POTYCZKI ALGORYTMICZNE

Rozwiązania zadań

Rundy zdalne 2021

1C – Koszulki

autor zadania: Mateusz Radecki

Treść

W zadaniu mamy dany ciąg liczb $a_1, a_2, \ldots, a_n$ oraz liczbę k. Jesteśmy proszeni o znalezienie jak najmniejszego podciągu tego ciągu, o mocy co najmniej k, takiego, że dla każdych i oraz j jeśli $a_i \geq a_j$ oraz j-ty element należy do wybranego podciągu, to a_i również do niego należy.

Rozwiązanie

Posortujmy ciąg $a_1, a_2, \ldots, a_n$ nierosnąco. Zastanówmy się kiedy jakiś jego podciąg jest poprawny. Na pewno jeśli jakiś element do niego należy, to wszystkie na lewo od niego też muszą do niego należeć. Sugeruje to, że tylko niektóre prefiksy są poprawne. Dodatkowo, aby prefiks był poprawny, to albo musi on być całym ciągiem, albo jego ostatni element musi być ściśle większy niż pierwszy element zaraz za nim.

Zatem tak długo jak $k \neq n$ oraz $a_k = a_{k+1}$ musimy zwiększać k o 1. Ostateczna wartość k jest naszym wynikiem. Złożoność całego algorytmu, ze względu na sortowanie, to $\mathcal{O}(n \cdot \log(n))$.

1B – Oranżada

autor zadania: Mateusz Radecki

Treść

W zadaniu mamy dany ciąg liczb $a_1, a_2, \ldots, a_n$ oraz liczbę k. W jednym ruchu możemy zamienić dwa sąsiednie elementy ciągu miejscami. Jesteśmy pytanie ile co najmniej ruchów potrzebujemy, aby pierwsze k elementów ciągu było parami różnych.

Rozwiązanie

Jeśli w ciągu jest mniej niż k różnych wartości, to nie da się osiągnąć celu, odpowiadamy zatem -1.

W przeciwnym wypadku spójrzmy na zbiór pozycji, z których elementy przemieszczą się na jedną z k pierwszych pozycji. Oznaczmy posortowany ciąg tych pozycji przez $p_1, p_2, \ldots, p_k$. Dla $i \neq j$ musi zachodzić $a_{p_i} \neq a_{p_j}$. Przemieszczenie jakiegoś elementu z pozycji i na pozycję j zajmuje nam $|i - j|$ ruchów. Okazuje się, że element z pozycji p_i opłaca się doprowadzić na pozycję i.

Dlaczego? Jeśli uprzemy się przy tej kolejności, to będziemy mogli doprowadzać elementy na miejsce jeden po drugim, idąc od lewej. Będziemy na to potrzebować $\sum_{i=1}^{n} |p_i - i|$ ruchów, a ponieważ musi zachodzić $p_i \geq i$, to $\sum_{i=1}^{n} |p_i - i| = \sum_{i=1}^{n} (p_i - i) = (\sum_{i=1}^{n} p_i) - (\sum_{i=1}^{n} i)$. Zauważmy, że przy każdym ruchu suma pozycji wybranych elementów może spadać co najwyżej o 1. Musimy zmniejszyć tę sumę o $(\sum_{i=1}^{n} p_i) - (\sum_{i=1}^{n} i)$, a skoro we wspomnianym sposobie zmniejszamy ją zawsze o 1, to nasze rozwiązanie jest optymalne.

Musimy zatem znaleźć takie k pozycji, których suma jest minimalna i wszystkie wartości, które się na nich znajdują, są parami różne. Łatwo zrobić to zachłannie. Możemy przejść się po ciągu od lewej i jeśli jakiejś wartości jeszcze nie widzieliśmy, to właśnie jej pozycję dodajemy do rozwiązania. Idziemy po ciągu tak długo aż spotkamy k różnych wartości.

Czemu jest to optymalne? Powiedzmy, że pierwszy raz napotykamy wartość w. Jeśli mielibyśmy wziąć do wyniku jakieś inne jej wystąpienie, to możemy poprawić wynik biorąc zamiast niego pierwsze wystąpienie w. Jeśli jednak mielibyśmy wziąć do rozwiązania całkiem inną wartość występującą później, to również możemy zamienić ją na pierwsze wystąpienie w.

Ponieważ jesteśmy w stanie zarezerwować tablicę, która pomoże nam śledzić które wartości już napotkaliśmy, to całkowita złożoność algorytmu to $\mathcal{O}(n)$.

1A – Od deski do deski

autor zadania: Mateusz Radecki

Treść

Mówimy, że ciąg jest dobry, jeśli możemy sprawić, że cały zniknie, wykonując dowolną liczbę następujących ruchów: wybieramy dwa równe elementy ciągu znajdujące się na różnych pozycjach i usuwamy je wraz ze wszystkimi elementami pomiędzy nimi. Następnie sklejamy ze sobą pozostały prefiks i sufiks ciągu.

Mamy dane liczby n oraz m. Należy policzyć (modulo duża liczba pierwsza) liczbę dobrych ciągów n-elementowych o wartościach z przedziału $[1, m]$.

Rozwiązanie

Zaznaczmy w oryginalnym ciągu pary pozycji, które odpowiadają parom wybieranych elementów. Spójrzmy na dwie pary (ℓ_1, r_1) oraz (ℓ_2, r_2). Oczywiście wszystkie wartości ℓ_1, r_1, ℓ_2 i r_2 muszą być parami różne. Załóżmy bez straty ogólności, że $\ell_1 < \ell_2$. Nie może zachodzić $\ell_2 < r_1$ i $r_1 < r_2$, gdyż wtedy usuwając najpierw pierwszy przedział, usunęlibyśmy ℓ_2, a usuwając najpierw drugi przedział, usunęlibyśmy r_1. Jeśli zachodzi $r_2 < r_1$, to przedział $[\ell_2, r_2]$ musieliśmy usunąć przed $[\ell_1, r_1]$. Łatwo zauważyć, że niepotrzebnie w ogóle usuwaliśmy przedział $[\ell_2, r_2]$, gdyż i tak cały zniknąłby przy usuwaniu przedziału $[\ell_1, r_1]$.

Możemy zatem wywnioskować, że ciąg jest dobry wtedy i tylko wtedy, gdy możemy podzielić go na przedziały długości co najmniej 2 takie, że każdy przedział zaczyna oraz kończy się tą samą wartością. Policzymy dobre ciągi za pomocą programowania dynamicznego. Niech $DP[i][j][l]$ oznacza liczbę takich ciągów i-elementowych, dla których istnieje dokładnie j wartości z przedziału $[1, m]$, które po dołożeniu na koniec ciągu zamieniały by go w dobry ciąg. Dodatkowo, jeśli $l = 1$, to zliczamy jedynie dobre ciągi, a jeśli $l = 0$, to zliczamy jedynie ciągi, które nie są dobre.

Programowanie dynamiczne inicjalizujemy jako $DP[0][0][1] = 1$. Przejdźmy się po i od 0 do $n - 1$ włącznie i wypychajmy wartości w przód.

Jeśli $l = 0$ i dołożymy wartość, która nie zamienia ciągu w dobry ciąg, to j nie ulega zmianie.

Wykonujemy zatem: $DP[i + 1][j][0] \mathrel{+}= dp[i][j][0] \cdot (m - j)$.

Jeśli zaś dołożymy wartość, która uczyni ciąg dobrym, to wartość j również się nie zmieni, ale ciąg będzie dobry.

Wykonujemy zatem: $DP[i+1][j][1] \mathrel{+}= DP[i][j][0] \cdot j$.

