Rozwiązania zadań z Mistrzostw Polski Szkół Średnich w Programowaniu Zespołowym 2024

Zadanie A

W zadaniu dane mamy wyniki dwóch meczów pomiędzy dwoma drużynami, gdzie pierwszy mecz był rozgrywany na stadionie pierwszej drużyny, a drugi na stadionie drugiej drużyny. Zadanie polega na określeniu która drużyna wygrała ten dwumecz, zakładając że w przypadku remisu wygrywa drużyna, która ma więcej strzelonych bramek na wyjeździe.

Rozwiązanie tego zadania to nic innego jak sprawdzenie która drużyna strzeliła sumarycznie więcej goli, a w przypadku równej ilości bramek, sprawdzenie która miała więcej bramek strzelonych na wyjeździe.

Zadanie B

W zadaniu mamy dane N punktów $p_1, \ldots, p_N$ na osi OX, wszystkie z przedziału $[0, D]$. Każdy z tych punktów ma określony promień zasięgu r_i (w lewo i prawo) oraz koszt c_i, który możemy zapłacić aby zwiekszyć ten promień o 1 (można tę akcję wykonać kilkukrotnie). Naszym celem jest zwiększenie promieni w taki sposób, żeby żadne zasięgi dwóch punktów nie przecinały się i żeby pokryły one co najmniej cały przedział $[0, D]$ (mogą pokrywać szerszy obszar), a jeśli jest kilka rozwiązań, to aby koszt był najmniejszy.

Głównym spostrzeżeniem, którego trzeba dokonać w tym zadaniu jest to, że jeżeli ustalimy już promień jednego z punktów, to okazuje się, że promień każdego kolejnego musi mieć już konkretną, zależną od niego wartość. Zatem wystarczy nam ustalić jaki będzie promień pierwszego punktu, a następnie policzyć czy wtedy pozostałe promienie będą poprawne i jaki osiągniemy wtedy koszt.

Złożoność powyższego rozwiązania naturalnie możemy oszacować jako $O(N \cdot D)$, gdyż pierwszy promień może mieć co najwyżej D różnych wartości, a dla każdej z nich mamy do sprawdzenia N punktów. Jednakże, jeśli napiszemy to rozwiązanie sensownie (czyli będziemy przerywać daną iterację w momencie, gdy wiemy, że jest zła), okazuje się, że możemy również oszacować to rozwiązanie jako $O(N + D) = O(D)$! Aby to uzasadnić, weźmy pewną iterację oraz przyjrzyjmy się *punktom styku* sąsiednich promieni. Jak się dokładniej przyjrzymy, to zdefiniowanie jednego takiego punktu jednoznacznie definiuje nam pozycje pozostałych. Oznacza to, że każdy taki punkt może wystąpić tylko w jednej iteracji pętli, a punktów takich jest $O(D)$. To rozwiązanie oczywiście mieściło się w limitach.

Na koniec możemy zauważyć, że w zadaniu istnieje także rozwiązanie o złożoności $O(N)$, wystarczy znaleźć *zakres dobrych promieni* dla pierwszego punktu. Aby to zrobić, należy sprawdzić każdy kolejny punkt i zweryfikować, czy nie wpłynie on na zmniejszenie naszego zakresu (dla pierwszego punktu), a jeśli tak, to odpowiednio zmniejszyć ten zakres. Mając dany ten zakres

możemy zauważyć, że zwiększenie promienia o 1 dodaje nam do sumarycznego kosztu wszystkie c_i dla nieparzystych wartości i, a odejmuje te z parzystych pozycji. Dlatego też rozwiązaniem będzie jeden ze skrajnych punktów tego zakresu.

Zadanie C

Treść zadania

W zadaniu otrzymujemy liczbę początkową S, końcową T, górny limit M, oraz ciąg znaków długości N, składający się z `W`, `P` i `?`. Rozpoczynamy z liczbą S i patrzymy na kolejne znaki w ciągu. W przypadku `W`, podwajamy aktualną wartość, a w przypadku `P` dzielimy ją bez reszty przez 2. Warto od razu pomyśleć o bitowej reprezentacji naszej liczby – wtedy operacje są równoważne usunięciu ostatniego bitu i dopisaniu na końcu 0. Jeśli natrafimy na `?` to musimy wybrać jedną z tych operacji. Na koniec takiego procesu chcemy otrzymać liczbę T, a także nigdy nie przekroczyć M.

Uproszczony problem

Zignorujmy na razie górny limit. Żeby nasze zadanie było w ogóle możliwe do wykonania, S i T muszą mieć jakiś wspólny prefiks (kilka najbardziej znaczących bitów musi być taka sama). Co więcej, za tym wspólnym prefiksem w T mogą znaleźć się już tylko bity 0.

Możemy więc policzyć, ile bitów musimy **co najmniej** i **co najwyżej** usunąć z S, żeby móc osiągnąć wartość T. Dla przykładu, jeśli S to 1010101, a T to 1010, to z S musimy usunąć co najmniej 3 bity, oraz nie możemy usunąć ich więcej niż 4.

Podczas samego procesu wykonywania operacji potrzebujemy trzymać tak naprawdę niewiele informacji:

- pozycję najbardziej znaczącego bitu aktualnej wartości (mamy tylko logarytmicznie wiele możliwości)
- czy usunęliśmy już minimalną liczbę bitów wymaganą do osiągnięcia T (oczywiście 2 możliwości)

Zauważmy, że jeżeli na koniec procesu pozycja najbardziej znaczącego bitu zgadza się z tą w T oraz usunęliśmy wymaganą liczbę bitów z naszej wartości, a podczas wykonywania operacji nigdy nie usunęliśmy więcej bitów niż maksymalna dozwolona liczba, to na sam koniec otrzymaliśmy T. Takie sformułowanie problemu narzuca proste rozwiązanie korzystające z programowania dynamicznego.

Górny limit

Jakie informacje musimy dodatkowo trzymać w przypadku górnego limitu M? Na szczęście niewiele więcej – jeżeli po prostu otrzymalibyśmy limit na pozycję najbardziej znaczącego bitu, to nie musimy pamiętać niczego więcej. Niestety, w zależności od tego, ile usunęliśmy już bitów z początkowej wartości, taki górny limit będzie się zmieniał. Na przykład dla S równego 101 oraz M równego 100000, do S możemy dopisać maksymalnie 2 zera, ale po usunięciu ostatniego bitu będziemy w stanie dopisać już 3.

Zawuażmy, że taki fenomen może wydarzyć się tylko raz, gdy porównując prefiksy reprezentacji bitowych obu liczb, pierwszy różny bit jest zapalony w S i zgaszony w M. Wtedy po usunięciu tego bitu z S limit na pozycję najbardziej znaczącego bitu podniesie się nam o 1.

Potrzebujemy więc w naszym rozwiązaniu trzymać jedną dodatkową informację – czy górny limit na pozycję najbardziej znaczącego bitu nam się podniósł, czy nie. Rozwiązanie zadania wygląda analogicznie do wersji uproszczonej – korzystamy z programowania dynamicznego, gdzie trzymamy $4 \cdot \log_2(\max(S, T, M))$ różnych stanów.

Notka

Istnieje dużo innych rozwiązań o podobnych złożonościach. Korzystając ze strategii zachłannej można nawet uzyskać liniową złożoność obliczeniową.

Zadanie D

Treść zadania

Mamy dane drzewo. Mamy ponumerować jego wierzchołki w taki sposób by żadne dwa połączone krawędzią nie różniły się o 1 (modulo n).

