Omówienie zadań z turnieju indywidualnego nr 30

Krzysztof Boryczka

8 marca 2025

Omówienie zawiera zadania (mniej więcej) w kolejności trudności. Niektóre z omówień zawierają także **Bonus**, do którego przeczytania szczególnie zachęcamy.

1. Trzy karty

W tym zadaniu mamy tylko 6 różnych możliwych wejść, zatem dla każdego z nich możemy ręcznie wyznaczyć odpowiedź.

Dostajemy rozwiązanie działające w czasie $\mathcal{O}(1)$.

Bonus: Warto zastanowić się, co gdyby na wejściu mogło być N liczb ($N \leq 500\,000$), a nie tylko 3.

Przydatne okazuje się być pojęcie *inwersji*, czyli takich par (niekoniecznie sąsiednich) elementów, że większy z nich znajduje się przed mniejszym.

Zauważmy, że jednym ruchem możemy liczbę inwersji tylko zmniejszyć lub zwiększyć o dokładnie 1. Ponadto, jeśli ciąg nie jest posortowany, to zawsze istnieje para sąsiednich elementów tworząca inwersję. Zatem minimalna liczba operacji to tak naprawdę liczba inwersji ciągu. Tak właśnie działają niektóre proste algorytmy sortujące, np. sortowanie przez wstawianie (Insertion Sort) lub sortowanie bąbelkowe (Bubble Sort).

Dzięki tej obserwacji możemy nieco uprościć nasze rozwiązanie, gdyż liczba inwersji ciągu (a, b, c) to po prostu $(a > b) + (a > c) + (b > c)$.

Liczenie liczby inwersji dla dowolnego ciągu jest standardowym problemem, który można zrealizować efektywnie, wykorzystując algorytm sortowania przez scalanie (Merge Sort) lub drzewo przedziałowe, w czasie $\mathcal{O}(N \log N)$.

2. Arbuz

Jeżeli każda z trzech części ma ważyć parzystą liczbę kilogramów, to również ich suma powinna być liczbą parzystą, zatem w musi być parzyste. Ponadto, skoro waga każdego z kawałków ma być dodatnia, to musi ona wynosić co najmniej 2, zatem $w \geq 6$.

Okazuje się, że dwa powyższe warunki są wystarczające, gdyż jednym z poprawnych podziałów jest $(2, 2, w - 4)$.

Dostajemy rozwiązanie działające w czasie $\mathcal{O}(1)$.

3. Dwa ułamki

Zauważmy, że jeśli założymy $x \leq y$, to $N < x \leq 2N \leq y$ (bo średnia arytmetyczna liczb $\frac{1}{x}$ i $\frac{1}{y}$ musi wynosić $\frac{1}{2N}$).

Zatem najpierw policzymy takie pary, że $x \leq y$, a następnie pomnożymy wynik przez 2 (aby dodać pary $x \geq y$) oraz odejmiemy 1 (bo dwukrotnie policzyliśmy parę $x = y$, która zawsze daje tylko jedno rozwiązanie $x = y = 2N$).

W tym celu wyznaczymy z równania y:

$$\begin{aligned} \frac{1}{x} + \frac{1}{y} &= \frac{1}{N} \\ \frac{1}{y} &= \frac{x-N}{xN} \\ y &= \frac{xN}{x-N} \end{aligned}$$

Zatem wystarczy, że przeiterujemy się wartością x po wszystkich liczbach od $N+1$ do $2N$ i sprawdzimy, czy liczba $\frac{xN}{x-N}$ jest liczbą całkowitą, to znaczy czy xN dzieli się przez $x - N$.

Dostajemy rozwiązanie działające w czasie $\mathcal{O}(N)$.

4. Dwa rejestry 2

Zauważmy, że każda z operacji działa podobnie jak w algorytmie Euklidesa (zwłaszcza te z minusem). W szczególności wykonanie dowolnej operacji zachowuje NWD rejestrów (największy wspólny dzielnik), zatem jeśli NWD obu par są różne, to wiemy że odpowiedź to `NIE`.

Okazuje się, że w przeciwnym wypadku odpowiedź to `TAK`.

Dla uproszczenia załóżmy chwilowo, że obie liczby są dodatnie. Korzystając z operacji `X-` oraz `Y-` możemy zasymulować algorytm Euklidesa i z początkowego stanu (X, Y) dotrzeć do stanu $(0, NWD(X, Y))$.

