A: Tree

Na wejściu jest n-wierzchołkowe drzewo T i szukamy takiej jego orientacji, która maksymalizuje liczbę par różnych wierchołków (a, b), takich że istnieje ścieżka z a do b.

Można udowodnić, że istnieje rozwiązanie optymalne następującej postaci: wybieramy pewien wierzchołek v, dzielimy składowe grafu $T - v$ na dwie grupy: te, których każdy wierzchołek ma mieć ścieżkę do v i te, których każdy wierzchołek ma mieć ścieżkę od v, a następnie orientujemy krawędzie w jedyny sposób, który to osiąga.

Dowód opiera się na następującym lemacie: żadna nieskierowana ścieżka $x_1, \ldots, x_k$ nie będzie miała w optymalnej orientacji dwóch „zmian kierunku", czyli wierzchołków x_i, x_j takich, że $x_{i-1} \to x_i \leftarrow x_{i+1}$ oraz $x_{j-1} \leftarrow x_j \to x_{j+1}$ – gdyby miała, można by odwrócić orientację w pewnej części drzewa i dostać lepsze rozwiązanie. Pełny (niezbyt trudny) dowód jest np. w pracy `https://arxiv.org/pdf/cs/0205042.pdf`.

Przy ustalonych v oraz podziale składowych na grupy, wynik to wartość $S \cdot T + X$, gdzie S i T to liczby wierzchołków w składowych mających ścieżkę odpowiednio do i od v, a X to liczba par różnych wierzchołków (a, b), takich że jeśli ukorzenimy T w wierzchołku v, to a jest potomkiem b. Wartość X jest niezależna od podziału składowych, więc łatwo zauważyć, że wynik będzie maksymalny, jeżeli wartości S i T będą najbliżej siebie.

Algorytm przebiega następująco. Najpierw dla każdego v obliczamy wartość X. Da się to zrobić w czasie liniowym, ale wystarczyło to zrobić w czasie kwadratowym. Następnie rozpatrujemy każdy wierzchołek v. Najpierw obliczamy listę wielkości składowych grafu $T - v$, co nie jest trudne, a potem rozwiązujemy instancję problemu plecakowego: znajdujemy podzbiór tej listy sumujący się do największej wartości niewiększej niż $\lfloor (n-1)/2 \rfloor$ – to jest nasza wartość S. W ten sposób dla danego v rozpatrzyliśmy maksymalne $S \cdot T + X$.

Ponieważ pojedyncza instancja problemu plecakowego jest rozwiązywana w czasie $\mathcal{O}(\deg(v) \cdot n)$, gdzie $\deg(v)$ oznacza stopień wierzchołka v, to czas potrzebny na rozwiązanie wszystkich instancji wynosi $\mathcal{O}(\sum_{v \in V(T)} \deg(v) \cdot n) = \mathcal{O}(n^2)$, i taka też jest złożoność całego algorytmu.

Okazuje się, że istnieje też prostsze rozwiązanie. Można udowodnić (znowu pomocna może być praca `https://arxiv.org/pdf/cs/0205042.pdf`), że opłaca się za v wybrać centroid drzewa T. Centroid to taki wierzchołek v, że każda składowa lasu $T - v$ ma co najwyżej $n/2$ wierzchołków (zauważ, że w każdym drzewie istnieje centroid). Wystarczy więc rozwiązać problem plecakowy raz, a nie n razy. Co do złożoności, przykład gwiazdy (drzewa, w którym każdy wierzchołek różny od korzenia jest jego dzieckiem) pokazuje, że pesymistyczny czas działania nowego rozwiązania również wynosi $\mathcal{O}(n^2)$, choć oczywiście w praktyce działa ono zauważalnie szybciej.

B: Bipartite graph

Dla każdej krawędzi $e \in E$ chcemy powiedzieć, czy graf $G - e$ jest dwudzielny. Tworzymy strukturę danych bardzo podobną do Find-Union, która trzyma spójne składowe grafu, a w każdej składowej dodatkowo pamięta, które wierzchołki są „czarne", a które „białe". Struktura ta potrafi robić następujące operacje:

- $Find(x)$ – podaj składową i kolor wierzchołka x,

- $Union(x, y)$ – połącz składowe krawędzią (x, y), poprawiając kolory,
- $Undo()$ – cofnij ostatni $Union$.