Rozwiązanie

Zauważmy, że zadanie można zinterpretować jako dodanie do grafu cyklu prostego długości n, takiego że nie korzysta on z żadnej krawędzi w oryginalnym grafie (krawędzie tego cyklu odpowiadają parom wierzchołków, które nie mogą być sąsiadami).

Zauważmy, że drzewo jest grafem dwudzielnym (każda ze stron dwudzielności jest względem siebie grafem pustym, więc możemy dowolnie rozkładać cykl prosty odwiedzając kolejno te wierzchołki). Do rozłożenia cyklu wystarczą nam zatem dwie pary wierzchołków po różnych stronach podziału dwudzielności, takie że nie ma między nimi krawędzi. Pozwoli to na rozpoczęcie cyklu jedną z tych krawędzi, następnie odwiedzenie wszystkich wierzchołków jednej części podziału (kończąc na wierzchołku będącym końcem drugiej wybranej krawędzi), powrót drugą krawędzią na drugą stronę i domknięcie cyklu.

Łatwo zauważyć że warunek dwóch krawędzi jest też konieczny.

Jak znaleźć takie dwie krawędzie o różnych końcach niewystępujące na wejściu? Na początku przyjmijmy, że każda ze stron dwudzielności ma przynajmniej 3 wierzchołki. W takim wypadku znajdźmy dowolną krawędź nieobecną na wejściu i zauważmy, że pozostałe wierzchołki nie mogą tworzyć względem siebie dwudzielnej kliki (istniałby cykl, a graf jest drzewem), więc wśród nich jest jakaś krawędź, nieobecna na wejściu.

Jeżeli jedna ze stron ma dokładnie dwa wierzchołki, to zauważmy, że oba te wierzchołki będą końcami szukanych krawędzi, możemy więc pozwolić sobie na wylistowanie dla każdego z nich potencjalnych końców po drugiej stronie. Dla każdego z nich, jeżeli nie jest jedynym elementem drugiego zbioru, to potrafimy znaleźć rozwiązanie.

UWAGA Istnieje zrandomizowana wersja tego rozwiązania, ale nie wydaje się ono prostsze od powyższego i posiada parę miejsc, gdzie mogą pojawić się błędy w analizie prawdopodobieństwa na sukces

Rozwiązanie alternatywne

Dla wejściowego grafu wyznaczmy centroid (wierzchołek taki, że gdy usuniemy go z drzewa, żadna spójna nie będzie miała rozmiaru większego niż n/2). Przeglądajmy teraz kolejno poddrzewa po uko-

rzenieniu w centroidzie (od największych do najmniejszych), przypisując wierzchołkom numery 1, 3, ... (n albo n - 1), a następnie (n-1 albo n), ..., 6, 4, 2, za każdym razem wybierając dzrzewo, przypisując mu numery z mniejszego zbioru (parzyte albo nieparzyste), który został. Może się okazać, że na końcu zostanie nam jakieś poddrzewo (+ centroid), którym trzeba będzie przypisać zarówno parzyste jak i nieparzyste numery, ale zauważmy, że zużyliśmy przynajmniej połowę parzystych i przynajmniej połowę nieparzystych, a ponieważ używaliśmy ich od innych stron, to ich wartości się "rozjeżdżają". Problem w tym, że na końcu drzewo może mieć na przykład rozmiar 1+centroid. Żeby istniało rozwiązanie wystarczą dwa drzewa głębokości przynajmniej 2, ale rozważenie tego i paru innych przypadków pozostawiamy czytelnikowi jako ćwiczenie.

Zadanie E

Rozważmy zbiór S_d pól, których odległość do końca jest co najwyżej d (odległość — minimalny koszt dotarcia do ostatniej kolumny, oznaczany przez $dist$). Zauważmy, że w dowolnym wierszu odległość do końca jest funkcją malejącą. Wynika to z tego, że jeśli weźmiemy optymalną ścieżkę (x_i, y_i) dla pola (x_0, y_0) oraz pole (x, y_0) dla $x_0 < x$ to dla pewnego i mamy $x_i = x + 1$ to koszt dojścia z (x, y_0) do (x_i, y_i) to dokładnie $|y_i - y_0|$. Łatwo zauważyć, że jest to też dolne szacowanie tego kosztu dla pola (x_0, y_0). Kopiując resztę ścieżki dostajemy, że $dist(x_0, y_0) \geq dist(x, y_0)$. S_0 łatwo policzyć. Będziemy chcieli znaleźć wszystkie S_d.

Zauważmy, że nie opłaca nam się skakać do następnej kolumny na pole, które jest dalej niż w sąsiednim wierszu — gdyby taki ciąg ruchów był optymalny to równie dobrze można by zmienić współrzędną y o 1 lub jej nie zmieniać (któreś z tych dwóch pól jest wolne) a potem powtórzyć dalszy ciąg ruchów.

Ustalmy rząd $1 < y < n$. Niech $(x_1, y - 1), (x_2, y + 1) \in S_d$ będą takie, że x_1, x_2 są możliwie najmniejsze. Wówczas oczywiście $(min(x_1, x_2) - 1, y) \in S_{d+1}$. Ponadto, jak pokazaliśmy, nie opłaca się skakać o więcej niż jeden wiersz, zatem jeśli oznaczymy przez x najmniejszą liczbę taką, że pomiędzy (x, y) a $(min(x_1, x_2) - 1, y)$ nie ma przeszkód to $(x, y) \in S_{d+1}$ oraz jest to minimalny x o tej własności (należy oddzielnie uwzględnić przypadek, gdy to pole ma odległość co najwyżej d). Zatem mając S_d możemy liniowo policzyć S_{d+1}. Oczywiście interesują nas tylko te wiersze y, dla których $(0, y) \notin S_d$. Zatem łącznie policzenie wszyskich istotnych S_d (i odpowiedzenie w międzyczasie na pytania — na każde można odpowiedzieć po posortowaniu offline w czasie stałym kiedy będziemy je mijać podczas zamiatania) zajmie nam tyle, ile wynosi $\sum_i^n dist(1, n)$.

Złożoność czasowa Załóżmy, że $m = n$. Chcemy oszacować sumę odległości do ostatniej kolumny pól z pierwszej kolumny. Szacowanie górne będzie korzystać z następujących ścieżek:

- idziemy do najbliższej liczby pierwszej, która jest brana w co najwyżej Δ wierszach
- dla tej liczby pierwszej idziemy do końca

dla pewnej stałej Δ dobranej później.

