Omówienie zadań z Turnieju 23 (kwalifikacje do Mistrzostw)

Karol Pokorski

1 Tempo biegu

Na wejściu podana jest prędkość w kilometrach na godzinę. Celem jest obliczyć czas (powiedzmy, że w sekundach) potrzebny na pokonanie jednego kilometra.

Prędkość V podana na wejściu oznacza, że w ciągu 3 600 sekund pokonujemy V kilometrów, a więc jeden kilometr pokonywany jest w ciągu $\frac{3\,600}{V}$ sekund. Wynik należało zaokrąglić w dół (na przykład rzutując wynik dzielenia na typ `int`). Niech wynik po zaokrągleniu to S sekund. Trzeba go odpowiednio sformatować (wypisać liczbę minut oraz sekund, oddzielając je dwukropkiem i dopełniając zerami).

Liczba minut, które należy wypisać to $\lfloor \frac{S}{60} \rfloor$, zaś liczba sekund to reszta z dzielenia S przez 60 (operacja *modulo*, operator `%` w C++). Należało jeszcze pamiętać o wypisaniu dodatkowych zer, jeżeli liczba minut lub sekund jest mniejsza niż 10.

Niektóre błędne odpowiedzi u zawodników brały się z niedokładności typów zmiennoprzecinkowych. Przykładowo, dla testu, w którym $V = 50$, prawidłowa odpowiedź to dokładnie 72 sekundy (`01:12`), a wiele programów, które w skutek niedokładności obliczeń uzyskiwały 71.99999... zaokrąglały ten wynik do `01:11`.

2 Antypalindromiczność

Kluczową obserwacją w tym zadaniu jest to, że wystarczy, żeby słowo nie zawierało palindromu długości 2 lub 3, żeby nie zawierało żadnego palindromu. A zatem, w wypisanym napisie S musi zachodzić warunek $S[i] \neq S[i+1]$ oraz $S[i] \neq S[i+2]$ dla każdego i, dla którego ma to sens.

Przykładowym napisem, który spełnia warunki zadania jest więc `abcabcabcabc`....

3 Maxdiff

Największą różnicę tworzy najbardziej skrajna para liczb w strukturze, a więc liczba największa z najmniejszą. A więc jednym ze sposobów rozwiązania tego zadania było skorzystanie ze struktury danych przechowującej uporządkowany zbiór (`std::set` w przypadku C++).

Struktura umożliwia wstawianie i usuwanie elementu w czasie $\mathcal{O}(\log n)$ oraz sprawdzanie czy dany klucz znajduje się w strukturze również w czasie $\mathcal{O}(\log n)$ (jest to pomocne do zignorowania operacji `insert` na liczbach, które są w zbiorze oraz operacji `erase` na liczbach, których nie ma w zbiorze). Ponieważ utrzymywany zbiór jest uporządkowany i mamy dostęp do iteratora oraz odwrotnego iteratora, możliwy jest łatwy dostęp do elementu najmniejszego i największego (`*(S.begin())` oraz `*(S.rbegin())`).

Ponieważ rozmiar danych był duży, należało również pamiętać o wyłączeniu synchronizacji operacji I/O z C i C++ (`ios_base::sync_with_stdio(false)`). Dla potrzeb wydajnościowych sensowne jest również odseparowanie strumieni wejścia i wyjścia (`cin.tie(0)`).

4 Dobry LIS

W zadaniu należy policzyć najdłuższy podciąg rosnący, w którym suma każdych dwóch sąsiednich wartości jest liczbą pierwszą.

4.1 Algorytm na LISa

Wzbogacimy w tym celu standardowy algorytm na LIS oparty o programowanie dynamiczne. W algorytmie tym utrzymujemy tablicę $LIS_i[]$, w której komórka $LIS_i[l]$ oznacza najmniejszy możliwy element jakim może kończyć się pewien podciąg rosnący o długości l z ciągu $T[1], T[2], \ldots, T[i]$. Algorytm ten bazuje na obserwacji, że tablice $LIS_{i+1}[]$ oraz $LIS_i[]$ różnią się jedynie jednym elementem, który łatwo znaleźć: poszukujemy po prostu największego elementu, który jest mniejszy niż $T[i+1]$. Tylko za nim można oraz opłaca się podmienić komórkę $LIS_{i+1}[]$ na $T[i+1]$. A ponieważ tablica $LIS.[]$ jest stale uporządkowana, możemy zastosować wyszukiwanie binarne do znalezienia tego elementu. Niepotrzebne jest też utrzymywanie wielu różnych tablic LIS_i dla różnych parametrów i, zaś całość można zamknąć w jednej tablicy LIS przechowującej LIS_i dla bieżącego $i \in \{1, 2, \ldots, N\}$ rozpatrywanego w danym momencie.