W przypadku liczb ujemnych analogicznie możemy przejść do $(0, -NWD(X, Y))$, a następnie naprawić znak przechodząc kolejno do: $(NWD(X, Y), -NWD(X, Y))$, $(NWD(X, Y), 0)$, $(NWD(X, Y), NWD(X, Y))$, $(0, NWD(X, Y))$.

Dodatkowo zauważmy, że każdą z operacji możemy odwrócić. Przykładowo zaczynając od (X, Y) i wykonując `X+` dostajemy $(X+Y, Y)$, co możemy odwrócić wykonując `X-`, tym samym otrzymując z powrotem (X, Y).

Zatem aby przejść z (X_p, Y_p) do (X_k, Y_k) (takich że $NWD(X_p, Y_p) = NWD(X_k, Y_k)$) możemy najpierw przejść z (X_p, Y_p) do $(0, NWD(X_p, Y_p))$, a następnie odwracając operacje przejść z $(0, NWD(X_k, Y_k))$ do (X_k, Y_k).

Dostajemy rozwiązanie działające w czasie $\mathcal{O}(\log(X + Y))$.

Bonus: Polecamy zastanowić się jak uogólnić rozwiązanie na N rejestrów $X_1, X_2, \ldots, X_N$, tak aby działało w czasie $\mathcal{O}(N + \log(\min X_i))$.

5. Turniej par

Posortujmy zadania po trudności, a następnie zastosujmy wyszukiwanie binarne po wyniku.

Mając ustaloną wartość T, którą chcemy osiągnąć możemy dla każdego zadania wyznaczyć minimalne doświadczenie, jakiego potrzebuje, aby czas jego wykonania wyniósł co najmniej T.

Różnych rodzajów par zawodników jest tylko 6, będziemy je oznaczać odpowiednio: MM, MZ, MP, ZZ, ZP, PP.

Zauważmy, że istnieją tylko dwie możliwe kolejności względem sumy doświadczenia:

$$MM \leq MZ \leq MP \leq ZZ \leq ZP \leq PP$$
$$MM \leq MZ \leq ZZ \leq MP \leq ZP \leq PP$$

Wynika stąd, że pary możemy utworzyć zachłannie. Istnieje kilka działających sposobów. Jeden z nich możemy uzyskać wykonując następujące kroki:

- wszystkim zadaniom, którym możemy przypisujemy MM,
- wszystkim zadaniom, którym możemy przypisujemy MZ,
- pozostałe M muszą zostać zużyte w ramach MP,
- wszystkim zadaniom, którym potrzeba przypisujemy PP,
- pozostałe P muszą zostać zużyte w ramach ZP,
- pozostałe Z muszą zostać zużyte w ramach ZZ.

Przypisanie konkretnych par do zadań również możemy zrobić zachłannie, tzn. im trudniejsze zadanie tym mniejszą parę powinno dostać. Na koniec wystarczy sprawdzić czy faktycznie udało się nam osiągnąć co najmniej T w przypadku każdej z par.

Dostajemy rozwiązanie działające w czasie $\mathcal{O}(N(\log N + \log T))$.

Bonus: Implementację da się nieco usprawnić robiąc coś *jeszcze bardziej zachłannego*, tzn. można posortować rodzaje par po ich sumie, a następnie każdemu zadaniu od najłatwiejszego przypisywać najmniejszą możliwą działającą (i jeszcze dostępną) parę. Polecamy zastanowić się nad dowodem takiego podejścia.

6. Czytanie treści

Posortujmy treści po malejącym czasie. Dzięki temu jesteśmy w stanie napisać programowanie dynamiczne – niech $DP[n][k]$ oznacza minimalny koszt przeczy-

tania n pierwszych kupek treści, w taki sposób, że k treści przeczytaliśmy (do tej pory) pojedynczo.

Każdą kolejną treść możemy albo przeczytać pojedynczo płacąc jej koszt (i zwiększając drugi parametr o 1), albo sparować z jakąś już przeczytaną treścią, nie płacąc nic (i zmniejszając drugi parametr o 1). Możemy tak zrobić, gdyż zadbaliśmy o malejące czasy kolejnych kupek.

Zapiszmy teraz wzór na nasze DP (gdzie w oznacza liczbę treści z n-tej kupki, które na razie czytamy pojedynczo):

$$DP[n][k] = \min_{w}(DP[n-1][k + x_n - 2w] + t_n \cdot w)$$

Oczywiście musimy pamiętać o $0 \leq w \leq \min(x_n, k)$.

