Řešení ukázkové úlohy
Bankovní pomocník

Řešení

Zformulujme si nejprve zadání úlohy bez všech nepodstatných věcí okolo. Na vstupu je N-prvková neklesající posloupnost čísel $S_1, \ldots, S_N$. Úkolem je najít v ní dva různé prvky, jejichž součet dává přesně K.

První řešení, které nás nejspíše napadne, bude asi tohle: Načti si vstupní posloupnost do pole S. Poté vyzkoušej všechny dvojice různých indexů pole, zda-li se na nich neskrývá hledaný součet K. Zapsáno pseudokódem asi nějak takto:

```
read N, K
for i=1 to N
     read S[i]

for i=1 to N
     for j=1 to N
         if i!=j and S[i]+S[j] = K then
              print "Reseni je " i " a " j "."
              exit
print "Reseni neexistuje."
```

Pojďme se zamyslet, jak takové řešení bude efektivní. Abychom dokázali předpovědět, jak se naše řešení bude chovat pro velké hodnoty N (kde nás to zajímá především), spočítáme, kolik se zhruba v algoritmu vykoná operací v závislosti na hodnotě N. Zde načtení vstupu trvá N kroků, nicméně pak následují dva do sebe vnořené for-cykly, z nichž každý se provádí N-krát a uvnitř testuje podmínku. To ale potrvá v nejhorším případě (to když řešení neexistuje) N^2 kroků. Tento algoritmus tedy provede řádově kvadraticky mnoho kroků vzhledem k N, čemuž říkáme, že má kvadratickou časovou složitost a značíme to zkratkou $O(N^2)$. Sami si vyzkoušejte, že pro velké hodnoty N (třeba $N = 100000$) už takový program bude pracovat velmi dlouho.

Zkusme nyní vymyslet řešení rychlejší. Všimněme si, že jsme dosud nijak nezužitkovali, že vstupní posloupnost je vzestupně setříděná. Náš druhý algoritmus bude fungovat takto: Zavedeme si dva indexy i a j, které budou ukazovat do pole S. Na začátku nastavíme $i = 1$ a $j = N$. Podíváme se, jestli náhodou prvky S_i a S_j nedávají součet K. Pokud ano, vyhráli jsme a výsledek vypíšeme. Pokud ne, nastávají dvě možnosti: buďto $S_i + S_j > K$, v tomto případě je aktuální součet příliš vysoký a posuneme tedy j o jedno místo doleva, anebo $S_i + S_j < K$, kdy je aktuální součet příliš nízký a posuneme tedy i o jedno místo doprava. Tyto kroky opakujeme tak dlouho, dokud buďto nenajdeme řešení, anebo dokud se indexy i a j nepotkají, což znamená, že řešení neexistuje.

Zanalyzujme nyní, jak rychle náš algoritmus poběží. Všimněme si, že v každém kroku (pokud tedy algoritmus neohlásil řešení a neskončil) se k sobě indexy i a j přiblíží o jedničku. Protože počáteční vzdálenost i a j byla N, znamená to, že se vykoná nejvýše N kroků, z nichž každý sestává z dvou testů podmínek a případné změny jednoho indexu. Algoritmus tedy vykoná řádově lineární množství operací vzhledem k N, neboli jeho časová složitost je $O(N)$. To je pochopitelně rychlejší řešení, než náš první pokus. Co se týká paměťové spotřeby, algoritmus vyžaduje pro svůj běh cca 3 pomocné proměnné a hlavně N paměťových buněk pro uložení prvků pole S. Jeho

paměťová složitost je tedy $O(N)$. (V programu samotném je sice pole fixně velké, ale asi není problém si představit, že pole bychom mohli alokovat na přesně N prvků až za běhu.)

Zbývá však nejdůležitější otázka, kterou jsme si nechali až na závěr: proč náš algoritmus vůbec funguje? Nemůže se třeba stát, že by nějakou dvojici tvořící řešení přeskočil? Nejprve si všimněme, že pokud v poli S správně řešení neexistuje, náš algoritmus ani žádné jiné vypsat nemůže (to se totiž stane jen tehdy, když se indexy i, j ocitnou na prvcích se součtem K).

Dobrá, a nemůže tedy nějaké řešení vynechat? Dejme tomu, že řešení se skrývá na indexech ℓ a r, $\ell < r$. Představme si, jak algoritmus běží. Uvažme první okamžik, kdy se buďto index i nastaví na ℓ nebo index j nastaví na r.

Dejme tomu, bez újmy na obecnosti, že nastala první možnost, neboli $i = \ell$ a tedy $j > r$. Dvojice ℓ, r by byla algoritmem přeskočena, pokud by nyní index i opustil hodnotu ℓ dříve, než index j dosáhne hodnoty r. A teď učiníme důležitý postřeh, ve kterém využijeme setříděnosti posloupnosti S: to se přeci nestane! V tomto okamžiku je totiž součet pod indexy i, j příliš velký, náš algoritmus jsme však navrhli tak, že v takovém případě posune j doleva, nikoli i doprava, a to musí dělat tak dlouho, dokud j nedokráčí na r.

Pokud nastala druhá možnost, tedy jako první bylo $j = r$, všimněme si, že projde analogický argument jako v předchozím odstavci. Můžeme tedy tuto diskuzi uzavřít s tím, že jsme dokázali, že náš algoritmus funguje správně.

K řešení připojujeme algoritmus naprogramovaný v jazyce C. Program samotný je přesným přepisem výše uvedeného algoritmu.