W drugim kroku oczywiście zapłacimy łącznie co najwyżej $n\Delta$. Oszacujmy zatem koszt pierwszego. Mamy co najwyżej $\frac{n}{\Delta}$ liczb pierwszych, które są złe (tzn. występują częściej niż Δ razy). Niech p_i będzie i-tą liczbą pierwszą. Oznaczmy $G := \max_{p_{i+1} \leq n}(p_{i+1} - p_i)$. Między każdą parą dobrych liczb pierwszych p, q płacimy łącznie $\frac{(p-q)^2}{2}$ na dojście dla przedziału $[p, q]$ na jego brzegi. Rozbijmy koszt na dwa przypadki, załóżmy że p_i oraz p_j są dobre a $p_{i+1} \ldots p_{j-1}$ są złe. Wówczas zapłacimy co najwyżej:

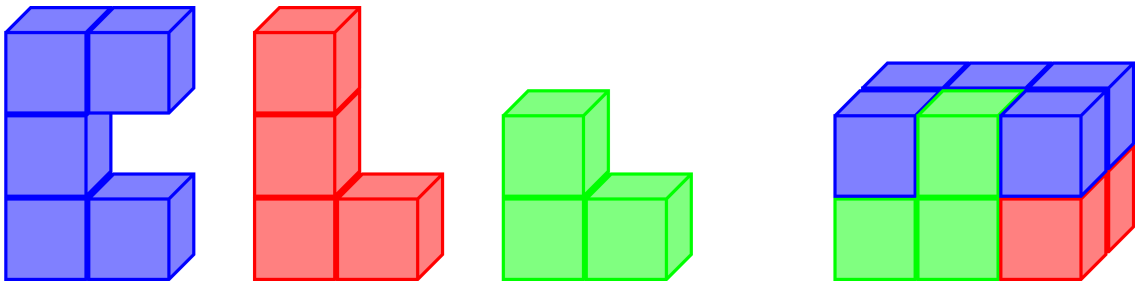
- gdy $i+1 < j$ to dla pojedynczej pary (i,j) płacimy $\frac{(p_j-p_i)^2}{2} \leq \frac{(G(j-i))^2}{2}$. Ale ponieważ jest co najwyżej $\frac{n}{\Delta}$ złych liczb pierwszych to suma $j-i-1$ w tym przypadku to co najwyżej $\frac{n}{\Delta}$, zatem można oszacować $\sum_{(i,j)} \frac{(G(j-i))^2}{2} = G^2 \sum_{(i,j)} \frac{(j-i)^2}{2} = G^2 \sum_{(i,j)} \frac{(j-i-1)^2}{2} + (j-i-1) + \frac{1}{2} \leq G^2 \sum_{(i,j)} \frac{(j-i-1)^2}{2} + 2(j-i-1) \leq G^2(\frac{2n}{\Delta} + \sum_{(i,j)} \frac{(j-i-1)^2}{2}) \leq G^2(\frac{2n}{\Delta} + \sum_{(i,j)} \frac{n}{\Delta} \frac{(j-i-1)}{2}) = G^2(\frac{2n}{\Delta} + \frac{n}{\Delta} \sum_{(i,j)} \frac{(j-i-1)}{2}) \leq G^2(\frac{2n}{\Delta} + \frac{n}{\Delta} \frac{n}{2\Delta}) = G^2 \frac{2n}{\Delta} + G^2 \frac{n^2}{2\Delta^2}$. Dominujący jest ostatni składnik.
- gdy $i+1 = j$ to łącznie co najwyżej $\sum_i \frac{(p_{i+1}-p_i)^2}{2} \leq \sum_i \frac{G(p_{i+1}-p_i)}{2} = G \sum_i \frac{(p_{i+1}-p_i)}{2} = \frac{Gn}{2}$.

Zatem interesuje nas dobranie takiej Δ, że $n\Delta + G^2 \frac{n^2}{2\Delta^2}$ jest jak najmniejsze. Zatem $n\Delta = G^2 \frac{n^2}{\Delta^2}$ i $\Delta = (nG^2)^{\frac{1}{3}}$ czyli łącznie mamy $\frac{3}{2} n^{\frac{4}{3}} G^{\frac{2}{3}}$. Wstawiając tam teoretyczne szacowania na G (`https://en.wikipedia.org/wiki/Prime_gap`) czyli $G = \mathcal{O}\left(n^{0.525}\right)$ dostajemy $\mathcal{O}\left(n^{1.68}\right)$. Wydawałoby się, że to dużo, jednak możemy powyższe szacowania przeprowadzić numerycznie.

Dla $n = 100\,000$ mamy $G = 72$. Ponieważ powyższe szacowania nie mają nieznanych stałych, możemy podstawić dane by uzyskać, że suma odległości to co najwyżej $1.2 \cdot 10^8$. Ale możemy przeszacować lepiej. Powyżej zakładamy, że możemy mieć $\frac{n}{\Delta}$ kolejnych liczb pierwszych z odległością G między każdą kolejną parą - należałoby się spodziewać, że liczby pierwsze występują gęściej. Przyjmijmy $\Delta = 200$. Przeszacujmy jeszcze raz $\sum_{(i,j)} \frac{(p_j-p_i)^2}{2} = \sum_{(i,j)} (j-i-1) \frac{(p_j-p_i)^2}{2(j-i-1)} \leq \frac{n}{2\Delta} \max \frac{(p_j-p_i)^2}{j-i-1}$. Dla danej wartości Δ mamy $\max \frac{(p_j-p_i)^2}{j-i-1} \approx 70\,000$, zatem łącznie da to oszacowanie na liczbę operacji około $3.7 \cdot 10^7$. Gdybyśmy zamiast szli do liczby pierwszej, szli do dowolnej liczby, która ma co najwyżej Δ przeszkód, numerycznie dałoby się oszacować jeszcze lepiej.

Zadanie F

W zadaniu mamy daną figurę składającą się z sześcianów oraz kilka klocków również składających się z takich sześcianów. Sumarycznie klocki mogą się składać maksymalnie z $M \leq 22$ takich sześcianów. Celem zadania jest określenie czy z tych klocków da się ułożyć podaną figurę, a jeśli tak, to opisanie który klocek będzie gdzie.



Przykładowe klocki oraz ułożoną z nich figurę można zobaczyć na powyższym rysunku.

Jak się łatwo domyślić, jedną z trudności zadania jest implementacja obracania klocków wokół osi OX, OY oraz OZ. Obroty takie implementuje się bardzo podobnie do obrotów wokół środka układy współrzędnych na płaszczyźnie w dwóch wymiarach. Obrót taki można wykonać na przykład zamieniając wszystkie współrzędne (x,y) na $(y,-x)$. W trzech wymiarach, jeśli chcemy wykonać na przykład obrót wokół osi OZ, możemy zamienić współrzędne każdego punktu (x,y,z) na $(y,-x,z)$.

Wszystkich możliwych obrotów każdego klocka jest do 24. Aby je łatwo wygenerować, możemy na przykład wygenerować wszystkie obroty klocka wokół osi OX, każdy z nich obrócić na 4 sposoby wokół osi OY, no i każdy z tych klocków obrócić jeszcze na wszystkie sposoby wokół osi OZ. Tych obrotów uzyskamy łącznie 64, dlatego musimy wyrzucić powtórzenia. W tym celu przydaje się normalizacja, czyli przesunięcie klocka w układzie współrzędnych na przykład w taki sposób, żeby sześcian o najmniejszych (leksykograficznie) współrzędnych miał współrzędne $(0, 0, 0)$. W ten sposób w naszym zestawie jeden klocek nie będzie wystąpował w dwóch możliwych przesunięciach względem środka układu współrzędnych.

Teraz moglibyśmy rozwiazać zadanie używając *backtrackingu* — moglibyśmy dla każdego sześcianu figury początkowej spróbować przyłożyć w nim każdy klocek, obrócony w każdy możliwy sposób. Niestety, takie rozwiązanie w niektórych przypadkach może być wolne i trudno jest je bezpośrednio przyspieszyć.

Tutaj z pomocą przychodzą nam maski bitowe. Ponumerujmy wszystkie sześciany docelowej figury numerami od 1 do M. Teraz możemy spróbować przyłożyć każdy obrót, każdego klocka, w każdym miejscu do naszej figury, i jeśli pasuje, to możemy doliczyć maskę bitową tego dopasowania do puli pasujących masek dla tego klocka.

