Opracowanie: MON
Monety

1 Analiza problemu

Przypomnijmy, że ciąg a_i ($1 \leqslant i \leqslant n$) oznacza rozmiary przegród na monety. Jako, że monety Bajtazara mają wielkości dokładnie $1, 2, \ldots, n$, zadanie sprowadza się do obliczenia wszystkich kolejności liczb $1, 2, \ldots, n$, takich, że i-ta z kolei liczba jest mniejsza lub równa a_i.

2 Rozwiązanie brutalne

Najprostszym rozwiązaniem jest wygenerowanie wszystkich kolejności liczb $1, 2, \ldots, n$ (czyli permutacji) i dla każdej z nich sprawdzenie powyższego warunku. Niestety liczba permutacji zbioru złożonego z n elementów wynosi $n!$, czyli $1 \cdot 2 \cdot \ldots (n-1) \cdot n$, co nawet dla niewielkich n może być ogromną liczbą ($10! = 3628800$, $15! = 1307674368000$). Można co prawda przerywać obliczenia, jeśli zauważymy, że któraś nierówność nie jest spełniona i nie generować więcej permutacji, ale nawet takie rozwiązanie nie ma szans zmieścić się w limitach czasowych.

3 Rozwiązanie wzorcowe

Jak widać musimy być sprytniejsi. Najpierw zauważmy, że wynik nie zależy od kolejności liczb a_i. Dlaczego tak jest? Jak już powiedzieliśmy kolejność liczb $1, 2, \ldots, n$ opisuje poprawne zapełnienie klasera wtedy i tylko wtedy gdy i-ta z kolei liczba jest mniejsza lub równa a_i. Wyobraźmy sobie sytuację, że zamieniamy między sobą pewne przegródki, ale wraz z nimi przesuwamy monety, które próbowaliśmy do nich włożyć. Wtedy poprawne przyporządkowanie oczywiście się nie popsuje, a niepoprawne nadal pozostanie niepoprawne.

Skoro nie musimy dbać o kolejność liczb a_i, to możemy przeglądać je w najwygodniejszej kolejności czyli niemalejącej. Do najmniejszej przegródki możemy włożyć oczywiście a_1 najmniejszych monet. Do drugiej przegródki zmieści się a_2 monet, ale pamiętajmy, że jedną już zużyliśmy, zatem pierwsze 2 przegródki zapełnimy na $a_1(a_2 - 1)$ sposobów. Może dalej będzie podobnie?

Załóżmy, że $k-1$ pierwszych przegródek jest już zapełnione. Każdą niewykorzystaną monetę mniejszą lub równą a_k możemy włożyć na k-te miejsce, a takich monet jest $a_k - k + 1$. Dzięki temu otrzymujemy elegancki wzór na wynik: $a_1(a_2 - 1)(a_3 - 2) \cdots (a_n - n + 1)$. Zauważmy, że klasera nie da się zapełnić tylko wtedy, gdy $a_k < k$ dla pewnego k. Zgadza się to z naszym wzorem, bo kolejne czynniki mogą zmniejszać się maksymalnie o 1 i $a_k < k$ pociąga pojawienie się 0 w iloczynie.

Zanim zaczniemy pisać rozwiązanie zwróćmy uwagę na haczyk — jeśli po prostu wymnożymy powyższe n liczb, możemy wyjść poza zakres liczb 64-bitowych! Aby tego uniknąć możemy po każdym mnożeniu pamiętać tylko resztę z dzielenia wyniku przez $10^9 + 7$. Oto pseudokod powyższego rozwiązania (a `modulo` b oznacza resztę z dzielenia liczby a przez b).

```
sort(a[1]...a[n]);
wynik = 1;
dzielnik = 1 000 000 007;

for i := 1 to n do
    wynik = (wynik * (a[i] - i + 1)) modulo dzielnik;

return wynik;
```

Przedstawiony algorytm działa w czasie $O(n)$ czyli liczba operacji jest proporcjonalna do n — o wiele szybciej niż pierwszy pomysł. Nie wolno zapomnieć o czasie sortowania liczb a_i — tu możemy użyć sortowania przez zliczanie, które również działa w czasie $O(n)$. Takie rozwiązanie można znaleźć w plikach `mon.cpp` oraz `mon.pas`. Jeśli użyjemy sortowania w czasie $O(n \log n)$, otrzymamy rozwiązanie niewiele wolniejsze.

4 Rozwiązanie alternatywne

Drugie rozwiązanie jest nieco trudniejsze do zrozumienia, ale również jest ciekawe. Polega na odwróceniu sposobu myślenia. Otóż zamiast zastanawiać się, ile monet możemy włożyć do jednego pojemnika, możemy zliczać, do ilu pojemników może trafić dana moneta. Oznaczmy przez $F(k)$ liczbę sposobów włożenia k najmniejszych monet do k najmniejszych przegródek. Wynikiem działania programu będzie oczywiście $F(n)$. Aby jednak otrzymać algorytm, musimy znaleźć sposób na obliczanie kolejnych wartości $F(k)$.

Najpierw posortujmy przegródki niemalejąco, podobnie jak w poprzednim rozwiązaniu. Początek obliczania funkcji F jest prosty: $F(1) = 1$, ponieważ $a_1 \geqslant 1$. Załóżmy, że znamy $F(k-1)$. Jak wtedy policzyć $F(k)$? Włożyliśmy $k-1$ monet i musimy znaleźć miejsce na monetę wielkości k. Możliwych miejsc jest tyle, ile przegródek wielkości co najmniej k wśród pierwszych (czyli najmniejszych) k przegródek — oznaczmy tę liczbę przez b_k. Jeśli $b_k = 0$, to widać, że się nam nie powiedzie i możemy przerwać obliczenia. Załóżmy więc, że $b_k > 0$. A co zrobić z monetą, która była wcześniej na wybranym miejscu? Ma ona rozmiar mniejszy od k, więc możemy wstawić ją w k-tą przegródkę (tutaj korzystamy z tego, że posortowaliśmy dane!).

Zatem z każdej poprawnej permutacji pierwszych $k-1$ monet otrzymujemy b_k poprawnych permutacji pierwszych k monet. Trzeba tutaj pokazać, że w ten sposób nie stworzymy dwukrotnie tej samej permutacji, ale zostawiamy to jakie łatwe ćwiczenie dla czytelnika. Daje nam to wzór $F(k) = b_k F(k-1)$ dla $k > 1$.

Zatem wynik to po prostu $b_2 \cdot b_3 \cdots b_n$ — zgadza się to z obserwacją, że jeśli któreś b_i jest równe 0 to nie uda nam się w żaden sposób wypełnić klasera.

Pozostaje kwestia wyznaczenia liczb b_i. Niech c_i oznacza pierwszą pozycję, na której $a_{c_i} \geqslant i$ (pamiętajmy, że ciąg a_i jest posortowany niemalejąco). Zauważmy, że $b_i = i - c_i + 1$. Liczby c_i można już policzyć w bardzo prosty sposób:

```
j := 1;
for i := 1 to n do
begin
    while j <= a[i] do
    begin
        c[j] := i;
        j := j + 1;
    end;
end;
```

W przypadku, kiedy liczby `a[i]` są przechowywane w kubełkach, to obliczanie wartości `c[]` jest jeszcze prostsze. To rozwiązanie również działa w czasie $O(n)$. Zaimplementowano je w plikach `mon2.cpp` oraz `mon2.pas`.