Řešení **2mis** (cvičení 5. 10. 2017)

Vilém Zouhar

Naivní řešení:

Nejjednodušší řešení je udržovat si bool pole, inicializované zpočátku na **0**. Stačil by tedy jeden průchod, ve kterém by čísla přicházející ze vstupu byla indexy pole a nastavovala by tyto prvky na **1**.

Kód (funkční python3, ale slouží jako pseudokód):

```
N = int(input())
present = [0]*N

for i in range(N-2):
    present[int(input())-1] = 1

for i in range(N):
    if not present[i]:
        print(i+1)
```

Toto několikařádkové řešení předpokládá znalost celkového počtu prvků. Pokud by to nebyla pravda, dal by se vstup ukládat do pomocného pole a z něj zjistit maximální prvek (pokud by chyběl právě ten maximální, pak by pole `present` bylo až na jednu nulu plné jedniček). Časová náročnost je **O(n)**, neboť pro každé číslo děláme řádově konstantní počet operací. Paměťová složitost je také **O(n)**, neboť potřebujeme úměrně velké pole. Lepší verze algoritmu musí tedy být alespoň v nějakém ohledu lepší. Časová složitost se zlepšit nedá, neboť musíme minimálně přečíst všechny prvky pole, je tedy třeba zlepšit paměťovou náročnost.

Lepší naivní řešení:

Pokud bychom se snažili optimalizovat paměťovou náročnost i na úkor té časové, mohli bychom si pole seřadit. K tomu bychom mohli využít například quick sort s in-place sjednocováním. Pak by stačil jeden průchod, kde bychom v nově seřazeném poli hledali místa, kde jsou prvky přeskočené. Časová složitost takového postupu by byla **O(n log(n))**, paměťová pak **O(log(n))** (předpokládejme, že vstupní pole není součástí paměťové náročnosti). Intuitivně ale rozumíme, že toto řešení je krkolomné, zbytečně náročné na implementaci a nenabízí mnoho výhod oproti naivnímu řešení.

Řešení s průměrem:

Nejprve si definujeme operaci na typu číselné pole. Operace `sum_le(pole, hranice)` vrací součet prvků v `pole` menší nebo roven `hranice`. Taková operace je časově **O(n)**, byť je problém, že musíme uchovávat součet řádově r^2, kde `r` je rozsah proměnné `n`. Tento problém nastává, i kdybychom nepoužívali vzorec pro součet (`n(n+1)/2`), ale postupně bychom přičítali čísla `N, N-1, N-2, N-3, …, 1` a od nich odčítali čísla na vstupu. Mohlo by se totiž stát, že na vstupu budou čísla `1, 2, 3, ... , N`. V polovině těchto částečných součtů by byl součet $n^2/4$.

Zde nemůžeme použít trik s `XOR`, neboť ten nám nedává jednoznačně sumu dvou čísel. Pro jednoduchost budeme používat vzorec pro součet.

První tedy vypočítáme `sum_le(pole, N)`, čímž získáme součet chybějících čísel (`a, b`), ze kterého dělením dvěmi získejme celou část průměr čísel `a, b`. BÚNO předpokládáme, že `a < b` (rovné si být nemohou). Součet `a` (menšího z hledaných) a všech prvků v poli menších než průměr a, b (`avg_ab`) je součet všech přirozených čísel menších než `avg_ab` (a "doplňuje chybějící místo"). Z toho vyjádříme a jako rozdíl `sum_le(pole, avg_ab) - avg_ab(avg_ab+1)/2`. `b` získáme triviálně jako `sum_ab - a`.

Kód (funkční python3, ale slouží jako pseudokód):

```
def sum_le(pole, hranice):
    suma = 0
    for num in pole:
        if num <= hranice:
            suma += num
    return suma

N = int(input())
arr = [0]*N

for i in range(N-2):
    arr[i] = int(input())

sum_ab = N*(N+1)/2 - sum_le(arr, N)
avg_ab = int(sum_ab/2)

a = int(avg_ab*(avg_ab+1)/2 - sum_le(arr, avg_ab))
b = int(sum_ab - a)

print(a)
print(b)
```

Časová náročnost je stále **O(n)**, avšak paměťová je konstantní, pokud pomineme problém s kapacitou proměnné pro sumu.

Pokračování:

Pro **1*^{m}** (m+1 chybějících čísel z posloupnosti) je řešení alespoň časově i paměťově **O(n)**, triviální úpravou z naivního řešení. Jakýkoliv jiný algoritmus musí tedy být paměťově lepší.