Nasze rozwiązanie teraz będzie miało formę backrtackingu, ale z zapamiętywaniem odwiedzonych stanów. A mianowicie, dla maski bitowej *niepokrytych sześcianów*, oraz dla kolejnych numerów klocków będziemy próbować wszystkich dopasowań tego klocka do tych sześcianów, a jeżeli któreś będzie pasowało, to wywołamy się rekurencyjnie. To rozwiązanie jest rozwiązaniem wzorcowym.

Nasuwa się jeszcze pytanie, jaka jest złożoność backtrackingu z ostatniej fazy. Na pierwszy rzut oka wygląda, że dla $2^M \cdot M$ stanów sprawdzamy wszystkie możliwe przyłożenia maski bitowej danego klocka, co brzmi na bardzo dużą liczbę. Jednakże jest dużo lepiej, gdyż możemy na przykład zauważyć, że po dołożeniu kilku klocków o sumarycznie k sześcianach, dla kolejnego klocka będziemy już rozważać tylko maski bitowe, które mają $M - k$ bitów. Zatem liczba rozpatrywanych stanów będzie wynosić co najwyżej 2^M, a im większe będą klocki, tym więcej będą miały dobrych przypasowań, ale liczba masek bitowych do rozpatrzenia będzie się zmniejszać.

Zadanie G

W zadaniu mamy dane n ofert pracy, każda oferta pracy jest ważna od chwili s_i do chwili e_i, jej wartość po zaakceptowaniu wynosi v_i, ale pojawia się ona z pewnym prawdopodobieństwem p_i. Możemy zaakceptować tylko jedną ofertę, jaka jest wartość oczekiwana uzyskanej oferty przy optymalnej strategii akceptacji ofert?

Zacznijmy od nieefektywnego rozwiązania za pomocą programowania dynamicznego. Oznaczmy poprzez $dp[t][S]$ wartość oczekiwaną optymalnej strategii wybierania ofert, jeśli nie rozważyliśmy jeszcze przedziałów o $s_i >= t$, a zbiór S oznacza istniejące przedziały spełniające $s_i \leq t \leq e_i$.

Takie programowanie dynamiczne można w prosty sposób liczyć:

- Jeśli w chwili t dodajemy nową ofertę r, to $dp[t][S] = p_r \cdot dp[t+1][S \cup \{r\}] + (1 - p_r) \cdot dp[t+1][S]$;
- Jeśli w chwili t kończy się oferta $r \in S$, to $dp[t][S] = \max\{dp[t+1][S \setminus \{r\}], v_r\}$;
- Jeśli w chwili t kończy się oferta $r \notin S$, to $dp[t][S] = dp[t+1][S]$.

Kluczowa obserwacja jest taka, że wystarczy rozważać zbiory S spełniające $|S| \leq 1$. Formalniej zapisując, dla dowolnego zbioru $|S| \geq 2$, istnieje element $s \in S$, że $dp[t][s] = dp[t][S]$.

Dowód tej obserwacji jest indukcyjny względem czasu t. Dla $t = n+n$, nasza teza jest prawdziwa, bo jedyny możliwy zbiór S, to zbiór pusty. Dla $t < n+n$ i ustalonego zbioru S, rozważmy możliwe przejścia w naszej indukcji.

- Dla przejścia $dp[t][S] = p_r \cdot dp[t+1][S \cup \{r\}] + (1-p_r) \cdot dp[t+1][S]$ rozważmy dwa elementy $e_1 \in S \cup \{r\}$, $e_2 \in S$, które spełniają $dp[t+1][e_1] = dp[t+1][S \cup \{r\}]$ oraz $dp[t+1][e_2] = dp[t+1][S]$. Zauważmy, że zachodzi $e_1 = e_2$ lub $e_1 = r$. Korzystając z tej zależności możemy wywnioskować, że $dp[t][e_2] = dp[t][S]$;
- Dla przejścia $dp[t][S] = \max\{dp[t+1][S \setminus \{r\}], v_r\}$, jeśli v_r maksymalizuje wartość, to $dp[t][r] = dp[t][S]$. W przeciwnym przypadku, istnieje element $e \in S \setminus \{r\}$, który spełnia $dp[t][e] = dp[t+1][S \setminus \{r\}]$. Wówczas zachodzi $dp[t][e] = dp[t][S]$;
- Dla ostatniego przejścia, korzystamy z założenia indukcyjnego dla $dp[t+1][S]$ i wprost z niego wynika dowód dla stanu $dp[t][S]$.

Obserwacja ta pozwala nam na optymalizację naszego dynamika do złożoności $\mathcal{O}(n^2)$, co nie jest jeszcze wystarczające do zaakceptowania zadania.

Aby zoptymalizować powyższe rozwiązanie, rozpiszmy przejścia dynamika z $|S| \leq 1$ w nieco inny sposób:

- Jeśli w chwili t dodajemy nową ofertę r, to dla każdej istniejącej oferty $s \neq r$ zapisujemy $dp[t][s] = \max\{dp[t+1][s], (1-p_r) \cdot dp[t+1][s] + p_r \cdot dp[t+1][r]\}$, a stan $dp[t][r]$ usuwamy;
- Jeśli w chwili t usuwamy ofertę r, to dodajemy stan $dp[t][r] = \max\{dp[t+1][0], v_r\}$, a pozostałe stany pozostają bez zmian.

Powyższy wzór jesteśmy w stanie liczyć dość szybko na dowolnej strukturze, która pozwala nam na następujące operację:

- Usuń element o wartości v;
- Dodaj element o wartości v;
- Zmodyfikuj wszystkie elementy o wartości $\leq v$ wzorem $\alpha \cdot v + \beta$. Zauważmy, że ta operacja nie zmienia kolejności sortowania elementów.

Pierwszą strukturą, która przychodzi na myśl, jak widzimy takie operacje, to zbalansowane drzewo binarne, przykładowo treap. W czasie $\mathcal{O}(\log n)$ można wszystkie powyższe operacje rozwiązać na tej strukturze danych.

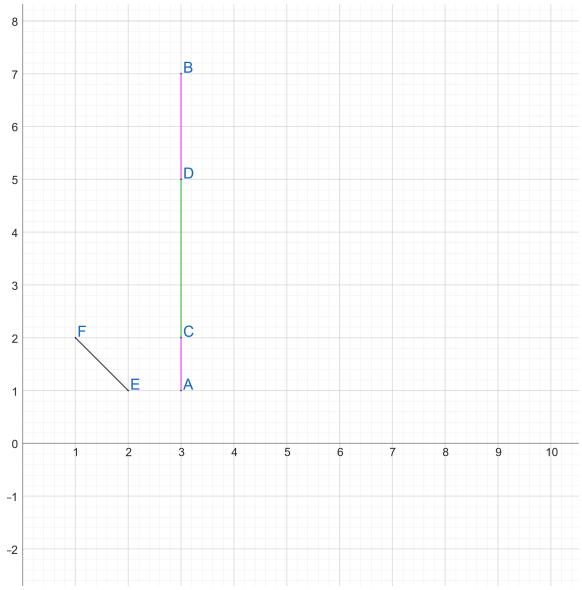
Zadanie to można rozwiązać jednak prościej, za pomocą struktury pierwiastkowej. Idea jest taka, aby utrzymywać aktywne stany, posortowane po wartościach dynamika, w kubełkach o długości pierwiastkowej. Pierwsze dwie operacje są łatwe do wykonania na tej strukturze, więc skupmy się na trzeciej.