Jeśli $l = 1$ i dołożymy wartość, która zamienia ciąg w dobry ciąg, to wartość j również się nie zmienia.

Wykonujemy zatem: $DP[i+1][j][1] \mathrel{+}= DP[i][j][1] \cdot j$.

Jeśli zaś dołożymy wartość, która nie zmienia ciągu w dobry ciąg, to zauważmy, że po każdym następnym wystąpieniu tej wartości będziemy w stanie usunąć cały ciąg (gdyż będziemy mogli usunąć prefiks długości i oraz w jednym ruchu wszystko to co jest po nim).

Wykonujemy zatem: $DP[i+1][j+1][0] \mathrel{+}= DP[i][j][1] \cdot (m-j)$.

Warto zauważyć, że wartość parametru j nigdy nie przekroczy wartości n. Całe programowanie dynamiczne ma zatem złożoność $\mathcal{O}(n^2)$.

2C – Zakłócenia

autor zadania: Kamil Dębowski

Treść

Mamy dany ciąg binarny długości $8n$. Pytamy, czy da się pomieszać kolejność bitów w tym ciągu tak, aby po podzieleniu go na n przedziałów długości 8, każdy przedział odpowiadał binarnej reprezentacji małej litery alfabetu angielskiego w kodzie ASCII.

Rozwiązanie

Po pierwsze warto zauważyć, że jedyne co podczas mieszania bitów nie może ulec zmianie, to liczba jedynek. Możemy za to ułożyć bity w każdy ciąg, o ile liczba jedynek w nim jest równa liczbie jedynek w wejściowym ciągu. Policzmy zatem jedynki w wejściowym ciągu i oznaczmy ich liczbę przez k.

Zauważmy, że binarne reprezentacje małych liter alfabetu angielskiego w kodzie ASCII mogą mieć zapalone 3, 4, 5 lub 6 bitów. Poprawny ciąg może mieć zatem od $3n$ do $6n$ jedynek. Jeśli więc k nie należy do przedziału $[3n, 6n]$, to odpowiedź nie istnieje.

Jeśli zaś odpowiedź istnieje, to rozważmy dwa przypadki. Jeśli $k = 6n$, to możemy n razy wypisać literę mającą dokładnie 6 jedynek w swojej reprezentacji binarnej (czyli na przykład literę `w`).

W przeciwnym przypadku, niech $p = \lfloor \frac{k}{n} \rfloor$ oraz $r = (k \mod n)$. Zauważmy, że p zawiera się w przedziale $[3, 5]$. Jako że $p \cdot (n - r) + (p + 1) \cdot r = k$, to możemy $(n - r)$ razy wypisać literę mającą p zapalonych bitów w reprezentacji binarnej i r razy wypisać literę mającą $(p+1)$ zapalonych bitów w reprezentacji binarnej. Takie litery można znaleźć za pomocą pętli lub ręcznie wpisać je do kodu. Całkowita złożoność algorytmu to $\mathcal{O}(n)$.

2B – Pandemia

autor zadania: Mateusz Radecki

Treść

Istnieje n miast położonych przy długiej drodze. Mamy dany ciąg binarny długości n. Informuje nas on, które miasta zarażone są wirusem, a które nie. Każdego dnia możemy zaszczepić jedno miasto wolne od wirusa, a każdej nocy każde miasto zarażone wirusem zaraża nim swoich niezaszczepionych sąsiadów. Wszystkie zarażenia dzieją się jednocześnie. Należy podać minimalną możliwą liczbę miast zarażonych wirusem przy optymalnej strategii szczepienia.

Rozwiązanie

Przypadek, gdy wszystkie miasta są zdrowe, jest prostym przypadkiem, zatem pomińmy go.

Podzielmy ciąg na maksymalne przedziały zdrowych miast, otoczone z dwóch stron zarażonymi miastami. Owy ciąg oznaczmy przez $a_1, a_2, \ldots, a_k$. Dodatkowo, obliczmy długości najdłuższego zdrowego prefiksu oraz sufiksu i oznaczmy je przez b_1 oraz b_2.

Wprowadźmy pojęcie *frontu choroby*. Każdy przedział otoczony zarażonymi miastami posiada dwa fronty choroby i przy każdej turze bez szczepień liczba zarażonych miast w nim wzrasta o 2. Oba „ogony" (tzn. prefiks oraz sufiks) posiadają po jednym froncie choroby i liczba miast zarażonych w nich przy każdej turze bez szczepień wzrasta o 1.

Zastanówmy się, co dodatkowo może się dziać. Fronty choroby mogą same znikać, ponieważ całe przedziały mogą zostać zarażone. Działa to w pewien sposób na naszą korzyść – wirus przestaje zarażać więcej miast. Dodatkowo, każdym szczepieniem jesteśmy w stanie wyeliminować jeden front choroby – możemy zaszczepić któryś koniec zdrowego przedziału miast (w przypadku któregoś z ogonów należy zaszczepić koniec sąsiadujący z zarażonym miastem).

W związku z tym, powinniśmy najpierw eliminować fronty, które najpóźniej zniknęłyby same z siebie. Okazuje się, że możemy zatem posortować ciąg $a_1, a_2, \ldots, a_k$ niemalejąco i dla kolejnych przedziałów w nim szczepić je po kolei. Mówiąc dokładniej, najpierw szczepimy pierwszy koniec pierwszego przedziału, potem jego drugi koniec, potem pierwszy koniec drugiego przedziału i tak dalej. Można robić to jednocześnie symulując rozwój choroby i skracając przedziały.

Pozostaje pytanie co zrobić z ogonami, tzn. wartościami b_1 i b_2. Załóżmy, że znamy jakąś strategię szczepienia, tzn. wiemy w jakiej kolejności chcemy zaszczepić które końce których przedziałów. Zmieńmy kolejność tych szczepień i zastanówmy się co się stanie. Teoretycznie nadal eliminujemy jeden front choroby dziennie, ale teraz możemy nie zdążyć zaszczepić jakiegoś przedziału, przez co zniknie on sam. Zauważmy jednak, że jedynie poprawia to naszą sytuację! Fronty, na których zniknięcie nie musimy marnować szczepień działają na naszą korzyść. Wynika z tego, że znając sam zbiór końców przedziałów, które chcemy zaszczepić, możemy zaszczepić je w dowolnej kolejności. Możemy zatem rozważyć cztery przypadki: czy chcemy zaszczepić ogon b_1 oraz czy chcemy zaszczepić ogon b_2. Dla każdego z tych dwóch szczepień, które zdecydujemy się wykonać, możemy wykonać je na samym początku, a następnie szczepić przedziały opisywane przez ciąg $a_1, a_2, \ldots, a_k$ od najdłuższych do najkrótszych, symulując cały czas rozwój choroby.

Oczywiście opisane powyżej rozwiązanie bazuje jedynie na intuicji, dlatego dokładny dowód wspomnianych faktów pozostawiamy jako ćwiczenie dla czytelnika. Jeśli zdecydujemy się na szybszy niż standardowy sposób sortowania, to ostateczna złożoność algorytmu będzie wynosić $O(n)$.