Mając taką strukturę, zadanie można zrobić metodą *divide-and-conquer*:

- dzielimy E na dwa w przybliżeniu równe zbiory $E = E_1 \cup E_2$;
- dla każdego $(x, y) \in E_1$ sprawdzamy, czy $Find(x) = Find(y)$:
  - jeśli tak, to sprawdzamy też, czy kolory x, y w tej składowej są równe – jeśli tak, to E_1 nie jest dwudzielny i odpowiedź dla każdej krawędzi z E_2 jest NIE;
  - jeżeli $Find(x) \neq Find(y)$, to robimy $Union(x, y)$;
- teraz mamy już znane składowe spójne, które generują krawędzie z E_1; jeśli E_1 był dwudzielny, wywołujemy się rekurencyjnie na E_2;
- cofamy wszystkie operacje na zbiorze do samego początku;
- powtarzamy całą procedurę, zamieniając rolami E_1 i E_2.

Jeśli rekursja wywoła się na jednoelementowym zbiorze krawędzi, odpowiedź jest TAK.

Pozostaje jeszcze pytanie, jak zrobić strukturę. Używamy klasycznego lasu zbiorów rozłącznych, ale każdy wierzchołek trzyma v dodatkową flagę $Mark(v)$ – jeśli $Mark(v) = true$, to wszystkie dzieci v mają odwrócony kolor. W lesie stosujemy łączenie według rang, bez kompresji ścieżek. Teraz łatwo jest policzyć kolor dowolnego wierzchołka (jest to liczba zamarkowanych wierzchołków na ścieżce do korzenia modulo 2), łatwo też obrócić kolory w całej składowej (po prostu zmieniamy stan korzenia składowej). To pozwala nam zrobić $Union$, łącząc korzenie składowych, w razie potrzeby odwracając kolor jednej z nich. Aby móc cofnąć $Union$, przy każdej operacji odkładamy na stosie każdy wierzchołek, w którym cokolwiek się zmieniło. Przy cofaniu ściągamy wierzchołki ze stosu i przywracamy ich stan. Operacje na strukturze działają w czasie $\mathcal{O}(\log n)$, gdzie n jest rozmiarem struktury. W jednym wywołaniu procedury mamy liczbę operacji proporcjonalną do liczby krawędzi, jakie dostaniemy. Stąd całkowita złożoność jednej procedury (nie licząc wywołań rekurencyjnych) to $\mathcal{O}(E \log E)$, a całego algorytmu – $\mathcal{O}(E \log^2 E)$.

C: Graph

W tym zadaniu mając nieskierowany graf spójny G byliśmy proszeni o znaleźienie cyklu Hamiltona w grafi G^3. Krawędź $\{x, y\}$ w G^3 jest wtedy i tylko wtedy, kiedy minimalna odległość pomiędzy x a y w G jest nie większa niż 3. Okazuje się, że cykl Hamiltona w takim grafie G^3 zawsze istnieje. Rozwiążemy ten problem dla drzew, a potem w oryginalnym grafie G weźmiemy dowolne drzewo rozpinające i odpalimy na nim nasz algorytm.

Niech mamy pewne drzewo ukorzenione T, v to jest korzeń tego drzewa, a przez $v_1, v_2, \ldots, v_m$ oznaczymy bezpośrednich dzieci v. Przez T_u będziemy oznaczać poddrzewo drzewa T, które ma korzeń w u. Zdefiniujmy sobie dwie funkcje pomocnicze:

- $f(u)$ zwraca nam ścieżkę Hamiltona w T_u^3, która zaczyna się w u, a kończy się w jakimś bezpośrednim dziecku u.

- $g(u)$ zwraca nam ścieżkę Hamiltona w T_u^3, która zaczyna się w jakimś bezpośrednim dziecku u, a kończy się w u.