Dla każdego kubełka będziemy utrzymywać parę liczb α i β, które oznaczają, że prawdziwa wartość każdego elementu w tym kubełku wynosi $\alpha \cdot v + \beta$. Wówczas, wystarczy przeiterować się po wszystkich kubełkach, które mają wszystkie elementy $\leq v$ z naszego zapytania i odpowiednio zmodyfikować ich wartości α i β. Z kolei w jedynym kubełku w którym mamy wartości mniejsze i większe od v, możemy przeiterować się po wszystkich elementach i je odpowiednio zmodyfikować.

Złożoność takiego rozwiązania, to $\mathcal{O}(\#buckets + \#bucket\_length)$, co przy odpowiednio dobranych parametrach daje nam złożoność $\mathcal{O}(\sqrt{n})$ na zapytanie. Pozwala to na zaakceptowanie danego zadania.

Zadanie H

Zadanie I

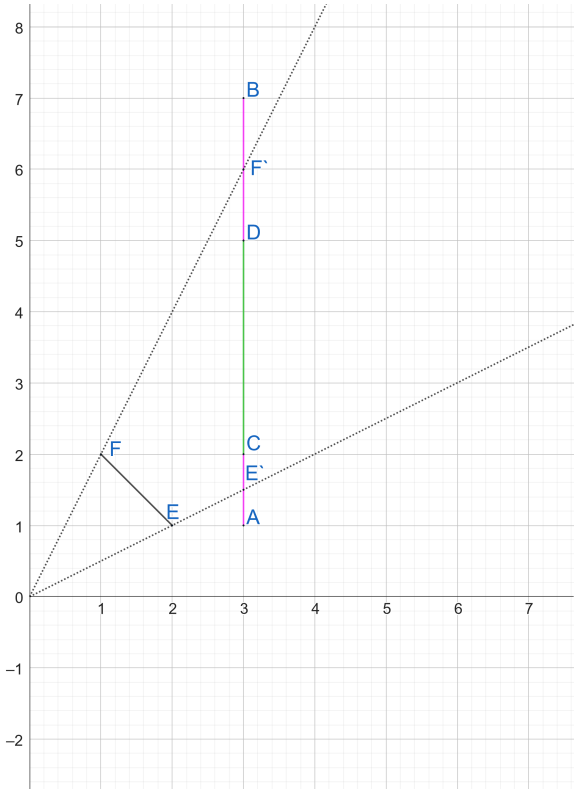


W zadaniu dane są trzy odcinki: AB równoległy do osi OY, CD zawarty wewnątrz AB oraz EF, który znajduje się na lewo od AB oraz na prawo od osi OY. Wszystkie współrzędne są liczbami całkowitymi. Należy sprawdzić, czy istnieje półprosta zaczynająca się w (0, 0) i przecinająca tylko odcinek AB.

Rozwiązanie: Zauważmy, że odcinek CD wydziela z AB dwa odcinki (być może zdegenerowane do punktu): AC oraz DB. Wystarczy więc sprawdzić, czy istnieje półprosta przechodząca przez jeden z nich, ale nie przez EF i nie przez punkty C ani D.

Rozważmy półprostą przechodząca przez AC, drugi przypadek jest analogiczny. Kiedy taka półprosta może istnieć? Wtedy, gdy odcinek EF nie "zasłania" całego odcinka AC.

Żeby to sprawdzić, możemy rzutować odcinek EF na prostą zawierającą odcinek AC i sprawdzić czy punkt E' (rzut punktu E) jest powyżej A lub F' jest poniżej C:



Współrzędną y punktu E' można obliczyć w następujący sposób:

$$E'_y = \frac{E_y}{E_x} \cdot A_x$$

Interesujące nas porównanie punktów E' oraz A ma zatem postać:

$$E'_y > A_y$$

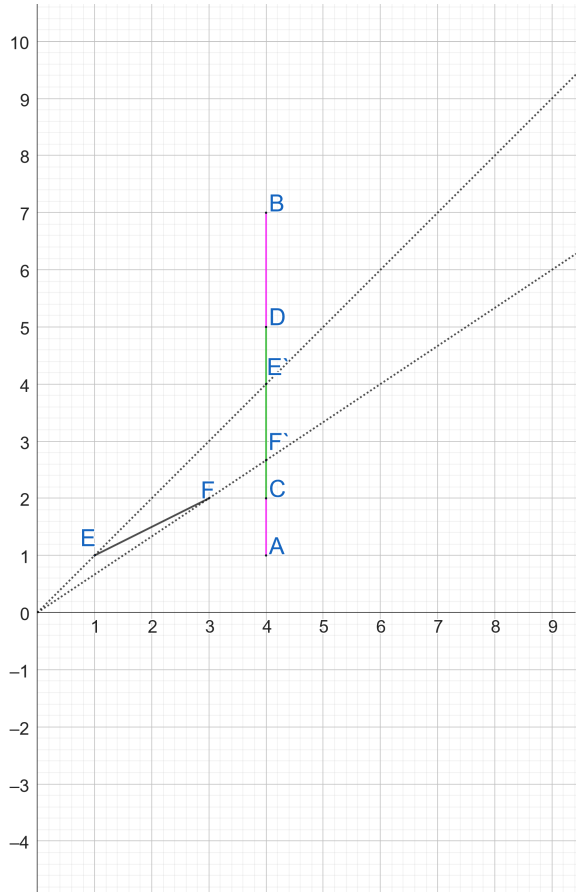
$$\frac{E_y}{E_x} \cdot A_x > A_y$$

$$E_y \cdot A_x > A_y \cdot E_x$$

Porównianie F' z C jest analogicznie

Uwagi:

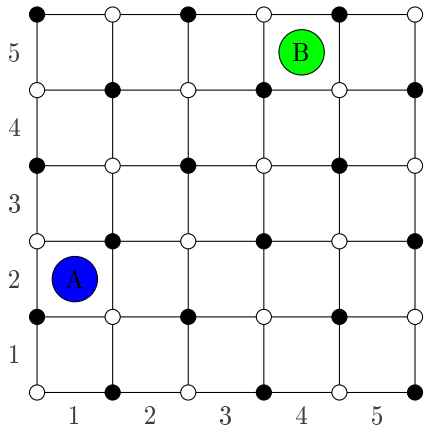
- Po rzucie odcinka EF, punkt F' może być niżej niż E', wtedy trzeba porównać pozycje F' z A oraz E' z C.



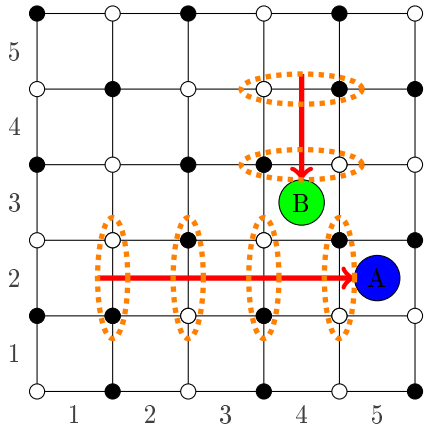
- Trzeba uważać na overflow przy wykonaniu mnożenia (najlepiej użyć większego typu, np. `long long`).
- Jeżeli do obliczeń używasz liczb zmiennoprzecinkowych albo funkcji trygonometrycznych, pamiętaj o korzystaniu z `long double` (i np. `atan2l`), inaczej precyzja będzie niewystarczająca.