2A – Poborcy podatkowi

autor zadania: Mateusz Radecki

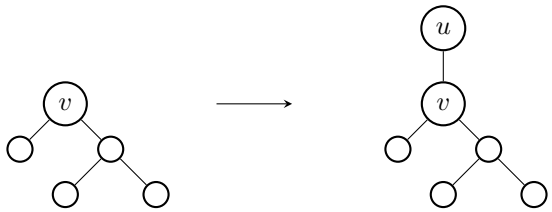
Treść

Dane jest drzewo z wagami na krawędziach. Należy wybrać dowolną liczbę rozłącznych krawędziowo ścieżek. Długość każdej ścieżki musi wynosić dokładnie 4, gdzie przez długość ścieżki rozumiemy liczbę krawędzi w niej. Należy zmaksymalizować sumę wag krawędzi pokrytych przez ścieżki.

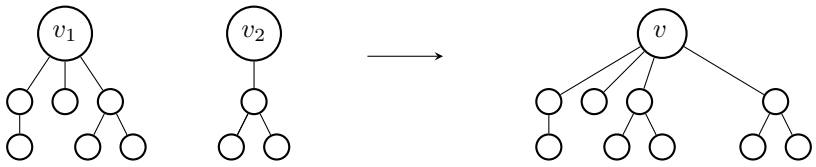
Rozwiązanie

Rozwiążmy zadanie za pomocą programowania dynamicznego. Ukorzeńmy drzewo w dowolnym wierzchołku. Niech $DP[v][i]$ (gdzie $0 \leq i < 4$). Oznacza maksymalną sumę wag krawędzi, jeśli wybieraliśmy ścieżki jedynie w poddrzewie zawieszonym w wierzchołku v, wszystkie wybrane ścieżki mają długość 4, z wyjątkiem jednej z nich, której jeden z końców to v i której długość wynosi i. Jak w standardowym programowaniu dynamicznym na drzewie, musimy umieć zrobić dwie rzeczy.

Dodać korzeniowi drzewa ojca:



Oraz połączyć korzeniami dwa drzewa, gdzie korzeń drugiego ma dokładnie jedno dziecko:



Pierwsze przejście jest prostsze. Znając tablicę $DP[v]$ i znając wagę krawędzi prowadzącej do ojca v (oznaczmy ją w), wystarczy wykonać:

$$DP[u][0] = max(DP[v][0], DP[v][3] + w)$$
$$DP[u][1] = DP[v][0] + w$$
$$DP[u][2] = DP[v][1] + w$$
$$DP[u][3] = DP[v][2] + w$$

Łączenie drzew jest nieco trudniejsze. Zauważmy, że ścieżki długości 1 łączą się ze ścieżkami długości 3, a ścieżki długości 2 łączą się z innymi ścieżkami długości 2. Ścieżki długości 0 (czyli brak ścieżki) nie wpływają na nic. Dla rozważonego prefiksu dzieci v interesuje nas zatem o ile więcej ścieżek długości 1 niż ścieżek długości 3 wybraliśmy, oraz jaka jest parzystość liczby wybranych ścieżek długości 2.

Oznaczmy zatem długość rozważonego prefiksu dzieci przez i, różnicę między liczbą wybranych ścieżek długości 1 i liczbą wybranych ścieżek długości 3 przez j, a parzystość liczby wybranych ścieżek długości 2 przez ℓ. Ten problem możemy również rozwiązać za pomocą programowania dynamicznego. Wprowadźmy tablicę $DP_2[i][j][\ell]$, która odpowiada maksymalnej możliwej sumie wag krawędzi w scenariuszu opisanym wyżej. Łatwo wywnioskować, że zachodzi $DP[v] = \{DP_2[d][0][0], DP_2[d][1][0], DP_2[d][0][1], DP_2[d][-1][0]\}$, gdzie d jest liczbą dzieci v. Z każdego stanu mamy stałą liczbę przejść (dla dziecka v możemy wybrać jedną z czterech możliwych długości ścieżek), stanów mamy jednak kwadratowo wiele, przez co potrzebujemy jakiejś optymalizacji.

Wśród dzieci v istnieje pewne optymalne przyporządkowanie każdemu dziecku długości ścieżki, która ma odchodzić w jego stronę. W tym przyporządkowaniu jest w szczególności podciąg dzieci, które będą miały przyporządkowaną ścieżkę długości 1 lub 3 (a liczności takich dzieci muszą finalnie różnić się co najwyżej o 1). Zauważmy, że gdybyśmy znali taką kolejność dzieci v, w której dzieci, którym należy przyporządkować ścieżki długości 1 oraz 3, występowały na zmianę, to parametr j wcale nie musiałby mieścić się w przedziale $[-n, n]$ – mógłby wtedy mieścić się w przedziale $[-1, 1]$. Niestety, nie znamy takiej kolejności, ale napawa nas to nadzieją – spróbujmy ułożyć dzieci v w losowej kolejności!

Wyobraźmy sobie długi ciąg binarny, w którym zera odpowiadają dzieciom, w które odchodzić będzie ścieżka długości 1, a jedynki odpowiadają dzieciom, w które odchodzić będzie ścieżka długości 3. Wiemy, że nasz ciąg został losowo przemieszany. Zastanówmy się zatem, jaka może być oczekiwana maksymalna różnica między liczbą zer i liczbą jedynek w pewnym prefiksie. Jeśli wszystkie elementy ciągu byłyby niezależnie od siebie losowane ze zbioru $\{0, 1\}$, to interesowałoby nas odchylenie standardowe dość prostej zmiennej, które podpowiadałoby, że szukaną przez nas odpowiedzią jest $\mathcal{O}(\sqrt{n})$. Dodatkowo wiemy, że sumaryczna liczba zer oraz liczba jedynek różnią się co najwyżej o 1, co okazuje się jedynie nam pomagać (jeśli do tej pory spotkaliśmy więcej zer niż jedynek, to następna wartość ma większe szanse być jedynką niż zerem i vice versa).

Warto jednak przekonać się o tym nieco dogłębniej, szczególnie, że musimy ręcznie wybrać jakąś stałą, przez którą ograniczymy możliwe wartości parametru j. Możemy zatem napisać dość prosty program pomocniczy, który (również korzystając z programowania dynamicznego) policzy w złożoności $\mathcal{O}(n \cdot c)$ prawdopodobieństwo, że przy ograniczeniu możliwego przedziału parametru j do $[-c, c]$, nasz algorytm zadziała poprawnie. Przemyślenie szczegółów wspomnianego programu pozostawiamy jako ćwiczenie dla czytelnika. Z napisanego programu łatwo uzyskać informację, że rzeczywiście wartość której szukamy jest rzędu $\mathcal{O}(\sqrt{n})$. Przy jego pomocy możemy również dobrać odpowiednią stałą, zależną od prawdopodobieństwa, którego oczekujemy. Możemy również oczywiście próbować jak najdokładniej dowodzić prawdopodobieństwa sukcesu naszego algorytmu, jednak w konkursach programistycznych, w których nie ma potrzeby dowodzenia poprawności swoich rozwiązań, wspomniany program powinien być w zupełności wystarczający.

Ostateczna złożoność algorytmu wynosi zatem $\mathcal{O}(n \cdot \sqrt{n})$.

3C – Sumy

autor zadania: Mateusz Radecki

Treść

Dany jest ciąg wag ryb. Jedna ryba może zjeść inną tylko, gdy jest od niej ściśle cięższa. Po zjedzeniu jej waga rośnie o wagę zjedzonej ryby, a zjedzona ryba znika. Dla każdej ryby pytamy, czy może w stawie nastąpić ciąg zjedzeń, po którym pozostanie ona ostatnią, niezjedzoną rybą.