Dodamy tylko, że jeśli T_u jest drzewem jednoelementowym, to $f(u) = g(u) = [u]$. Odpowiedzią do naszego zadania jest po prostu $f(v)$. Teraz powstaje pytanie, jak liczyć $f(v)$ i $g(v)$? Można łatwo zauważyc, że:

- $f(v) = [v] + g(v_1) + g(v_2) + \ldots + g(v_m)$.
- $g(v) = f(v_1) + f(v_2) + \ldots + f(v_m) + [v]$.

gdzie + oznacza konkatenację ciągów. Możemy rekurencyjnie wyliczyć $f(v_1), \ldots, f(v_m)$ i $g(v_1), \ldots, g(v_m)$, a potem wyliczyć $f(v)$ i $g(v)$. Ponieważ dla każdego wierzchołka chcemy wyliczyć tylko albo $f(v)$, albo $g(v)$, możemy zrobić to wydajnie, po prostu odpalając algorytm *DFS* i dodając wierzchołki w dobrej kolejności do pewnej globalnej listy albo wektora.

Złożoność: $\mathcal{O}(n + m)$, gdzie n - liczba wierzchołków w G, a m - liczba krawędzi w G.

D: Sequence

Mając dany ciąg liczb całkowitych $a_1, \ldots, a_n$, spełniający dodatkowy warunek $1 \leq a_i \leq i$, chcemy go podzielić na dwa podciągi o równej sumie wyrazów. Jeżeli suma wszystkich wyrazów ciągu jest nieparzysta, taki podział jest niemożliwy. Okazuje się, że w przeciwnym wypadku podział zawsze jest możliwy. Iterujemy się po i od n do 1 i wybieramy $b_i \in \{-1, 1\}$ tak żeby zminimalizować wartość bezwzględną sumy $a_i b_i + \cdots + a_n b_n$. Utrzymujemy w ten sposób niezmiennik, że wartość ta jest mniejsza lub równa i. Po zakończeniu iteracji, wartość ta należy do zbioru $\{-1, 0, 1\}$, a że jej parzystość jest zdeterminowana, to w interesującym nas przypadku wynosi 0.

Alternatywne rozwiązanie opiera się na obserwacji, że dowolny ciąg $a_1, \ldots, a_n$ liczb całkowitych nieujemnych, spełniający dla każdego i warunek $1 + a_1 + a_2 + \cdots + a_{i-1} \geq a_i$, pozwala wygenerować jako sumę podciągu dowolną liczbę całkowitą z przedziału od 0 do $a_1 + \cdots + a_n$. Dowód tej obserwacji, indukcyjny po n, jest konstruktywny.

Oba podejścia dają algorytm działający w czasie $\mathcal{O}(n)$.

E: Subsequence

W tym zadaniu chodziło o to, żeby w podanym na wejściu ciągu $a_1, a_2, \ldots, a_N$ znaleźć najdłuższy podciąg-rollercoaster (dokładna definicja była podana w treści).

Z pewnością każdy doświadczony zawodnik szybko zobaczy, że tak naprawdę to zadanie wymaga zaaplikowania programowania dynamicznego. Niech $f(type, len, p)$ to jest długość takigo najdłuższego podciągu ciągu a, który jest prawie rollercoasterem (jest rollercoasterem pod warunkiem, że nie będziemy pod uwagę ostatniego maksymalnego fragmentu monotonicznego), który kończy się na elemencie a_p; długość ostatniego maksymalnego fragmentu monotonicznego jest równa len, a $type$ to jest jego typ (czy ostatni fragment jest rosnący czy malejący). Zauważmy też, że nas niezbyt interesuje, czy ostatni fragment ma długość 3 czy więcej, niż 3, dlatego możemy sobie założyć, że $1 \leq len \leq 3$,

a jeśli $len = 3$ to nam chodzi o to, że długość ostatniego maksymalnego fragmentu monotonicznego jest nie mniejsza, niż 3. W takim przypadku liczba stanów tego dynamika to $\mathcal{O}(N)$, ale jedno przejście dynamika trwa $\mathcal{O}(N)$, ponieważ musimy ustalić pozycję poprzedniego elementu w konstruowanym podciągu. Dlatego złożoność całości wynosi $\mathcal{O}(N^2)$.