4.2 Algorytm na dobrego LISa

Skorzystamy z tego algorytmu, jednak w komórkach $LIS[l]$ nie będziemy utrzymywali jedynie najmniejszego elementu dla danej długości l. Zamiast tego, każda komórka tablicy $LIS[l]$ będzie utrzymywała uporządkowany zbiór wszystkich elementów jakie mogą być na końcu dobrego ISa o długości l (o ile te elementy nie mogłyby być ostatnimi elementami jeszcze dłuższych dobrych ISów).

Mając obliczony stan tablicy $LIS_i[]$ i chcąc obliczyć $LIS_{i+1}[]$, postępujemy podobnie jak w oryginalnym algorytmie. Ponownie chcemy znaleźć jak największą długość dobrego ISa, który kończy się elementem $T[k]$ mniejszym niż $T[i+1]$, że wartość $T[k] + T[i+1]$ jest liczbą pierwszą.

Liczby pierwsze są dość gęsto rozłożone w zbiorze liczb naturalnych: w zbiorze liczb $\{1, 2, \ldots, M\}$ znajduje się $\Theta(\frac{M}{\log M})$ liczb pierwszych. Ponieważ w treści zadania podane było, że dane są losowe, możliwe jest naiwne przejrzenie (przeciętnie $\Theta(\log M)$) kandydatów na poprzedzający $T[i+1]$ element najdłuższego dobrego ISa, poczynając od kandydatów mniejszych niż $T[i+1]$ kończących możliwie najdłuższe ISy. Długość l takiego przedłużalnego ISa ponownie możemy wyszukiwać binarnie, zaś już w samym zbiorze $LIS[l]$ kandydatów możemy sprawdzać po kolei iteratorem na strukturze `std::set` (od najmniejszego do największego). Jeżeli dojdziemy do końca listy lub do kandydata, który jest większy niż $T[i+1]$, przechodzimy do listy $LIS[l-1]$ i ponawiamy wyszukiwanie, w razie potrzeby analizując listy $LIS[l-2], LIS[l-3], \ldots$. W ten sposób koszt znalezienia stosownego przedłużalnego dobrego ISa jest proporcjonalny do liczby przejrzanych kandydatów plus $\mathcal{O}(\log N)$ na znalezienie wartości l.

4.3 Sprawdzanie pierwszości

Aby wydajnie sprawdzać czy liczba do $2 \cdot 10^9$ jest liczbą pierwszą, najlepiej sprawdzać ewentualną podzielność jedynie przez liczby pierwsze do zakresu $\sqrt{2 \cdot 10^9}$ (czyli wcześniej przygotować sobie tablicę liczb pierwszych o tej długości, na przykład za pomocą sita).

Możliwe, choć wcale niekoniecznie przynoszące lepszy efekt, jest również zastosowanie szybszych asymptotycznie algorytmów sprawdzania pierwszości: np. faktoryzacji „rho" Pollarda lub testów pierwszości (np. Millera-Rabina).

4.4 Odzyskiwanie wyniku

Należy pamiętać, że tablica $LIS[]$ w oryginalnym problemie nie reprezentuje LISa, lecz najmniejsze elementy kończące ISa o danej długości. Aby odzyskać wynik najłatwiej jest więc zapamiętywać dla każdego dokładanego elementu, kto był jego poprzednikiem (dbając o to, żeby przez przypadek sobie tego poprzednika nie nadpisać w późniejszych iteracjach pętli, na przykład dla powtarzających się wartości). Wtedy możliwe jest standardowe odzyskanie całego dobrego LISa, a nie tylko jego długości, cofając się po ścieżce, zaczynając od dowolnego elementu na ostatniej liście $LIS[]$.

4.5 Uwagi końcowe

Jak się okazuje, optymalna długość dobrego LISa dla losowych danych nie jest zbyt duża. Możliwe więc, że system zaakceptował również rozwiązania, które nie używają wyszukiwania binarnego, a zamiast tego liniowo znajdują największą długość l ISa, który można przedłużyć elementem $T[i+1]$. Realnie bowiem czas rozwiązania okazuje się być zdominowany przez sprawdzanie pierwszości sum $T[i+1]$ z potencjalnymi poprzednikami w dobrym LISie.

