни начињени избор. Похлепни алгоритми су врло ефикасни, и применљиви у многим задацима, али у многим задацима и нису. Препознавање и доказивање испуњености услова потребних за гаранцију да похлепним алгоритмом заиста добијамо решење, претпоставља извесно знање математике. Како је разматрање слично у многим проблемима, створена је чврста теоријска основа са потребним алгебарским структурама, која знатно олакшава потребну анализу.

Неко ко математику не познаје довољно за овакав начин решавања задатака, можда би задатак решавао чак и бектреком, што је један од најспоријих начина да се овакав задатак реши.

Остали примери

Следи још неколико задатака једноставне формулације, који се могу једноставно и решити, али није одмах јасно да је такво решење исправно. Стога је већа пажња посвећена начину размишљања и доказу коректности понуђених решења. Решења задатака „срећа“ и „дисјунктне дужи“ могу бити неефикасна у специјално конструисаним примерима, али се њихова ефикасност може побољшати пажљивијим избором почетне поделе, као и избором итеративног корака.

Оптимално спаривање бројева

Дат је низ од $2N$ елемената. Потребно је формирати N парова, тако да највећи збир пара елемената буде што мањи. На пример, за бројеве

$2, 8, 5, 3, 1, 4$ *једно оптимално спаривање је* $(2, 3)$, $(8, 1)$, $(5, 4)$, *јер* $\max(2+3, 8+1, 5+4) = 9$, *а свако друго спаривање даје исти или већи збир.*

Размотримо једноставан случај, када имамо само 4 броја. Нека су то (поређани у растући редослед) a_1, a_2, a_3 и a_4. Имамо три могућа спаривања:

$$(a_1, a_2),\ (a_3, a_4);\ X = \max(a_1 + a_2, a_3 + a_4) = a_3 + a_4, \quad \text{јер } a_1 \leqslant a_3, a_2 \leqslant a_4,$$
$$(a_1, a_3),\ (a_2, a_4);\ Y = \max(a_1 + a_3, a_2 + a_4) = a_2 + a_4, \quad \text{јер } a_1 \leqslant a_2, a_3 \leqslant a_4,$$
$$(a_1, a_4),\ (a_2, a_3);\ Z = \max(a_1 + a_4, a_2 + a_3).$$

Како је $a_1 + a_4 \leqslant a_3 + a_4 = X$, $a_2 + a_3 \leqslant a_4 + a_3 = X$, следи $Z \leqslant X$. Слично, важи: $a_1 + a_4 \leqslant a_2 + a_4 = Y$, $a_2 + a_3 \leqslant a_2 + a_4 = Y$, одакле следи $Z \leqslant Y$.

Закључујемо да је треће спаривање оптимално. Више од тога, ако у задатку са већим бројем парова бројева издвојимо два пара, највећи збир се неће повећати (а може се смањити) када бројеве из ова два пара преспаримо тако да су у једном пару први и четврти, а у другом пару други и трећи по величини.

Пређимо сада на општи случај. Посматрајмо пар бројева у коме се налази најмани број и пар у коме се налази највећи број. Ако то није исти пар, онда извођењем описане размене, можемо добити ново спаривање које није лошије од полазног. Изоставимо затим из разматрања најмањи и највећи број, па поновимо поступак. Долазимо до закључка да се оптимално спаривање може добити на следећи једноставан начин: сортирамо бројеве у растући поредак, а затим спаримо први са последњим, други са претпоследњим, ... N-ти са $N+1$-вим.

Упоредимо ово елегантно решење са дуготрајним испробавањем разних могућих спаривања бројева, па ће бити јасније колико је у програмирању важна примена математике.

Дисјунктне дужи

Дата су два низа, A и B, од по N тачака у равни. Спарити сваку тачку из низа A са тачно једном тачком из низа B, тако да се добије N дисјунктних дужи.

Нека је дато једно бијективно спаривање тачака и тиме N дужи. Уколико постоји пар дужи које се секу, можемо тим дужима разменити B крајеве, тако да се после ове размене добија пар дисјунктних дужи. Назовимо овај поступак раскрштањем дужи. Приметимо да се новодобијене дужи могу сећи са неким од преосталих дужи, са којима се две полазне дужи нису секле. Због тога се поставља питање, да ли се раскрштањем појединих парова уопште може доћи до скупа међусобно дисјунктних дужи, односно да ли би одговарајући итеративни поступак био коначан. Могу се поставити и друга питања, на пр. да ли је потребно раскрштати дужи које испуњавају неки додатни услов (на пр. раскрштати дужи у неком посебном редоследу) да би поступак био коначан, и сл. али када дамо (можда неочекивано једноставан) одговор на прво постављено питање, многа од ових додатних питања постају сувишна.

Две дужи које се секу можемо схватити као дијагонале конвексног четвороугла. Раскрштањем таквих дужи, пар дијагонала замењујемо паром наспрамних

страница, које (према неједнакости троугла) имају мањи збир дужина него дијагонале. Спаривања тачака има коначно много (тачније $(2N)!/N!$), а могућих збирова дужина тако добијених N дужи може бити највише исто толико, па и тих збирова има коначно много. Како се сваким раскрштањем дужи укупан збир дужина смањује, он је различит од свих претходних, па због тога поступак не може трајати неограничено. То значи да примењујући раскрштање у било ком редоследу, морамо доћи до спаривања у коме су све дужи дисјунктне.