Niestety, takie rozwiązanie jest zbyt wolne. Możeby delikatnie zmienić nasz dynamik w następujący sposób.

Będziemy iterować się po ciągu a i liczyć funkcję $g(type, len, value)$, która jest zdefiniowana prawie tak samo, jak funkcja f, z tym wyjątkiem, że nasz ciąg kończy się na elemencie, który znajduje się przez rozważaną pozycją, a jego wartość wynosi $value$. Pozornie wydaje się, że taka funkcja w żaden sposób nam nie pomoże, ale to nie jest tak! Jeśli ustalimy sobie znaczenia $type$ i len (a ich jest $\mathcal{O}(1)$), to wtedy nas będzie interesować pewien spójny przedział wartości $value$! A to znaczy, że zamiast liczyć te wartości na tablicy, możemy dla ustaloch $type$ i len trzymać drzewo przedziałowe, które pozwala nam szukać maksimum na przedziale i modyfikować znaczele elementu na pewnej pozycji.

Warto tylko dodać, że wartości elementów mogą być duże, ale na samym początku możemy zrobić preprocessing i skompresować wszystkie wartości do prezdziału $[0, N-1]$ w taki sposób, żeby jeśli do zmiany $a_i \leq a_j$ to $new(a_i) \leq new(a_j)$. Można to zrobić w czasie $\mathcal{O}(N \log N)$.

Złożoność: $\mathcal{O}(N \log N)$, gdzie N - liczba elementów w ciągu a.

F: Sequence and graph

Jeśli pakując n smaków utworzymy k par, a pozostałe $n - 2k$ smaków zapakujemy do torebek pojedynczo, to sumaryczna liczba torebek wyniesie $(n - 2k) + k = n - k$. Tym samym minimalną sumaryczną liczbę torebek osiągniemy, jeśli utworzymy maksymalną możliwą liczbę par.

Przy ustalonym n, niech $D(n)$ oznacza zbiór zawierający wszystkie liczby pierwsze ściśle większe od $\frac{n}{2}$ i nie większe od n, oraz dodatkowo liczbę 1. Zauważmy, że żadna liczba ze zbioru $D(n)$ nie może zostać połączona w żadną parę, tym samym góre ograniczenie na liczbę utworzonych par wynosi $\lfloor \frac{n-|D(n)|}{2} \rfloor$.

Pokażemy, że powyższe ograniczenie jest osiągalne. W tym celu, musimy połączyć w pary wszystkie liczby z $R(n) = \{1, \cdots, n\} \setminus D(n)$ z wyjątkiem co najwyżej jednej (jeśli rozmiar $R(n)$ jest nieparzysty).

Niech $f(x)$ dla $x \geq 2$ oznacza największy dzielnik pierwszy liczby x. Podzielmy liczby z $R(n)$ na grupy ze względu na wartość $f(x)$.

Zauważmy, że dla każdej części takiego podziału, jeśli odpowiada liczbom dającym pewną ustaloną wartość $f(x)$, to w tej części jest również liczba $2f(x)$. Jest tak dlatego, że $f(x)$ jest pierwsze, a więc największym dzielnikiem pierwszym liczby $2f(x)$ jest $f(x)$, do tego $2f(x) \leq n$ (co wynika z definicji $R(n)$).

Każde dwie liczby z danej części podziału mają wspólny dzielnik $f(x)$, a więc mogą zostać one sparowane ze sobą w dowolny sposób. Tym samym, każdą część podziału parzystego rozmiaru możemy połączyć w pary w całości, a w częściach nieparzystego rozmiaru łączyć tak, żeby niesparowana pozostała nam liczba $2f(x)$. Z niektórych części pozostaną tam liczby $2f(x)$, wszystkie te liczby są parzyste, a więc i je możemy połączyć w pary dowolnie. W ten sposób połączymy w pary wszystkie liczby z $R(n)$ z wyjatkiem

co najwyżej jednej, co kończy dowód naszego lematu.