Zadanie J

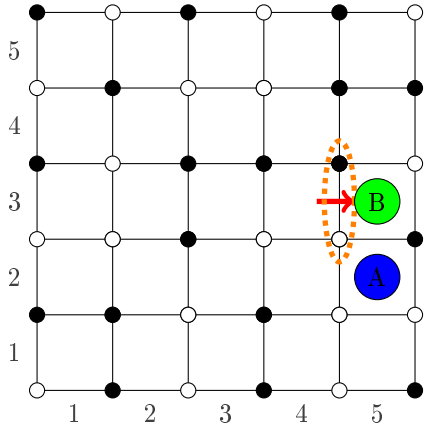
Rozwiązanie istnieje gdy początkowo pionki stoją na polach (x_a, y_a), (x_b, y_b) takich, że nie zachodzi $|x_a - x_b| = |y_a - y_b| = 1$ (to znaczy, gdy pola nie sąsiadują rogami). Istnieje wówczas sekwencja ruchów taka, że pionki znajdą się w jednym polu. Wówczas można ją odwrócić, tylko zamieniając ich role. Jeśli $x_a = x_b$ albo $y_a = y_b$ to rozwiązanie jest proste - wówczas pionki po prostu idą w swoją stronę i każdy ruch w tej osi jest dozwolony, bo między każdą parą kolejnych pól bedą czarne i białe żetony. Załóżmy dalej bez straty ogólności, że $y_b \geq y_a + 2$ oraz $x_a < x_b$:

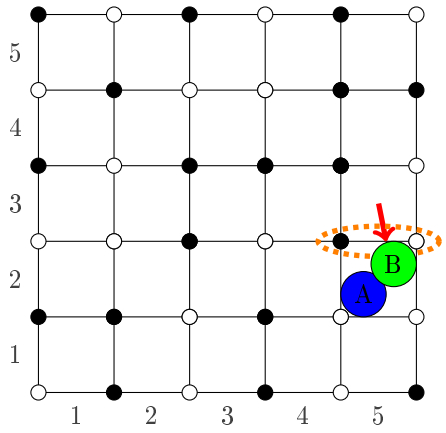


Ustawmy je tak jak na rysunku poniżej (tzn. żeby $x'_a + 1 = x'_b$ oraz $y'_a = 1 + y'_b$):



Następnie zróbmy dwa ruchy pionkiem B:





Potem odwracamy ruchy - ruszamy A do góry, potem A w lewo itd.

Co z $dist(A, B) = \sqrt{2}$?

Zadanie K

W zadaniu mamy dane drzewo o n wierzchołkach przy czym i-ty z nich jest etykietowany liczbą całkowitą d_i. Dla każdego wierzchołka v należy obliczyć iloczyn funkcji J obliczonej dla każdego poddrzewa powstałego z usunięcia wierzchołka v z grafu:

$$J = \sum_{t=1}^{\infty} t \cdot W_{\mathrm{cnt}_t}$$

Gdzie współczynniki W są dane (przy czym $W_0 = 0$), a cnt_t to liczba etykiet w danym poddrzewie, które są wielokrotnościami t.

Załóżmy, że mamy obliczony współczynnik J pewnego poddrzewa i rozważmy jak wygląda jego aktualizacja jeśli dodalibyśmy wierzchołek o etykiecie d. Gdybyśmy znali wartości cnt_t dla wszystkich dzielników etykiet z poddrzewa, moglibyśmy zaktualizować wartość J iterując po wszystkich dzielnikach d:

$$J' = J + \sigma_{t|d} t \cdot W_{cnt_t+1} - t \cdot W_{cnt_t}$$

Wartości cnt_t możemy utrzymywać wprost, ponieważ interesują nas tylko $t \leq n$. W ten sposób możemy uzyskać strukturę danych, która utrzymuje wartość J wymagając $\mathcal{O}(n)$ pamięci i pozwala dodać oraz usunąć (w analogiczny sposób) wierzchołek w czasie liniowym względem liczby dzielnikow jego etykiety. Przy jej użyciu będziemy obliczać wartości J dla poddrzew. Mówiąc dokładniej, przechodząc po drzewie algorytm obliczy współczynnik J dla wszystkich poddrzew w ukorzenionym drzewie w taki sposób, że kazdy wierzchołek drzewa zostanie dodany i usunięty ze struktury $\mathcal{O}(\log n)$ razy. Osiągniemy to za pomocą techniki "mniejszy do większego".

Będziemy przetwarzać wierzchołki używając przeszukiwania w głąb (dfs) zaczynając w dowolnie wybranym korzeniu. Podczas przechodzenia drzewa będziemy utrzymywać niezmiennik, że w mo-

mencie gdy zaczynamy przetwarzanie wierzchołka, struktura jest pusta, a gdy kończymy, zawiera ona dokładnie wierzchołki z jego poddrzewa. Wchodząc do nowego wierzchołka v:

1. Rekursywnie przetwarzamy wszystkie dzieci v z wyjątkiem jednego o największym poddrzewie (y). Po przetworzeniu każdego poddrzewa, przechodzimy je jeszcze raz usuwając wszystkie wierzchołki ze struktury danych.
2. Rekursywnie przetwarzamy dziecko v o największym poddrzewie (y).
3. Dodajemy v do struktury oraz wszystkie wierzchołki z poddrzew dzieci v z pierwszego punktu. Wtedy struktura zawiera wszystkie wierzchołki z poddrzewa v, więc możemy zapisać wartość J dla poddrzewa v.

Powyższa procedura oblicza współczynnik jakości dla podrzew wszystkich wierzchołków. Do uzyskania wyniku będziemy również potrzebować wyników dla ich dopełnień (tzn. dla każdego wierzchołka v musimy jeszcze znać J dla zbioru wszystkich wierzchołków w grafie oprócz tych w poddrzewie v). Te wartości można uzyskać utrzymując siostrzaną strukturę, w której na początku są wszystkie wierzchołki. Za każdym razem, gdy do pierwszej struktury dodajemy wierzchołek, będziemy go usuwać z drugiej i na odwrót: gdy z pierwszej usuwamy, do drugiej dodajemy.

Teraz pozostaje pokazać, że każdy wierzchołek jest dodany lub usunięty ze struktury $\mathcal{O}(\log n)$ razy. Rozważmy moment przetwarzania pewnego wierzchołka v. W jego trakcie do struktury dodajemy i usuwamy po jednym razie wszystkie wierzchołki należące do poddrzew dzieci v z wyjątkiem tego o największym poddrzewie (usuwamy w kroku pierwszym a dodajemy w trzecim). Pozostałe operacje są dokonywane podczas przetwarzania innych wierzchołków. Rozważmy dowolny wierzchołek u dodawany i usuwany ze struktury podczas przetwarzania v. Zauważmy, że:

- v jest przodkiem u w ukorzenionym drzewie.
- Niech x będzie ostatnim przodkiem u przed v: ponieważ x nie jest dzieckiem v o największym poddrzewie (a przynajmniej nie jedynym, ponieważ $y \neq x$), poddrzewo v jest co najmniej dwa razy większe niż poddrzewo x.

Z powyższego wynika, że dla ustalonego u może istnieć co najwyżej $\mathcal{O}(\log n)$ takich v, podczas przetwarzania których u może być dodawane / usuwane ze struktury.