Rozwiązanie

Po pierwsze zauważmy, że jeśli jakaś ryba może pozostać ostatnią rybą po jakimś ciągu zjedzeń, to równie dobrze ona mogła zjeść każdą inną rybę. Powiedzmy, że istnieją ryby A, B i C o wagach odpowiednio a, b i c. Jeśli B zjadła C, a potem A zjadła B, to musiało zachodzić $b > c$ oraz $a > b + c$. Łatwo zauważyć, że w takim razie musiało też zachodzić $a > b$ oraz $a + b > c$, czyli A mogła zjeść też bezpośrednio C.

Teraz zauważmy, że jeśli jakaś ryba mogła pozostać ostatnia, to wszystkie ryby cięższe od niej również mogą pozostać ostatnie. Powiedzmy, że znamy dwie ryby A oraz B i zachodzi $A \geq B$. Za każdym razem, gdy B miałaby zjeść jakąś inną rybę, zamiast tego A mogłaby ją zjeść. W momencie, gdy B zjadłaby A, to zamiast tego A może zjeść B.

Daje nam to informację, że jeśli posortujemy ryby niemalejąco po wadze, to ryby na pewnym sufiksie mogą pozostać ostatnie. Pozwala nam to użyć wyszukiwania binarnego, by mnożąc złożoność przez $\log(n)$ sprowadzić zadanie do sprawdzenia, czy pewna konkretna ryba może pozostać ostatnia.

Pozostaje jedynie pytanie w jakiej kolejności sprawdzana ryba powinna próbować zjadać pozostałe. Jeśli może zjeść którąkolwiek z nich, to na pewno może zjeść najlżejszą, co tylko podniesie jej wagę i być może pozwoli zjeść większe ryby. Skoro i tak posortowaliśmy ciąg, to przejdźmy się po nim od najlżejszych ryb próbując po kolei je jeść. Jeśli którejś zjeść nie możemy, to znaczy, że sprawdzana ryba nie może pozostać ostatnia.

Cały algorytm, zarówno przez sortowanie, jak i wyszukiwanie binarne, ma złożoność $\mathcal{O}(n \cdot \log(n))$.

3B – Mopadulo

autor zadania: Kamil Dębowski

Treść

Dany jest ciąg liczb całkowitych. Należy policzyć (modulo $10^9 + 7$) sumę jego dobrych podziałów na przedziały. Podział ciągu na przedziały jest dobry, jeśli reszta z dzielenia sumy liczb w każdym z przedziałów przez 10^9+7 jest parzysta.

Rozwiązanie

Dla zadanego ciągu $a_1, a_2, \ldots, a_n$ policzmy ciąg jego sum prefiksowych, a dokładniej ich reszt z dzielenia przez 10^9+7. Oznaczmy go $pref_0, pref_1, \ldots, pref_n$, gdzie $pref_0 = 0$ i $pref_i = (pref_{i-1} + a_i) \mod (10^9 + 7)$. Zastanówmy się, jaka jest reszta z dzielenia przez $10^9 + 7$ sumy liczb z przedziału $[\ell, r]$. Jeśli zachodzi $pref_r \geq pref_{\ell-1}$, to suma ta to po prostu $pref_r - pref_{\ell-1}$. Jeśli zaś zachodzi $pref_r < pref_{\ell-1}$, to policzona w ten sposób suma okazałaby się ujemna, musimy ją zatem zastąpić przez $pref_r - pref_{\ell-1} + 10^9 + 7$.

Łatwo zatem wywnioskować kiedy reszta z dzielenia sumy liczb z jakiegoś przedziału jest parzysta. Dla przedziału $[\ell, r]$, jeśli zachodzi $pref_r \geq pref_{\ell-1}$, to musi zachodzić $pref_r \equiv pref_{\ell-1} \pmod 2$. Jeśli zaś zachodzi $pref_r < pref_{\ell-1}$, to musi zachodzić $pref_r \not\equiv pref_{\ell-1} \pmod 2$.

Użyjmy więc programowania dynamicznego. Niech $DP[i]$ oznacza liczbę podziałów na przedziały prefiksu długości i. Zaczynamy oczywiście od $DP[0] = 1$. Aby policzyć $DP[i]$ przeiterujmy się po początku ostatniego przedziału w podziale. Niech zaczyna się na pozycji j. Jest on poprawny wtedy i tylko wtedy, gdy reszta z dzielenia sumy liczb w nim jest parzysta. Musimy zatem wtedy zwiększyć $DP[i]$ o $DP[j-1]$. Gdybyśmy zaimplementowali przejścia brutalnie, skończylibyśmy ze złożonością kwadratową, która zdecydowanie nas nie zadowala.

Z zebranymi obserwacjami łatwo jest jednak wywnioskować, co należy zrobić. Skoro by obliczyć $DP[i]$ musimy poznać sumę po $DP[j]$ dla takich j, że $pref_j \leq pref_i$ oraz $pref_i \equiv pref_j \pmod 2$, to możemy w tym celu użyć drzewa przedziałowego. Po przeskalowaniu współrzędnych (którymi są sumy prefiksowe) pozwoli nam ono odczytywać sumę wartości z dowolnego prefiksu. Ponieważ interesuje nas jedynie suma dla wartości o zadanej parzystości, to możemy zbudować dwa drzewa przedziałowe – jedno dla wartości parzystych i jedno dla nieparzystych.

Dzięki temu będziemy mogli pytać zarówno o prefiks wartości o zadanej parzystości, jak i sufiks wartości o przeciwnej parzystości, a także wprowadzać zmiany (dodawać nowo policzone $DP[i]$ na pozycji $pref_i$). Każda z tych operacji będzie standardową operacją na drzewie przedziałowym. Zamiast drzewa przedziałowego, w celu uproszczenia kodu, możemy również użyć drzewa Fenwicka (zwanego również BIT lub drzewem potęgowym). W obu wariantach, ze względu na przeskalowanie oraz użycie odpowiedniej struktury danych, kończymy ze złożonością $\mathcal{O}(n \cdot \log(n))$.

3A – Wystawa

autor zadania: Mateusz Radecki

Treść

Dany jest ciąg par liczb (a_i, b_i) oraz liczba k. Należy stworzyć ciąg $c_1, c_2, \cdots, c_n$. W tym celu, musimy wybrać dokładnie k pozycji i ustalić na nich $c_i := a_i$. Na pozostałych pozycjach należy ustalić $c_i := b_i$. Naszym celem jest zminimalizowanie maksymalnej sumy liczb tworzących spójny przedział ciągu $c_1, c_2, \ldots, c_n$. Dodatkowo, jesteśmy proszeni o odtworzenie wyniku, tzn. skonstruowanie dokładnego ciągu $c_1, c_2, \ldots, c_n$.

Rozwiązanie

Spróbujmy stworzyć optymalny ciąg bez względu na parametr k. W tym celu dla każdej pozycji i wybieramy mniejszą z dwóch wartości: a_i oraz b_i. W tak stworzonym ciągu któregoś wyboru dokonaliśmy za dużo razy – bez straty ogólności załóżmy, że za dużo razy wybraliśmy a_i. Możemy zatem wywnioskować, że na każdej pozycji, na której wybraliśmy b_i, był to wybór którego nie należy zmieniać. Możemy zatem na tych pozycjach ustalić $a_i := b_i$. Dzięki temu zyskujemy założenie, że dla każdego i zachodzi $a_i \leq b_i$, które okaże się być wygodne podczas implementacji. Musimy teraz wybrać dokładnie k pozycji, na których zdecydujemy się na wybór a_i. Zauważmy, że wspomniane przypisanie sprawia, że możemy myśleć o wybraniu co najwyżej k pozycji, na których zdecydujemy się na wybór a_i, co również może okazać się wygodne, zależnie od szczegółów implementacji.