5 Zliczanie MISów

Spróbujmy rozwiązać to zadanie grafowo. Wierzchołkami grafu będą pozycje w ciągu. Stworzymy skierowaną krawędź w grafie między pozycją i oraz j, jeżeli $T[i]$ może bezpośrednio poprzedzać $T[j]$ w MISie. Dzieje się tak wtedy, gdy $i < j$, $T[i] < T[j]$ oraz $T[j]$ jest najmniejszym spośród elementów $\{T[i+1], T[i+2], \ldots, T[j-1], T[j]\}$.

5.1 Rozmiar grafu

Najpierw zauważmy, że mamy prawo oczekiwać, że utworzony graf będzie niezbyt duży. Zastanówmy się, które pozycje zostaną końcami krawędzi wychodzącej z pozycji i. Spośród pozycji $i+1, i+2, \ldots, N$ skupiamy się jedynie na tych pozycjach, dla których wartości w ciągu T są większe niż $T[i]$. Niech te (tylko większe niż $T[i]$) wartości, w kolejności od lewej do prawej tworzą ciąg C. Rozważmy minima prefiksowe ciągu C. Prawdopodobieństwo, że j-te minimum prefiksowe jest inne niż $(j-1)$-sze minimum prefiksowe, jest rzędu $\frac{1}{j}$ przy losowym doborze wartości ciągu T (a taki mieliśmy gwarantowany w treści zadania). Czyli z takim prawdopodobieństwem mamy jedną zmianę minimum (i jedną nową krawędź w grafie), w pozostałych przypadkach krawędzi nie tworzymy. Suma tych prawdopodobieństw (albo raczej wartości oczekiwanych tych zmiennych losowych) daje wartość oczekiwaną liczby krawędzi, które będą wychodziły z i. Możemy tę wartość ograniczyć z góry przez $\frac{1}{1} + \frac{1}{2} + \ldots + \frac{1}{N} = \Theta(\log N)$, a więc łączna liczba krawędzi jest $\mathcal{O}(N \log N)$.

5.2 Obliczanie liczby MISów na podstawie grafu

Zakładając, że mamy już obliczony graf, o którym mowa w poprzednich akapitach, możemy obliczyć liczbę MISów. Każda maksymalna ścieżka (od wierzchołka o stopniu wejściowym 0, do wierzchołka o stopniu wyjściowym 0) reprezentuje pewien podzbiór pozycji, którego elementy tworzą MISa.

Liczbę takich ścieżek możemy obliczyć programowaniem dynamicznym: przeglądamy graf w naturalnym porządku topologicznym i sumujemy wyniki z krawędzi wchodzących do każdego wierzchołka. Należy oczywiście pamiętać o ustawieniu wyniku na 1, gdy mamy do czynienia z wierzchołkami startowymi i o modulowaniu wyników przez $10^9 + 7$ przy każdym dosumowaniu.

5.3 Konstrukcja grafu

Pozostaje pytanie jak szybko skonstruować żądany graf pozycji. Możemy go konstruować zaczynając od pozycji, na których stoją największe wartości.

Utrzymujemy drzewo przedziałowe indeksowane pozycjami w ciągu. W momencie obliczania krawędzi wychodzących z wierzchołka i, w węzłach drzewa reprezentującym przedziały pozycji w ciągu przechowywane są uporządkowane zbiory wartości elementów większych niż $T[i]$ z tego przedziału pozycji.

Poszukując następnej krawędzi wychodzącej z i po wcześniej wstawionej krawędzi do j, rozbijamy przedział $j+1 \ldots N$ na przedziały bazowe. Przeglądamy przedziały bazowe od lewej do prawej i znajdujemy pierwszy, w którym znajduje się choć jedna wartość mniejsza niż $T[j]$. Ponieważ w tym przedziale może się znajdować potencjalnie wiele takich wartości, a my chcemy znaleźć nie najmniejszą, a tę, która jest na najwcześniejszej pozycji, możemy zacząć rozbijać ten przedział bazowy na mniejsze podprzedziały (zacząć iść w dół drzewa przedziałowego), za każdym razem preferując lewego syna, o ile zawiera on chociaż jeden element mniejszy niż $T[j]$, idąc do prawego syna tylko w przeciwnym przypadku. W ten sposób możemy wygenerować kolejną krawędź grafu w czasie $\mathcal{O}(\log N)$.

Ponieważ oczekiwana liczba krawędzi grafu jest $\Theta(N \log N)$, otrzymaliśmy rozwiązanie o oczekiwanym czasie działania $\mathcal{O}(N \log^2 N)$.

6 Kwadrat z punktami

Zauważamy najpierw, że wystarczy sprawdzać tylko kwadraty, które mają co najmniej jeden punkt na swoim lewym boku, co najmniej jeden punkt na swoim dolnym boku oraz co najmniej jeden punkt na boku prawym lub górnym.