Wobec tego algorytm wykonuje $\mathcal{O}(n \log n)$ operacji na strukturze. Oznaczając przed S sumę liczb dzielików d_i (mamy oszacowanie $S \leq 240n$), otrzymujemy algorytm o złożoności $\mathcal{O}(S \log n)$.

Bonus: rozwiązanie bez loga

Kompresja drzewa do zbioru wierzchołków S to stworzenie drzewa na wierzchołkach $T = \{lca(a, b) : a, b \in S\}$ i dodanie krawędzi, jeśli para różnych wierzchołków z T nie ma na ścieżce między sobą żadnego innego wierzchołka z T. Można pokazać, że $|T| \leq 2|S| - 1$. Więcej materiałów tutaj: Zadanie Kolacje z XXVI OI, omówienie pewnego zadania z Atcodera. Kompresję tę można znaleźć liniowo od $|S|$, wykorzystując algorytm na LCA (najniższego wspólnego przodka) w czasie stałym (Blog na Codeforces) lub, bardziej praktycznie, w $\mathcal{O}(n\alpha(n))$ bazującym na Union-Findzie Algorytmem Tarjana.

Dla każdej krawędzi i każdego d będziemy chcieli wskazać wartość funkcji z zadania dla konkretnego d biorąc pod uwagę wierzchołki które są po tej samej stronie tej krawędzi co jeden z jej końców i na końcu wziąć sumę po d dla każdej krawędzi. Dla każdego d rozważmy drzewo skompresowane do tych wierzchołków, które mają wagi będące wielokrotnościami d. Mając już je, możemy na każdej

ścieżce odpowiadającej krawędzi w skompresowanym drzewie zamieścić informację o tym ile jest w dół i w górę wielokrotności d. Stosując sumy prefiksowe, możemy to zrobić w czasie liniowym od rozmiaru drzewa po kompresji, propagując je dopiero na końcu jak już zajmiemy się wszystkimi d. Zatem łącznie da nam to czas liniowy od sumy liczby dzielników.

Zadanie L

W tym zadaniu mieliśmy dane pięć sytuacji na boisku. Dla każdej z nich dostaliśmy informację ile razy się zdarzyła podczas meczu oraz liczbę sekund, którą arbiter powinien doliczyć do podstawowego czasu gry za każde zajście takiej sytuacji. Pytaniem jest ile minut musi ostatecznie doliczyć arbiter na koniec meczu.

Aby rozwiązać zadanie należało dla każdej z tych sytuacji pomnożyć liczbę sekund z treści razy liczbę zajść danej sytuacji. Następnie należało dodać do siebie te wszystkie wyniki, a na koniec podzielić przez 60.

Ewentualnymi miejscami w których można było tutaj popełnić błąd było zaokrąglenie w górę przy dzieleniu (wystarczyło sprawdzić czy liczba jest podzielna przez 60 i ewentualnie dodać do wyniku dzielenia całkowitego wartość 1), albo można było zapomnieć o użyciu typu `long long` (co prawda wynik końcowy mieścił się w zakresie typu `int`, ale za to wyniki pośrednie niekoniecznie).

Zadanie M

W zadaniu mamy dany zbiór liczb M, z których wszystkie są z przedziału $[0, 2^k - 1]$ (dla $k = 20$) i reprezentują maski bitowe. Następnie mamy dane dużo zapytań składających sie z dwóch liczb a i b, reprezentujących maski z tego samego przedziału. Zadanie polega na znalezieniu dla każdego takiego zapytania maski $m \in M$, która zarazem jest podmaską maski a (czyli każdy bit, który jest zapalony w masce m, musi też być zapalony w masce a), ale też m nie może być podmaską maski b.

Przeanalizujmy najpierw jedno zapytanie. Zauważmy, że jeśli znaleziona przez nas maska m nie może być podmaską maski b, to znaczy, że musi istnieć co najmniej jeden bit, który jest zapalony w masce m, za to nie jest zapalony w masce b. Zatem dla takiego zapytania wystarczy, że sprawdzimy wszystki bity i, które nie są zapalone w masce b, a następnie spróbujemy dla każdego z nich znaleźć maskę z M, która zarazem jest podmaską maski a, ale też ma na pewno zapalony bit i.

Zauważmy teraz, że różnych zapytań powyższego typu moze być co najwyżej $2^{20} \cdot 20 \approx 20\,000\,000$, co nie wydaje się być dużą liczbą ani ze względu na użytą pamięć, ani ze wzgledu na czas działania programu. Zatem możemy napisać program, który dla każdej pary a oraz i wyznaczy pewną maskę ze zbioru M, która zawiera się w a i ma zapalony bit i. Użyjemy do tego celu programowania dynamicznego na maskach bitowych.

Niech $DP[a][i]$ oznacza dowolną maskę z M, która jest podmaską a oraz ma zapalony bit i. Zauważmy, że na początku możemy dla każdej maski $m \in M$ oraz dla każdego zapalonego w niej bitu i ustawić $DP[m][i] = m$. Pozostałe wartości możemu ustawić na przykład na -1. Następnie, dla każdej maski a (rozpoczynając iterację od najmniejszych wartości do największych) i dla każdego bitu i możemy spróbować zgasić w a pewny bit j inny niż i. Jeżeli po zgaszeniu pewnej takiej wartości j maska ma jakieś rozwiązanie, to mozemy przypisać to rozwiązanie do $DP[a][i]$. W przeciwnym przypadku $DP[a][i] = -1$.

Oszacujmy złożoność tego rozwiązania. Przyjmując, że w zadaniu mamy Q zapytań, początkowe liczenie tablicy DP zajmuje $\mathcal{O}\left(2^k \cdot k^2\right)$ czasu. Następnie odpowiedzenie na każde zapytanie zajmuje $\mathcal{O}(k)$ czasu, co oznacza, że całkowity czas działania programu to $\mathcal{O}\left(2^k \cdot k^2 + Q \cdot k\right)$, natomiast pamięć to $\mathcal{O}\left(2^k \cdot k\right)$, co mieściło się w limitach.

Używając operacji bitowych (przy założeniu, że rozmiar słowa maszynowego jest $\geq k$) możemy przyspieszyć powyższy algorytm. Zauważmy że w przejściu przy DP możemy, iterując się po bitach sprawdzać w $\mathcal{O}(1)$ czy podmaska bez danego bitu daje nam jakieś nowe maski z M odpowiadające bitom, których jeszcze nie mamy. Natomiast przy zapytaniu możemy sprawdzić (ANDem) czy przekrój niepustych pól z $DP[a]$ oraz zgaszonych bitów b jest niepusty. Daje nam to złożoność czasową $\mathcal{O}\left(2^k \cdot k + Q\right)$, bez zmiany złożoności pamięciowej.

0.1 Alternatywne rozwiązanie

Zauważmy, że w zapytaniu możemy bez straty ogólności założyć, że b jest podmaską a. Dzieje się tak, bo interesują nas tylko bity, które są zapalone w a.

Oznaczmy przez $sum[i]$ liczbę podmasek i należących do M. Można policzyć $sum[i]$ za pomocą programowania dynamicznego (Sum Over Subset, https://codeforces.com/blog/entry/45223) w czasie $\mathcal{O}\left(2^k \cdot k\right)$ i pamięci $\mathcal{O}\left(2^k\right)$. Za pomocą tablicy sum można łatwo stwierdzić czy maska będąca podmaską a i niebędąca podmaską b istnieje – wystarczy sprawdzić czy $sum[a] > sum[b]$ (przypomnijmy o założeniu, że b jest podmaską a). Pokażemy jak można taką maskę odzyskać, a nawet wylistować t takich masek w $\mathcal{O}(tk)$.

