Проблем 1. Фреквентна сума

Дат је низ бројева $A = (a_1, \ldots, a_N)$. Посматрајмо скуп свих сума узасотпних чланова

$$S = \{a_i + ... + a_j | 1 \leq i \leq j \leq N\}$$

Исписати вредност из скупа S која се најчешће појављује, као и колико пута се појављује. У случају да има више таквих, исписати ону чија је вредност највећа.

Улаз. (Улазни подаци се учитавају са стандардног улаза.)У првом реду стандардног улаза налази се природан број N ($1 \leq N \leq 3000$). У следећем реду налази се N природних бројева, редом $a_1, \ldots, a_N$, сваки из интервала $[0, 3000]$.

Излаз. (Излазни подаци се исписују на стандардни излаз.)У први и једини ред стандардог излаза исписати два природна броја, који редом представљају, број који се најчешће појављује у скупу S, и колико пута се појављује. У случају да постоји више таквих бројева, исписати онај који има највећу вредност.

Ограничења. У 30% тест примера ће бити $1 \leq N \leq 100$.

Пример 1.

standardni ulaz	standardni izlaz
3 1 2 3	3 2

Објашњење. Скуп $S = \{1, 1+2, 1+2+3, 2, 2+3, 3\} = \{1, 3, 6, 2, 5, 3\}$.

Пример 2.

standardni ulaz	standardni izlaz
8 17 13 17 13 5 6 5 6	30 3

Објашњење. Приметимо да се и сума 11 појављује 3 пута, али је 30 већа сума.

Решење. Проблем се може решити коришћењем *counting* – *sort*-а за сортирање елемената скупа S.

Brojanje svake sume

Једноставно решење за овај задатак се може дати следећим псеудо-кодом:

```
res := 0
for i := 1 to n do
  for j := i to n do
    sum := racunajSumu(i, j)
    cnt := 0
    for idx1 := 1 to n do
      for idx2 := idx1 to n do
        if sum == racunajSumu(idx1, idx2) then
          cnt++;
    if cnt > res then
      res := cnt

return res
```

За сваку суму која се може јавити у скупу S, алгоритам броји колико пута се та сума појављује у целом низу, у скуп S. Претпоставимо да функција $racunajSumu(i, j)$ ради у времену $O(j - i)$, односно редом сабира елементе. У том случају, временска сложеност овог приступа је $O(n^5)$.

Rachunanje sume u konstantnom vremenu

Позив функције $racunajSumu(i, j)$ рачуна следеће $sum = a_i + \ldots + a_j$. Вредност sum може да се запише на другачији начин, као

$$sum = a_i + \ldots + a_j = (a_1 + \ldots + a_j) - (a_1 + \ldots + a_{i-1}) = A_j - A_{i-1},$$

где $A_k = a_1 + \ldots + a_k$. Дакле, ако имамо вредности A_k, $k = 1 \ldots n$, тада се позив $racunajSumu(i, j)$ може израчунати у константном времену. Приметимо да $A_i = A_{i-1} + a_i$. Из техничких разлога ћемо поставити $A_0 = 0$. Користећи ове једнакости, цео низ A_k, $k = 1 \ldots n$, можемо израчунати у $O(n)$. Тада сложеност алгоритма из претходног одељка постаје $O(n^4)$.

Grupisanje istih vrednost

Две унутрашње петље приказане у првом одељку, заправо генеришу скуп S и проверавају колико пута се одређени елеменат налази у њему. Скуп S смо могли генерисати само једном, на почетку програма, а потом користити одговарајућу структуру података да проверимо колико пута се одређени елеменат појављује у том скупу. Тиме бисмо добили бржи алгоритам, али и даље недовољно ефикасан. Да бисмо добили ефикасанији алгоритам, треба да приметимо следеће – ако нам је дат низ од n елемената, и желимо да проверимо који елеменат у датом низу се најчешће појављује, довољно је да сортирамо цео низ и бројимо дужине блокова узастопних једнаких елемената. Следећи псеудо-коде илуструје ову идеју:

```
B = niz iz ulaza
sort(B)
res := 0
idx := 1
while idx <= n
  j := idx
  cnt := 0
  while j <= n and B[idx] == B[j] do
    j++
    cnt++
  if cnt > res then
    res := cnt
  idx := j

return res
```

Дакле, да бисмо добили знатно ефикасније решење довољно је на почетку програма генерисати скуп S, сортирати елементе који се налазе у њему и применити управо описани алгоритам. Скуп S има $O(n^2)$ елемената. Користећи одговарајући алгоритам за сортирање, можемо сортирати све елементе скупа у времену $O(n^2 \log n)$ користећи, на пример $quick - sort$ или $merge - sort$. Описан алгоритам ради у времену $O(n^2)$. То даје временску сложеност $O(n^2 \log n + n^2) = O(n^2 \log n)$.

Counting-sort

Приступ који смо описали у прошлом одељку за $n = 3000$ направи око 100,000,000 операција. Ово ће уз додатну оптимизацију можда проћи на свим тест примерима. Волели бисмо да

имамо алгоритам који ће сигурно радити за све тест примере у времену. Идеје које смо описали у претходном одељку су веома згодне и ефикасне у општем случају. У овом случају су улазне вредности низа релативно мале. У таквим ситуацијама *counting* − *sort* може да буде ефикаснији од *quick* − *sort*-а или *merge* − *sort*-а. Управо је то случај у овом проблему. *Counting* − *sort* ради у времену разлике највеће и најмање вредности које можемо имати у низу, што је у овом проблему 9,000,000 и 0. С друге стране, видели смо да *quick* − *sort* ради у времену $O(n^2 \log n)$. У овом проблему, максимална вредност израза $n^2 \log n \approx 100,000,000$. Из приложеног је јасно да је *counting* − *sort* ефикаснији. Ако из прошлог одељка за сорт алгоритам одаберемо *counting*−*sort*, добијамо временску сложеност решења $O(MAX\_V \cdot n + n^2)$. Овакво решење је довољно ефикасно да у времену реши све тест примере.

Тестирање. Тестирање решења се вршило над корпусом од 25 тест примера. Вредност сваког примера је 4 поена. Тест примери су у већини случајева генерисаи користећи генератор случајних бројева уз одређена ограничења. Генерисани су тест примери у којима је оптимално решење сума 0. Потом су генерисана решења у којима постоји више сума са највећим бројем појављивања.

Аутор:
Слободан Митровић