Niech $X = \max x_i$. W tak zdefiniowanym DPku mamy $\mathcal{O}(NX^2)$ stanów, a każdy z nich możemy obliczyć w czasie $\mathcal{O}(X)$.

Pierwszym krokiem w stronę szybszego rozwiązania jest następująca obserwacja. Zawsze opłaca nam się mieć co najwyżej X niesparowanych jeszcze treści.

Szkic dowodu przebiega następująco.

Jeśli byłby taki moment, że mamy co najmniej $X + 1$ niesparowanych treści, to zarówno przed nim, jak i za nim mamy dwie różne kupki, które w tych sparowaniach biorą udział. Skoro tak, to opłaca nam się coś podmienić, aby otrzymać nie gorsze rozwiązanie, zatem sprzeczność.

W ten sposób zmniejszyliśmy liczbę stanów do $\mathcal{O}(NX)$.

Pozostaje nam już tylko przyspieszenie obliczania kolejnych wartości DP. Można to zrobić na przykład używając drzewa przedziałowego lub struktury RMQ. Szczegóły pozostawiamy do samodzielnego dopracowania.

Dostajemy rozwiązanie działające w czasie $\mathcal{O}(NX \log X)$.

Bonus: Zadanie da się rozwiązać w czasie $\mathcal{O}(NX)$ używając przy tym dosyć prostej struktury danych (w szczególności nie chodzi nam o liniowe RMQ :)).

7. Daj kamienia!

Stwórzmy graf, w którym wierzchołkami są puste pola planszy, a krawędzie utwórzmy pomiędzy sąsiednimi polami.

Okazuje się, że w tej grze drugi gracz wygrywa wtedy i tylko wtedy, gdy w tak zbudowanym grafie istnieje skojarzenie doskonałe (perfect matching). Uzasadnijmy sobie to.

Jeśli w grafie istnieje skojarzenie doskonałe, to niezależnie od wyboru pierwszego wierzchołka, drugi gracz może przejść (położyć kamień) do wierzchołka skojarzonego z nim. Następnie pierwszy gracz musi wybrać pewien inny wierzchołek (nieskojarzony z aktualnym), a drugi gracz znowu może odpowiedzieć wybierając

wierzchołek skojarzony z tym innym wierzchołkiem. Stosując tę strategię drugi gracz zawsze może zagwarantować sobie zwycięstwo.

Jeśli w grafie nie istnieje skojarzenie doskonałe, to znajdźmy w nim największe skojarzenie. Niech pierwszy gracz wybierze dowolny nieskojarzony wierzchołek. Teraz role się odwracają (i odbywa się to co w poprzednim przypadku). Drugi gracz musi wybrać pewien wierzchołek, który jest skojarzony (ale nie z obecnym), a następnie pierwszy gracz może odpowiedzieć wybierając skojarzony do niego wierzchołek. Drugi gracz nigdy nie będzie w stanie wybrać nieskojarzonego wierzchołka, gdyż powodowałoby to istnienie ścieżki powiększającej, a znaleźliśmy największe skojarzenie.

Powyższą argumentację można uogólnić na liczenie wygrywających wierzchołków startowych, tzn. wierzchołek startowy jest wygrywający wtedy i tylko wtedy, gdy istnieje największe skojarzenie nie zawierające go.

Ten warunek, dla pojedynczego wierzchołka, można sprawdzić porównując rozmiar największego skojarzenia, z rozmiarem największego skojarzenia po usunięciu tego wierzchołka z grafu. Jeśli rozmiar pozostał taki sam, to takie skojarzenie istnieje, a jeśli spadł o 1, to takie skojarzenie nie istnieje. Jako że plansza to graf dwudzielny, to do znajdowania skojarzeń możemy użyć swojego ulubionego algorytmu (np. Hopcrofta-Karpa).

Dostajemy rozwiązanie działające w czasie $\mathcal{O}(N^2M^2\sqrt{NM})$.

Bonus: Zadanie da się rozwiązać w czasie $\mathcal{O}(NM\sqrt{NM})$ znajdując skojarzenie tylko raz. Mając już największe skojarzenie wyznaczone, wystarczy sprawdzić, dla każdego skojarzonego wierzchołka, czy istnieje ścieżka alternująca wychodząca z niego o parzystej długości.