Zacznijmy od bardzo wolnego rozwiązania i optymalizujmy je. W tym celu warto zastanowić się, jakie znamy algorytmy liczące przedział o największej sumie i który będzie dla nas przydatny. Skorzystamy tutaj z algorytmu, który iteruje się po kolejnych prefiksach i dla każdego z nich oblicza maksymalną sumę liczb na jakimś sufiksie tego prefiksu. Powiedzmy, że rozważyliśmy już prefiks długości i, a największa suma liczb na jego sufiksie wynosi s. Przechodząc do prefiksu długości $i + 1$ ustalamy $s := s + a_{i+1}$. Następnie, mamy możliwość wyzerować długość sufiksu o największej sumie – jeśli największy sufiks zawiera ostatni element, to przed nim na pewno używamy dokładnie tego sufiksu, który był optymalny dla prefiksu krótszego o 1. Jeśli go nie zawiera, to suma liczb w optymalnym sufiksie to po prostu 0. W tym celu ustalamy $s := max(s, 0)$.

Pamiętamy również globalną zmienną m, ustaloną początkowo na 0. Po każdym przedłużeniu prefiksu ustalamy $m := max(m, s)$. Końcowa wartość zmiennej m jest szukanym wynikiem. Pozwala nam to skonstruować rozwiązanie wykładnicze. Idąc od lewej, rekurencyjnie rozważamy dla każdej pozycji, czy zdecydujemy się na wybranie na niej wartości a_i czy b_i. Interesującymi nas wartościami, są oczywiście wartości s oraz m, a także liczba pozycji na których zdecydowaliśmy się na wybór a_i. Możemy zatem opisać stan krotką czterech liczb: (i, j, s, m), gdzie i oznacza długość rozważonego prefiksu, a j oznacza liczbę pozycji, na których wybraliśmy element ciągu $a_1, a_2, \ldots, a_n$.

W podobnych sytuacjach, gdy musimy minimalizować maksimum po jakichś wartościach, warto jest rozważyć wyszukiwanie binarne po wyniku. Sprawia to, że mamy globalne założenie, którego musimy pilnować. Spróbujmy więc zastosować je tutaj i skupmy się na sprawdzeniu, czy da się sprawić, by suma liczb w żadnym przedziale nie przekraczała X. Okazuje się zmieniać to stan z (i, j, s, m) w (i, j, s) – nie interesuje nas maksymalna do tej pory wartość s, ważne jedynie, że nigdy nie przekroczyła ona X.

Pozwala to nam wysunąć obserwację: dla ustalonych wartości i oraz j nie interesują nas wszystkie możliwe wartości s, do których mogliśmy dojść bez przekraczania X – interesuje nas jedynie najmniejsza z nich! Możemy zatem użyć programowania dynamicznego: niech $DP[i][j]$ oznacza minimalną możliwą do osiągnięcia wartość s, przy założeniu, że rozważyliśmy już prefiks długości i, j razy wybraliśmy na nim element ciągu $a_1, a_2, \ldots, a_n$ i suma w żadnym przedziale do tej pory nie przekroczyła X. Dodatkowo, niektóre j mogą w ogóle nie być osiągalne. Oznacza to, że nie mogliśmy wybrać tylko tyle wartości a_i, żeby nigdzie nie przekroczyć X. Osiągalny będzie sufiks możliwych wartości j. Aby nieco uprzyjemnić implementację warto w tym momencie zmienić definicję – ustalamy zatem $k := n - k$ i mówimy, że musimy co najmniej k razy wybrać wartość b_i (czyli tę większą). Sprawi to, że tylko prefiks wartości j będzie osiągalny. Powyższe podejście ma złożoność kwadratową, więc potrzebujemy jeszcze jednej obserwacji.

Zaimplementowanie powyższego programowania dynamicznego pozwala zauważyć ważną własność: dla ustalonego i, wartości $DP[i][j]$ są oczywiście niemalejące, ale również wypukłe! Innymi słowy, wartości $DP[i][j] - DP[i][j-1]$ również są niemalejące. Doświadczeni zawodnicy powinni w tym momencie wiedzieć, co należy zrobić. Zamiast utrzymywać wszystkie wartości $DP[i][j]$, utrzymujmy jedynie różnice między nimi oraz wartość $DP[i][0]$. Zastanówmy się, jak zmieniają się owe różnice przy przejściu do następnego prefiksu. Musimy wykonać $DP[i][j] = max(min(DP[i-1][j-1] + b_i, DP[i-1][j] + a_i), 0)$. Spróbujmy rozłożyć to przejście na nieco mniejsze składowe. Po pierwsze, możemy każdą wartość zwiększyć o a_i – wystarczy w tym celu zwiększyć $DP[i][0]$ o a_i oraz nie zmieniać różnic. Następnie, dla każdego elementu jednocześnie, musimy wybrać mniejszą z dwóch możliwości – $DP'[i][j] = min(DP[i][j], DP[i][j-1] + b_i - a_i)$. Warto zauważyć, że skoro różnice były posortowane, to opisana operacja jest równoważna włożeniu wartości $b_i - a_i$ do posortowanego ciągu różnic. Sprawia to, że możemy do utrzymywania tego ciągu użyć gotowych struktur danych oferowanych przez popularne języki programowania.

Następną operacją, którą musimy wykonać, jest zwiększenie wszystkich ujemnych wartości do zera. Ponieważ nadal wartości $DP[i]$ są niemalejące, to jedynie prefiks wartości mógł stać się ujemny, musimy więc go zwiększyć. Zauważmy, że jeśli musielibyśmy zwiększyć do zera wiele pierwszych wartości, to różnice między nimi staną się zerami. Aby nasze rozwiązanie było dostatecznie szybkie, możemy zamiast trzymać w strukturze wszystkie zerowe różnice, trzymać jedynie ich liczbę – dzięki temu nie będziemy musieli się po nich wszystkich iterować i nasze rozwiązanie będzie się dobrze amortyzowało – potencjałem będzie liczba niezerowych różnic.

Ostatnią rzeczą jest ewentualne usunięcie sufiksu wartości, które przekraczają X. Tutaj musimy oprócz wartości $DP[i][0]$ i ciągu różnić pamiętać również sumę różnic. Pozwoli nam to łatwo obliczać ostatni element ciągu, stwierdzać czy jest on większy niż X i ewentualnie usuwać ostatnie wartości.

Umiemy zatem powiększać prefiks o 1 w złożoności $\mathcal{O}(\log(n))$. Po przejściu przez cały ciąg wystarczy sprawdzić, czy w strukturze mamy co najmniej k różnic, czyli czy wartość $DP[n][k]$ jest osiągalna. Jeśli tak, to mogliśmy wybrać co najmniej k różnych pozycji i ustalić na nich b_i tak, by suma liczb w żadnym przedziale nie przekroczyła X. Zauważmy również, że z przebiegu algorytmu możemy wyciągnąć wszystkie informacje potrzebne nam do odtworzenia wyniku – możemy dla każdej różnicy pamiętać której pozycji ona odpowiada.

Całe rozwiązanie, ze względu na wyszukiwanie binarne oraz użycie odpowiedniej struktury danych, ma złożoność $\mathcal{O}(n \cdot \log(n) \cdot \log(n \cdot A))$, gdzie A jest górnym ograniczeniem na wartości w ciągach $a_1, a_2, \ldots, a_n$ oraz $b_1, b_2, \ldots, b_n$.