Robimy pętlę, która zgaduje pozycję boku lewego i dolnego ($\mathcal{O}(N^2)$ iteracji). Następnie zgadujemy rozmiar boku kwadratu. Ponieważ im większy bok kwadratu, tym więcej punktów w jego wnętrzu, możemy do tego celu zastosować wyszukiwanie binarne.

6.1 Przeindeksowanie współrzędnych

Żeby sprawdzić ile punktów jest we wnętrzu konkretnego kwadratu, możemy zastosować sumy prefiksowe. Na początku programu można zrobić reindeksowanie wszystkich współrzędnych, żeby najmniejsza współrzędna otrzymała indeks 1, druga najmniejsza indeks 2 itd. Można w tym celu zastosować strukturę `std::map` lub stworzyć tablicę par (pozycja, przeindeksowana pozycja). Aby przeindeksować później oryginalną współrzędną, można użyć operacji `lower_bound` lub wyszukiwania binarnego.

6.2 Sumy prefiksowe

Następnie możemy stworzyć tablicę $T[][]$. Punkt (x, y) zaznaczamy jako jedynka na pozycji $T[r(x)][r(y)]$, gdzie $r()$ to funkcja reindeksacji współrzędnej. Następnie z tablicy $T[][]$ możemy policzyć tablicę sum prefiksowych $S[][]$. Wartość $S[x][y]$ oznacza więc liczbę punktów leżących na lewo od reindeksowanej współrzędnej x oraz w dół od reindeksowanej współrzędnej y. Aby policzyć ile punktów zawiera konkretny kwadrat między reindeksowanymi współrzędnymi (x_1, y_1) oraz (x_2, y_2) wystarczy skorzystać z (zasady włączeń i wyłączeń) wyrażenia:

$$S[x_2][y_2] - S[x_1 - 1][y_2] - S[x_2][y_1 - 1] + S[x_1 - 1][y_1 - 1]$$

6.3 Uwagi końcowe

Prowadzi to do rozwiązania $\mathcal{O}(N^2 \log^2 N)$, które przy dobrej implementacji powinno być akceptowane przez system zawodów. Możliwe jest niewielkie usprawnienie tego rozwiązania i nie uruchamianie wyszukiwania binarnego dla każdej pary (lewy bok, dolny bok). Ustalmy losową kolejność analizy tych par. Powiedzmy, że przeanalizowaliśmy już jakiś prefiks tego ciągu i najlepszy dotychczasowy kwadrat ma bok o długości l. Jeżeli przeglądamy następną parę i kwadrat o boku długości l zawiera za mało punktów, to nie ma sensu sprawdzać tego $\Theta(\log N)$ razy, żeby i tak na końcu stwierdzić, że to jest gorszy wynik niż najlepszy dotychczas znany. A ponieważ, jak już wiemy z poprzedniego zadania, minimum losowego ciągu zmienia się rzadko... wiele z tych wyszukiwań będzie można ominąć.

7 Superpermutacja

W treści zadania było podane sporo linków do materiałów popularnonaukowych w internecie i prac naukowych. W szczególności: jeden z najbardziej plusowanych komentarzy do ostatniego zalinkowanego filmu o superpermutacji zawiera superpermutację o długości 5 907 dla $N = 7$.

Inny zaś link wyjaśnia prostą, naiwną konstrukcję, która na podstawie superpermutacji dla N, generuje stosunkowo krótką (ale nie dość krótką na potrzeby naszego zadania) superpermutację dla $N + 1$.

W tym zadaniu należało przejrzeć te materiały i połączyć kropki: zamiast stosować standardową konstrukcję zaczynając od superpermutacji dla $N = 1$ i już dla $N = 6$ osiągnąć suboptymalne rozwiązania, lepiej zacząć od superpermutacji z komentarza dla $N = 7$, która względem standardowej konstrukcji ma już „oszczędzone" sześć znaków. Otrzymamy w ten sposób superpermutację dla $N = 9$ o długości 409 107, o jeden znak krótszą niż limit z zadania.

8 Funkcja różnowartościowa

Zastosujemy podejście grafowe. Wierzchołki reprezentujące wartości $f(x)$ dostają skierowaną krawędź do wierzchołków reprezentujących wartości $g(x)$.

Ponieważ f jest różnowartościowa, każda spójna składowa n wierzchołków tego grafu ma albo n krawędzi, albo $n-1$ krawędzi. Innymi słowy: albo jest drzewem, w którym wszystkie krawędzie zbiegają do jednego ujścia (jeszcze nieużywanej wartości $g(\cdot)$), albo składową złożoną z cyklu, do którego zbiegają być może podczepione do niego drzewa (nazwijmy takie składowe *meduzami*).