Do zakończenia zadania pozostaje obliczyć wartość $|D(n)|$, do tego zaś wystarcza umieć wyliczać wartość $\pi(n)$ (tj. ilość liczb pierwszych mniejszych lub równych n). Istnieje (klepalny, aczkolwiek skomplikowany) algorytm który wykonuje to w złozoności około $\mathcal{O}(n^{\frac{2}{3}})$ (patrz omówienie zadania https://codeforces.com/contest/665/problem/F) - na szczęście, nie jest on tutaj wymagany, a oczekiwane rozwiązanie jest o wiele prostsze.

By rozwiązać zadanie, zauważmy najpierw, że po preprocessingu w czasie $\mathcal{O}(\sqrt{n}\log\log n)$, ilość liczb pierwszych na przedziale $[l, r]$ dla $1 \leq l \leq r \leq n$ możemy wyznaczać w czasie $\mathcal{O}((r-l)\log\log(r-l) + \sqrt{r})$. Istotnie, wystarczy że najpierw (w trakcie preprocessingu) wyznaczymy używając algorytmu sita Erastotenesa wszystkie liczby pierwsze nie przekraczające $\sqrt{n}$, żeby potem dla przedziału $[l, r]$ móc wykonywać lekko zmodyfikowane sito Erastotenesa: dla każdej liczby pierwszej p nie większej niż $\sqrt{r}$, wyznaczamy pierwszą liczbę w przedziale $[l, r]$ podzielną przez p, i skacząc o p odznaczamy wszystkie liczby podzielne przez p z tego przedziału. Nieoznaczone na koniec liczby to interesujące nas liczby pierwsze.

W naszym zadaniu $n \leq 10^{11}$, a więc stać nas na wymagany czas preprocessingu. Następnie, dla każdego $0 \leq d < 10^4$ wyliczamy ilość liczb pierwszych na przedziale $[d \cdot 10^7, (d+1) \cdot 10^7 - 1]$. Wartości te wyznaczamy opisanym algorytmem sita na przedziale. Obliczenia przeprowadzamy lokalnie na swojej maszynie (czas potrzebny na to jest zależny od szczegółów implementacyjnych ale nie powinien wynieść więcej niż kilka minut).

Obliczone wartości wklejamy do kodu naszego finalnego rozwiązania. Teraz gdy potrzebujemy wyliczyć $\pi(n)$, mając policzone ilości w blokach, pozostaje nam do doliczenia przedział długości nie większej niż 10^7, co już nasze rozwiązanie wykonuje podczas działania na sprawdzaczce (ponownie używając algorytmu sita na przedziale).

Finalnie, nasze rozwiązanie wymaga czasu $\mathcal{O}(n \log\log n)$ lokalnie, zaś wysłane rozwiązanie wykonuje się na sprawdzaczce w czasie $\mathcal{O}(\frac{n}{b}\log\log n + \sqrt{n})$, gdzie b to liczba bloków (w tym wypadku 10^4) dla których policzyliśmy odpowiedź z góry. Im większa wartość b, tym nasze finalne rozwiązanie wykona się szybciej, ale wartość b jest ograniczona przez maksymalny rozmiar kodu źródłowego (podczas UZI było to $100KB$).

G: Vladik

Vladikowi uda się znaleźć cztery długości tworzące prostokąt jeśli zachodzi co najmniej jeden z warunków

- istnieją co najmniej dwie wartości takie, że każda z nich występuje co najmniej dwa razy
- istnieje wartość, która występuje co najmniej cztery razy

Warunki te możemy prosto sprawdzić jeśli zliczymy najpierw dla każdej liczby ze zbioru $\{1, 2, \cdots, 10^5\}$ ile razy występuje ona na wejściu.

Złożoność: $\mathcal{O}(n + C)$ gdzie C to ograniczenie górne na długości patyczków Vladika.