4C – Ranking sklepów internetowych

autor zadania: Mateusz Radecki

Treść

Dana jest permutacja liczb od 1 do n. Przez medianę ciągu rozumiemy środkowy element ciągu po posortowaniu, lub jeśli ciąg ma parzyście wiele elementów, średnią arytmetyczną dwóch środkowych elementów. Wartością ciągu określamy jego długość powiększoną o dwukrotność jego mediany. Należy wyznaczyć maksymalną wartość przedziału zadanej permutacji oraz policzyć ile przedziałów ma właśnie taką wartość.

Rozwiązanie

Pierwszą istotną obserwacją jest spostrzeżenie, że maksymalna wartość przedziału wynosi dokładnie $2n + 1$. Zawsze da się uzyskać co najmniej tyle, gdyż na przykład przedział zawierający jedynie liczbę n ma właśnie taką wartość. Czemu nie da się uzyskać więcej, przekonamy się nieco później.

Załóżmy, że przedział ma mieć długość d. W takim razie, by mieć wspomnianą, maksymalną wartość, mediana elementów w nim musi wynosić $\frac{2n+1-d}{2}$. Innymi słowy, dla przedziałów długości $1, 2, 3, 4, \ldots$, mediana musi wynosić dokładnie $n, n-1+\frac{1}{2}, n-1, n-2+\frac{1}{2}, \ldots$.

Zastanówmy się co dokładnie to oznacza. Jeśli przedział ma mieć nieparzystą długość postaci $2k-1$, to musi on zawierać k największych elementów, a reszta elementów nie ma żadnego znaczenia. Łatwo też zauważyć, że w przypadku takich przedziałów mediana nie może być większa.

Jeśli zaś przedział ma mieć parzystą długość $2k$, to musi on zawierać $k+1$ największych wartości, a pozostałe z nich nie mają znaczenia. Tutaj również nietrudno się przekonać, że mediana nie może być większa.

Dla każdej długości przedziału wiemy zatem ile największych wartości przedział ten musi zawierać, by osiągać maksymalną możliwą wartość. Możemy więc dla owego sufiksu wartości znaleźć skrajnie lewą i skrajnie prawą, ponieważ tylko one są interesujące. Pozwoli to nam, w czasie stałym, wyznaczyć liczbę przedziałów mieszczących się w całym ciągu, które dodatkowo zawierają wszystkie wymagane elementy.

Całkowita złożoność algorytmu wynosi zatem $\mathcal{O}(n)$.

4B – Skrzyżowania

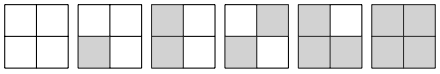
autorzy zadania: Bartosz Kostka i Mateusz Radecki

Treść

Dana jest prostokątna plansza $n \times m$ placów, gdzie każda kolumna placów oraz każdy rząd placów są oddzielone ulicą. Dla każdej czwórki placów tworzących kwadrat 2×2 znamy cykl świateł dla pieszych – dany jest ciąg znaków mówiący dla każdej sekundy czy w danym momencie na zielono świecą się światła przecinające poziomą czy pionową ulicę. Wszystkie cykle są dość małej długości. Mamy dane dużo zapytań o pieszego wyruszającego z jakiegoś placu w jakimś momencie czasu oraz informację o jego docelowym placu. Dla każdego zapytania należy powiedzieć, kiedy najwcześniej pieszy może dotrzeć do swojego celu.

Rozwiązanie

Oczywiście możemy zinterpretować place jako wierzchołki grafu i przejścia dla pieszych jako krawędzie między nimi. Dla ustalonego momentu w czasie zastanówmy się, które place są z których osiągalne. Rozważmy zbiór placów osiągalny z pewnego ustalone placu. Spójrzmy w szczególności na jakieś cztery place układające się w kwadrat 2×2. Z dokładnością co do obrotów, osiągalne place (zaznaczone ciemniejszym kolorem) w tym kwadracie mogą w teorii wyglądać na jeden z następujących sposobów:



Pierwszy, trzeci i szósty układ od lewej są w porządku i mogą wystąpić na planszy, jednak dla drugiego, czwartego i piątego układu nie istnieje stan świateł na środku rysunku, który prowadziłby do właśnie takiej kombinacji osiągalnych placów. Z tego możemy zatem wywnioskować, że każda spójna składowa w grafie jest prostokątem, którego albo zarówno górny bok sięga do samej góry planszy, a dolny bok do dołu planszy, albo lewy i prawy bok sięgają aż do odpowiednio lewego i prawego boku planszy.

Może zdarzyć się tak, że wszystkie światła przecinające jakąś ulicę świecą się w danym momencie na czerwono. Wtedy w żaden sposób nie można przedostać się z jej jedną stronę na drugą. Takie zjawisko, które jest jedynym sposobem na oddzielenie od siebie dwóch spójnych składowych grafu, będziemy określać jako *barierę*.

Oczywiście w żadnym momencie nie może istnieć zarówno pionowa i pozioma bariera. Korzystając z tego faktu możemy wnioskować dalej, że mając przejść z placu (x_1, y_1) do placu (x_2, y_2), kompletnie niezależnymi problemami jest pokonać tę trasę w pionie oraz w poziomie. Wynika to z tego, że jeśli w momencie startu istnieje jakakolwiek bariera oddzielająca pole startowe od pola końcowego, to na pewno nie istnieje na planszy żadna bariera o przeciwnej orientacji, zatem możemy momentalnie zrównać odpowiednią współrzędną pieszego z tą współrzędną pola końcowego. Od tej pory będziemy rozwiązywać problem niezależnie dla poziomych oraz pionowych barier. Dodatkowo, możemy dla każdego wymiaru rozważyć dwa przypadki, zależnie od tego czy pieszy chce przemieścić się w stronę rosnących czy malejących współrzędnych. Dla ustalenia uwagi załóżmy, że chce on przemieścić się w stronę wierszy o większych numerach, tzn. chce przejść z wiersza ℓ do wiersza r, gdzie zachodzi $\ell < r$.

Zauważmy teraz, że ponieważ długości wszystkich cykli nie przekraczają 8 (oznaczmy tę stałą jako c), to cykl tego jak wygląda cała plansza nie przekracza najmniejszej wspólnej wielokrotności liczb od 2 do c, która to jest równa 840. Oznaczmy tę stałą przez k.

Chcielibyśmy zatem dla każdej poziomej ulicy obliczyć maskę bitową rozmiaru k, która mówi w jakich momentach w czasie jest ona barierą. Nie warto ryzykować zrobieniem tego za wolno, więc zastanówmy się jak szybko możemy to osiągnąć. Po pierwsze, dla ustalonej ulicy i dla każdej liczby i od 2 do c, chcielibyśmy obliczyć w jakich momentach (modulo i) wszystkie skrzyżowania na tej ulicy o długości cyklu i nie pozwalają przez nią przejść. Ponieważ wszystkie skrzyżowania na tej ulicy muszą nie pozwalać przez nią przejść, to interesuje nas bitowy `AND` wszystkich masek dla tych skrzyżowań. Łatwo obliczyć go liniowo względem ich liczby. Następnie, mając maski dla każdej długości i od 2 do c, możemy przeiterować się po reszcie z dzielenia momentu w czasie przez k i dla każdego z nich, w złożoności $\mathcal{O}(c)$ sprawdzić, czy rzeczywiście bariera wtedy istnieje. Sprawi to, że w sumarycznej złożoności $\mathcal{O}(nm + (n+m)ck)$, dowiemy się w których momentach w czasie która bariera istnieje. Jeśli zależy nam na jeszcze większym przyspieszeniu, możemy skorzystać z preprocessingu w złożoności $\mathcal{O}(2^c k)$, który pozwoli nam policzyć to samo w złożoności $\mathcal{O}(2^c k + nm + (n+m)c\frac{k}{64})$, jednak nie jest to kluczowe przyspieszenie i nie poświęcimy mu szczególnej uwagi.