Срећа

У друштву од N познаника неки се симпатишу, а неки не. Степен симпатије између две особе задаје се целим бројем, различитим од 0 (што је тај број већи, особе се више симпатишу, и обрнуто). Потребно је поделити ових N људи у две групе. При томе се срећа сваке особе израчунава као збир степена симпатија са људима из исте групе минус збир степена симпатија са људима из супротне групе. Распоредити све особе у две групе, тако да ниједна особа не буде несрећна, тј. да срећа сваке особе буде ненегативан цео број.

Придружимо свакој подели људи на две групе следећу матрицу A: ако су особа i и особа j у истој групи, елементу $A(i,j)$ додељујемо вредност једнаку степену симпатије између i-те и j-те особе, а ако су у разним групама, додељујемо му супротну вредност. Тада је збир i-те колоне (или i-те врсте, јер је матрица A симетрична) једнак срећи i-те особе. Приметимо и то, да преласком i-те особе у супротну групу, сви бројеви i-те врсте и i-те колоне мењају знак. Закључујемо да пребацивањем „несрећне особе“ у супротну групу, та особа постаје „срећна“. Наравно, тада ће можда неке друге особе постати несрећне (збир неких других колона може постати негативан). Можемо ли доказати да поступак заснован на овој идеји неће трајати бесконачно дуго?

Када нађемо колону са негативним збиром (несрећну особу), па свим бројевима те колоне и одговарајуће врсте променимо знак, збир свих елемената матрице постаје већи, и тиме различит од свих претходних. Како подела у групе има коначно много (тачније 2^{N-1}), то и збирова свих елемената матрице има коначно много (највише исто толико). С обзиром да се после сваког „усрећавања“ неке особе добија нови збир, следи да је поступак коначан. Према томе, пре или касније морамо доћи до матрице без негативног збира колоне, а тиме и до поделе без несрећних особа. Могуће је да овако добијена подела није оптимална, тј. да је могуће стање веће „опште среће“, међутим такво стање се није ни тражило у задатку.

Најмањи број који није збир осталих

Дат је низ A од N природних бројева. Наћи најмањи природан број који се не може представити као збир неких елемената низа. Сваки елемент низа може учествовати највише један пут као сабирак у једном збиру.

Претпоставимо ради анализе, да су елементи низа сортирани у растући (тачније неопадајући) поредак. Означимо збир првих k елемената низа са S_k,

$k = 0, N$ ($S_0 = 0$). Претпоставимо даље да је за неко k, сваки елемент низа закључно са k-тим највише за 1 већи од збира свих претходних елемената низа, а да је $k+1$-ви елемент низа за више од један већи него збир S_k првих k елемената. Докажимо да се сви бројеви од 1 до S_k могу добити сабирањем неких од првих k елемената низа, и за то је довољно првих j елемената ако је дати број мањи или једнак S_j. У сврху доказивања претпоставимо супротно, да се неки природан број мањи или једнак S_j не може добити сабирањем неких од првих j елемената низа, $0 < j \leqslant k$. Нека је X најмањи такав број. Тада јасно $X \neq S_p$ за све p, јер би иначе X био добијен сабирањем првих p елемената низа. Нека је $S_{j-1} < X < S_j$ (у противном изаберимо мање j). Како је $j \leqslant k$, из дефиниције броја k следи $S_{j-1} + 1 \leqslant a_j$, и зато $a_j \leqslant X < S_j$. Одузимањем a_j од сва три израза из ове двоструке неједнакости, добијамо $0 \leqslant X - a_j < S_j - a_j = S_{j-1}$. Како је $X - a_j < X$, а X је најмањи број који се не може добити, $X - a_j$ се може добити помоћу неких од првих $j-1$ елемената низа (можда ниједног), а тада је X могуће добити додавањем a_j на тај збир! Добијена контрадикција доказује тврђење.

Приметимо још да се за $S_k + 1 < a_{k+1}$ број $X = S_k + 1$ не може добити сабирањем елемената низа. Заиста, елементи који су већи од X не могу учествовати у збиру, а елементи који су мањи или једнаки X нису довољни ($S_k < X$).

Из наведеног разматрања следи закључак: тражени број X једнак је најмањем $S_k + 1$ које је мање од a_{k+1}, $0 \leqslant k < N$, односно једнак је $S_N + 1$, ако таквог $S_k + 1$ нема.

Тако долазимо до једноставног алгоритма (претпоставка је да су елементи поређани по величини, у противном би било потребно користити низ, да би се елементи сортирали у растући поредак).

```
var
  M, i, n, prvine:  longint;
  nasao:boolean;
begin
  write('n= '); readln(n);
  i:=1; prvine:=1; nasao:=false;
  while (i<=n) and not nasao do
  begin
    write('M[', i:2, ']= ');
    readln(M);
    if(M>prvine) then nasao:=true
    else begin prvine:=prvine+M; i:=i+1 end;
  end;
  writeln('Trazeni broj je ', prvine);
end.
```