Przeiterujmy się po bitach. Rozważamy i-ty bit, a', b' są podmaskami a, b z usuniętym i-tym bitem. Jezeli $sum[a'] > sum[b']$ to możemy ograniczyć się do szukania dla pary (a', b'). Dodatkowo, konstrukcja ta ma tę własność, że jeśli na końcu powstała para (a, b) to $a \in M$. Jest to dosyć proste do udowodnienia - jeśli istniałaby inna maska spełniająca warunki dla pary (a, b) i różniłaby się na i-tym bicie to rozważając wcześniej i-ty bit, zmienilibyśmy parę na taką, która nie ma go zapalonego, co daje sprzeczność. Można pokazać nawet więcej, jest to najmniejsza maska spełniające warunki z zapytnania. Modyfikując odpowiednio ten algorytm, moglibyśmy pytać o t-tą najmniejszą maskę w $\mathcal{O}(k)$.

Złożoność czasowa tego rozwiązania to $\mathcal{O}\left(2^k \cdot k + Q \cdot k\right)$, natomiast pamięciowa to $\mathcal{O}\left(2^k\right)$.

Zadanie N

W zadaniu jest dane N par liczb, każda z zakresu od 0 do N. Liczba 0 oznacza, że dana pozycja jest wolna, każda inna wartość oznacza, że dana pozycja jest zajęta przez tę liczbę (np. para `0 10` ma pierwszą pozycję wolną, a drugą zajętą przez liczbę 10). Każda liczba od 1 do N pojawi się na wejściu co najwyżej dwa razy. i-ta para jest poprawna, jeżeli na obu pozycjach jest i. Naszym zadaniem jest tak uzupełnić pary, żeby każda liczba od 1 do N pojawiła się dokładnie 2 razy, oraz żeby było jak najwięcej poprawnych par i jedynie wypisać liczbę poprawnych par w optymalnym uzupełnieniu.

Obserwacja: Zauważmy, że wystarczy jedynie policzyć pary, które można uzupełnić poprawnie. Nie musimy się przejmować niepoprawnymi parami – je zawsze da się uzupełnić liczbami, których poprawne miejsce było już zajęte albo jedno z wystąpień było na złej pozycji.

Rozwiązanie: Stwórzmy najpierw tablicę i zapiszmy do niej, jaka wartość znajduje się na każdej pozycji. Następnie obliczmy, ile razy występowała każda z liczb. Możemy to zrobić przy pomocy dodatkowej tablicy (gdzie i-ta pozycja to liczba wystąpień liczby i): przechodzimy po pierwszej tablicy i odpowiednio zwiększamy wartości w drugiej tablicy. Teraz wystarczy zliczyć pary, które można poprawnie uzupełnić. i-ta para może być poprawnie uzupełniona, gdy:

- obie pozycje są od początku poprawne (obie wartości są równe i),
- jedna pozycja to 0, druga to i oraz licznik wystąpień i jest równy jeden (jeżeli licznik byłby równy dwa, to druga liczba i jest już użyta gdzieś indziej),
- obie pozycje są równe 0 i licznik wystąpień i jest równy zero.

Zadanie O

W zadaniu dane są trzy ciągi, każdy złożony z N zer i N jedynek, nazwijmy je A, B i C.
Niech pref_A oznacza ciąg sum prefiksowych stworzony z ciągu A, tzn. $\text{pref}_A[1] = A[1]$, $\text{pref}_A[2] = A[1] + A[2]$, $\text{pref}_A[3] = A[1] + A[2] + A[3]$, itd.

Chcemy sprawić, żeby dla każdego i od 1 do $2 \cdot N$ zachodziło $\text{pref}_A[i] \leq \text{pref}_B[i] \leq \text{pref}_C[i]$. W tym celu możemy zamienić sąsiednie elementy w dowolnym z ciągów. Mamy wypisać, ile minimalnie trzeba wykonać takich operacji, żeby zachodziła powyższa zależność.

Obserwacja 1.: Niech $\text{pos}_A[i]$ oznacza pozycję i-tej jedynki w ciągu A. Zauważmy, że wymagana właściwość:

$$W_1: \text{Dla każdego } i \text{ od 1 do } 2 \cdot N: \text{pref}_A[i] \leq \text{pref}_B[i] \leq \text{pref}_C[i]$$

Zachodzi wtedy i tylko wtedy, gdy zachodzi:

$$W_2: \text{Dla każdego } i \text{ od 1 do } N: \text{pos}_A[i] \geq \text{pos}_B[i] \geq \text{pos}_C[i]$$

Skrótowy dowód, przy ograniczeniu do tylko dwóch ciągów, A i B:

- Jeżeli zachodzi W_2 to w oczywisty sposób zachodzi W_1: skoro w ciągu A każda jedynka jest na tej samej lub późniejszej pozycji co w ciągu B oraz liczba jedynek jest równa, to sumy prefiksowe będą mniejsze lub równe.
- W drugą stronę skorzystamy z dowodu nie wprost. Załóżmy, że zachodzi W_1, ale nie zachodzi W_2. Zatem istnieje pozycja j taka, że $pos_A[j] < pos_B[j]$, czyli j-ta jedynka występuje wcześniej w ciągu A niż w B. Weźmy najmniejsze takie j i niech $p = pos_A[j]$. Zatem j-ta jedynka występuje w ciągu A na pozycji p, ale w ciągu B j-ta jedynka występuje później. Zatem $pref_A[p] = j$ i $pref_B[p] < j$. Czyli $pref_A[p] > pref_B[p]$. Zachodzi sprzeczność z założeniem, zatem jeżeli zachodzi W_1 to zachodzi też W_2

Obserwacja 2.: Ustalmy i, i popatrzmy na zamiany i-tych jedynek w każdym z trzech ciągów w jakimś optymalnym rozwiązaniu. Zauważmy, że jeżeli zamieniliśmy i-tą jedynkę w ciągu B w lewo, to zamiast tego możemy zamienić i-tą jedynkę w ciągu A w prawo, żeby zachować W_2. Analogicznie, zamiast zamiany w prawo w B, możemy zamienić w lewo w C. Powtarzając tą operację, otrzymamy rozwiązanie, w którym jedynki w ciągu B nie zmieniają swoich pozycji.

Rozwiązanie: Wystarczy zatem zsumować dla każdego i:

$$\max(0, pos_B[i] - pos_A[i]) + \max(0, pos_C[i] - pos_B[i])$$

Taka liczba zamian jest wystarczająca: każda jedynka z ciągu A oraz C, która była ustawiona niepoprawnie zrówna się pozycją z jedynką z B. Nie będzie też problemu, że zamieniając jakąś jedynkę, zmienimy pozycję innej: jedynki w ciągu B są ustawione w kolejności, a zamieniając jedynki w ciągu A oraz C, tylko je dorównujemy do pozycji jedynki w B. Jedynki w ciągu A można zamienić idąc od lewej strony (bo zawsze przesuwamy je w prawo), a jedynki w ciągu C idąc od prawej (bo przesuwamy je w lewo).

Zauważmy też, że rozwiązanie korzystające z mniejszej liczby jedynek nie będzie poprawne: w takim rozwiązaniu zostanie jakaś jedynka, która wciąż będzie na złej pozycji.