Zastanówmy się zatem, w jaki sposób brutalnie moglibyśmy próbować przejść z placu w wierszu ℓ do placu w wierszu r, znając startowy czas t. Oczywiście musimy poczekać w wierszu ℓ tak długo, aż bariera oddzielająca wiersz ℓ od wiersza $\ell+1$ zniknie. Wtedy opłaca się nam zachłannie przejść do wiersza $\ell+1$ i kontynuować trasę. Wyobraźmy sobie zatem prostokątną planszę rozmiaru $(n+1) \times k$. Dla każdego pola (i, j) (gdzie $0 \leq i \leq n$ oraz $0 \leq j < k$) wiemy, czy musimy przejść z niego do pola $(i, (j+1) \mod k)$, czy do pola $(i+1, j)$. Możemy zatem wyobrazić sobie, że z każdego pola (poza tymi w ostatnim w rzędzie) wychodzi jedna krawędź skierowana, o wadze 1 lub 0, zależnie od tego, czy owa krawędź oznacza poczekanie w tym samym rzędzie, czy ruszenie się o jeden rząd do przodu.

Dodatkowo, dla każdej bariery musi istnieć moment, gdy ona nie istnieje, ponieważ zgodnie z treścią zadania każde światło kiedyś zmienia swój stan. Możemy zatem wywnioskować, że jeśli potraktujemy opisane pola jako graf, to jest on ukorzenionym lasem, gdzie pola w ostatnim rzędzie są korzeniami.

Dla każdego zapytania (ℓ, r, t) interesuje nas odległość między polem $(\ell, t \mod k)$, a najbliższym mu polem w rzędzie r. Zamiast iść od pola startowego do końcowego rzędu, możemy użyć algorytmu DFS na każdym z korzeni. Owe przeszukiwanie w głąb powinno pamiętać aktualny wierzchołek oraz jego odległość do korzenia. Dodatkowo, przyda się nam globalna tablica rozmiaru $n+1$, nazwijmy ją $WHEN[i]$. Za każdym razem, gdy w przeszukiwaniu w głąb będziemy przechodzić z rzędu i do rzędu $i-1$, zapiszemy w polu $WHEN[i]$ odległość od pola, z którego właśnie wyszliśmy, do korzenia. Jeśli dla każdego pola stworzymy listę zapytań, które zaczynają właśnie w tym polu, to odwiedzając je w trakcie algorytmu DFS będziemy znali zarówno odległość od tego pola do korzenia, jak i odległość od odpowiedniego pola w docelowym rzędzie do korzenia. Czas, jakiego pieszy potrzebuje na pokonanie trasy, jest oczywiście równy różnicy tych dwóch wartości.

Ze wszystkimi zebranymi obserwacjami i pomysłami, jesteśmy w stanie rozwiązać zadanie w złożoności $\mathcal{O}(2^c k + nm + (n+m)c\frac{k}{64} + (n+m)k + q)$.

4A – Areny

autorzy zadania: Mateusz Radecki i Marek Sokołowski

Treść

Dany jest graf skierowany o n wierzchołkach, w którym z każdego wierzchołka wychodzi przynajmniej jedna krawędź. Musimy rozwiązać następujące zadanie dla każdej liczby k z przedziału $[1, n]$:

Załóżmy, że każdy wierzchołek jest biały lub czarny. W jednym ruchu możemy kliknąć dowolny czarny wierzchołek o indeksie nieprzekraczającym k, a losowy wierzchołek, do którego prowadzi krawędź z klikniętego wierzchołka, staje się czarny (niezależnie od tego jakiego koloru był). Mówimy, że para wierzchołków (v, u) (gdzie $v \neq u$) jest dobra, jeśli zaczynając ze stanem, w którym wszystkie wierzchołki poza v są białe, a wierzchołek v jest czarny, jesteśmy w stanie wymusić, by w skończonej liczbie kroków kliknąć wierzchołek u. Zadaniem jest policzyć dobre pary wierzchołków.

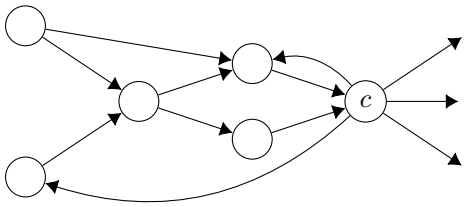
Rozwiązanie

Po pierwsze spróbujmy lepiej zrozumieć losowy proces opisany w treści zadania. Skoro musimy wymusić, by coś stało się w skończonym czasie, to możemy założyć, że wszystko co w opisanym procesie jest losowe, jak tak naprawdę dla nas złośliwe i dzieje się tak, jakbyśmy tego nie chcieli. Możemy więc o opisanym procesie myśleć tak, jakby istniał drugi gracz, który (dla ustalonej pary wierzchołków (v, u)) działa tak, byśmy na pewno nie byli w stanie dojść do wierzchołka u (o ile to dla niego możliwe).

Na początek spróbujmy obliczyć wynik dla $k = n$, czyli załóżmy, że możemy klikać we wszystkie wierzchołki. Spróbujmy policzyć, dla ustalonego wierzchołka u, ile istnieje wierzchołków v, że para (v, u) jest dobra. Jeśli z każdego wierzchołka wychodzi przynajmniej jedna krawędź, która nie prowadzi do u, to drugi gracz może przy każdym kliknięciu jakiegokolwiek wierzchołka zamalowywać na czarno wierzchołek różny od u. Oznaczałoby to, że nie istnieją takie dobre pary aren. Jeśli zaś istnieje wierzchołek (nazwijmy go w), z którego wszystkie krawędzie prowadzą do u, to jeśli on byłby czarny, to klikając w w, na pewno zamalujemy na czarno wierzchołek u, w który następnie będziemy w stanie kliknąć.

Jeśli znajdziemy taki wierzchołek w, to zmienia się jeszcze jedna rzecz: teraz, gdy istnieje wierzchołek z, z którego wszystkie krawędzie prowadzą do u lub do w, to jeśli jest on czarny, to również na pewno będziemy w stanie kliknąć w u. Gdyby drugi gracz zamalował na czarno wierzchołek u, to oczywiście możemy go kliknąć. Jeśli zaś zamaluje wierzchołek w, to wiemy, że jesteśmy w stanie wymusić kliknięcie u. Sprawia to, że para (z, u) również jest dobra. Proces ten możemy kontynuować. Powiedzmy, że wierzchołek x jest dobry, gdy $x = u$ lub para (x, u) jest dobra. Zawsze, gdy istnieje wierzchołek y, z którego krawędzie prowadzą do dobrych wierzchołków, to oznacza to, że on również jest dobry. Jeśli zaś nie istnieje taki wierzchołek, to należy zakończyć proces.