8.1 Rozpoznawanie i naprawianie drzew

Możliwe jest zidentyfikowanie drzew, poprzez rozpoczęcie od wierzchołków o stopniu wyjściowym 0. W każdym takim drzewie, jeżeli zdecydujemy się podmienić wartość $f(x)$ na $g(x)$ (wybrać jakąś krawędź, żeby nią podążyć), to musimy też wybrać krawędź od $g(x)$ i tak dalej aż do osiągnięcia korzenia, gdzie ten ciąg wymuszeń się kończy. Nie możemy wtedy jednak wybrać żadnej innej krawędzi.

Oznacza to, że z każdej składowej, która jest drzewem można wybrać maksymalnie tyle krawędzi ile liczy sobie najdłuższa ścieżka do korzenia w tym drzewie. Tę można znaleźć prostym programowaniem dynamicznym na drzewie. Uruchamiamy DFS na grafie składającym się z odwróconych krawędzi i zakładamy, że wyniki dla liści są pustymi ścieżkami, a wyniki dla węzłów wewnętrznych powstają z przedłużenia najdłuższej ścieżki przyniesionej przez jakiegoś z synów o jego krawędź do ojca.

8.2 Rozpoznawanie i naprawianie meduz

Wszystkie wierzchołki, które nie zostały dotychczas odwiedzone, należą do składowych będących meduzami. W każdej takiej składowej możemy ewentualnie podjąć decyzję o wybraniu wszystkich krawędzi leżących na cyklu lub pozostawieniu funkcji niezmienionej. Oczywiście, przy celu zadania, chcemy zidentyfikować wszystkie cykle wszystkich meduz i zaznaczyć wszystkie występujące tam krawędzie.

Łatwo zagwarantować, odpowiednim oznaczaniem kolejno odwiedzanych wierzchołków, żeby procedura przeglądająca wierzchołki przetwarzała każdy wierzchołek drzewa meduzy tylko jeden raz, zaś każdy wierzchołek jej cyklu co najwyżej dwa razy (raz przy pierwszym wejściu do niego i raz, za pierwszym razem, gdy już wiemy, że ten wierzchołek leży na cyklu). W ten sposób procedura wybierająca krawędzie będzie działała w czasie liniowym.

9 Zwiększanie wyniku

Kluczowa obserwacja w tym zadaniu jest taka, że chcemy za każdym razem zwiększyć o 1 najmniejszą liczbę w ciągu. W istocie: zamiana liczby z k na $k+1$, mnoży cały iloczyn wszystkich liczb przez czynnik $\frac{k+1}{k}$. Ułamek jest tym większy im mniejsza jest wartość k.

Niestety, z ograniczeń zadania wynika, że takich zwiększeń chcemy wykonać $K \leqslant 10^9$. Chciałoby się powiedzieć, że skoro wynik nie może być duży, to nie da się zrobić zbyt wielu zwiększeń, ale nie jest to prawda i w zbiorze testów znalazły się odpowiednio złośliwie dobrane przypadki.

Kluczowe jest precyzyjne zrozumienie, co dokładnie wynika z faktu, że iloczyn liczb po wykonaniu K zwiększeń nie może przekroczyć 10^{18}. Najpierw możemy (i musimy) wykonać wszystkie zwiększenia zer na jedynki. Potem już możemy powiedzieć, że w powstałym ciągu ani przed, ani po wykonaniu pozostałych zwiększeń nie może się znajdować zbyt wiele różnych liczb. W istocie: $1 \cdot 2 \cdot 3 \cdot \ldots \cdot 20 > 10^{18}$, a więc jeżeli w ciągu (już bez zer) znajdzie się więcej niż 20 różnych liczb, to ich iloczyn przekroczy dozwolony w ograniczeniach limit.

Możliwe jest więc zgrupowanie takich samych liczb (zliczając ile liczb każdego typu mamy) i zwiększanie wszystkich liczb leżących w najmniejszej grupie jednocześnie do wartości drugiej najmniejszej grupy (lub do mniejszej wartości, jeżeli miałoby się okazać, że szybciej wyczerpiemy limit K zwiększeń, lub, w jeszcze innym przypadku, zwiększenie tylko K liczb z najmniejszej grupy o 1). Każde takie zwiększenie albo kończy program (zmniejsza K do zera), albo zmniejsza liczbę grup różnych liczb, co ogranicza liczbę iteracji do 20.