Umiemy zatem w czasie wielomianowym znaleźć wszystkie dobre pary wierzchołków. Musimy jednak zatroszczyć się, by cały proces przebiegał szybko. Wprowadźmy pojęcie „grupy" wierzchołków oraz „czoła" grupy. Niech grupą wierzchołków będzie podzbiór wierzchołków wraz ze wszystkimi krawędziami, które z nich wychodzą. Każda grupa powinna posiadać jeden wierzchołek, który nazywamy czołem. Wszystkie krawędzie z wierzchołków różnych od czoła muszą prowadzić do wierzchołków, które należą do grupy. Krawędzie z czoła mogą prowadzić zarówno poza grupę, jak i do niej. Dodatkowo, nie może istnieć cykl składający się jedynie z wierzchołków należących do grupy i niebędących jej czołem. Przykładowa grupa, z czołem zaznaczonym literą c, może wyglądać następująco:



Ustalmy więc, na początku działania algorytmu, że każdy wierzchołek jest niezależną grupą. Naszym zadaniem będzie wykrywać sytuację, gdy wszystkie krawędzie z pewnej grupy x (a dokładniej z jej czoła) trafiają do tej samej grupy y (oraz $x \neq y$).

W tym celu, możemy dla każdego wierzchołka v, wylosować pewną wartość $rand[v]$. Dla każdego wierzchołka v, będziemy również pamiętać wartość $hash[v]$, będącą sumą po wartościach $rand[head[u]]$ po wszystkich krawędziach $v \to u$. Wartość $head[v]$ oznacza czoło grupy, do której aktualnie należy v. Znając dowolną krawędź wychodzącą z v łatwo jest sprawdzić, czy grupę, w której czołem jest v, należy podłączyć do innej grupy. Jeśli ta krawędź wychodzi do wierzchołka u, to łączyć grupy musimy gdy $hash[v] = deg(v) \cdot rand[head[u]]$ oraz $head[u] \neq v$, gdzie $deg(v)$ jest liczbą krawędzi, które wychodzą z v.

Pytanie, które musimy sobie zadać, to jak sprytnie łączyć grupy, aby skończyć z dobrą złożonością czasową. Odpowiedź jest dość prosta: możemy przepisywać mniejszą grupę do większej! Za każdym razem, gdy musimy połączyć dwie grupy, iterujemy się po wszystkich wierzchołkach, należących do mniejszej z nich oraz po wszystkich krawędziach, które do nich wchodzą. Dzięki temu możemy łatwo aktualizować wszystkie wspomniane wartości i wykrywać kiedy należy połączyć jakieś grupy. Złożoność czasowa będzie taka, jak zawsze gdy korzystamy z przepisywania mniejszej grupy do większej – $\mathcal{O}(m \cdot \log(n))$ (gdzie m jest liczbą krawędzi w grafie). Podczas implementowania, warto rozważyć niereprezentowania swojej grupy przez jej głowę, co może ułatwić przepisywanie. Koniecznie jednak musimy pamiętać, który wierzchołek jest czołem grupy.

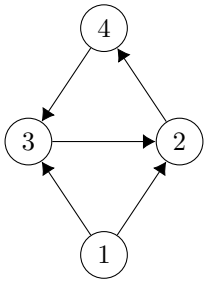
W momencie, w którym nie musimy już łączyć żadnych grup, warto zastanowić się, jak jakaś grupa może wyglądać. Nie może być dobrych par wierzchołków, w których oba wierzchołki należą do różnych grup, więc możemy traktować je kompletnie niezależnie. Ustalmy zatem jedną, konkretną grupę, i nie przejmujmy się innymi.

Na razie zapomnijmy o krawędziach wychodzących z czoła. Wiemy, że każda ścieżka z każdego wierzchołka w grupie prowadzi w końcu do czoła. Dla każdego z nich ustalimy pierwszy taki wierzchołek, przez który na pewno prowadzi każda ścieżka wychodząca z niego (i nazwiemy go $next[v]$). Zauważmy, że jest to jeden konkretny wierzchołek, ponieważ jeśli startując z wierzchołka v na pewno przejdziemy przez wierzchołek u oraz na pewno przejdziemy przez wierzchołek w, to albo startując z u przejdziemy przez w, albo startując z w przejdziemy przez u, ale nie mogą zachodzić oba. Dzieje się tak, ponieważ grupa jest acykliczna. Z tego samego względu istnieje dla niej porządek topologiczny. Znajdźmy go zatem (lub zapamiętajmy go podczas łączenia grup). Rozważajmy wierzchołki w grupie (poza czołem) w kolejności narzuconej przez porządek topologiczny, zaczynając od tych najbliżej czoła. Powiedzmy, że rozważamy teraz wierzchołek v. Prosto przekonać się, że graf wyznaczany przez wartości $next[u]$ dla już rozważonych wierzchołków u, jest skierowanym drzewem ukorzenionym w czole grupy. Nieco trudniej przekonać się, że wartością $next[v]$ będzie najniższy wspólny przodek wszystkich wierzchołków u, dla których istnieją krawędzie $v \to u$, we wcześniej wspomnianym drzewie. Bardziej doświadczonych zawodników do uświadomienia sobie tego może przekonać fakt, że podzadanie, które właśnie rozwiązujemy, to szukanie dominatorów w grafie skierowanym. Powinniśmy zatem, w trakcie budowania drzewa, dla każdego dodawanego do niego wierzchołka, obliczać dla niego jump pointery, które pozwolą efektywnie znajdować najniższego wspólnego przodka.

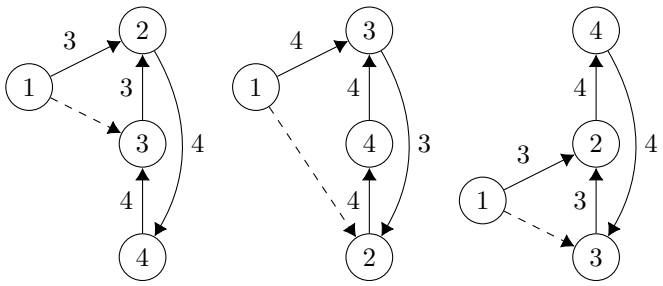
Musimy rozważyć teraz dwa przypadki: albo z czoła grupy prowadzi jakakolwiek krawędź poza nią, albo wszystkie krawędzie wychodzące z niego prowadzą z powrotem do niej. W tym pierwszym para wierzchołków (v, u) jest dobra wtedy i tylko wtedy, gdy u jest przodkiem v w drzewie wyznaczanym przez wartości $next$. W tym drugim wiemy, że po wyjściu z czoła na pewno wrócimy do tej samej grupy, więc być może istnieją też dobre pary wierzchołków, w których czoło jest pierwszym wierzchołkiem. Pierwszy wierzchołek, który na pewno napotkamy wychodząc z czoła, to znowu LCA wszystkich wierzchołków, do których prowadzą krawędzie wychodzące z czoła. Jeśli zatem ustawimy taką wartość $next[c]$, gdzie c jest czołem, to skierowane drzewo zamieni się w skierowaną meduzę. Dla niej również dość prosto jest obliczyć liczbę par wierzchołków takich, że z pierwszego istnieje ścieżka do drugiego.

Umiemy zatem w złożoności $\mathcal{O}(m \cdot \log(n))$ obliczyć wynik dla $k = n$, ale nie jest to jednak koniec zadania. Spójrzmy na jakiś wierzchołek v i na jego wartość $next[v]$. To, co chcielibyśmy zrobić, to znaleźć wierzchołek o największym indeksie, który leży na jakiejkolwiek ścieżce z v do $next[v]$ (wliczając w to v oraz $next[v]$). W tym celu możemy zmodyfikować działanie wcześniej wspomnianych jump pointerów. Skoro chcemy, by każda krawędź miała wagę, to owe jump pointery zamiast trzymać jedynie informację „w jakim wierzchołku wylądujemy skacząc o 2^p w górę", trzymać też informację „jakie jest maksimum z wag na krawędziach, które miniemy skacząc w ten sposób". W ten sposób w przypadku, gdy grupa jest drzewem, dla każdej dobrej pary aren (v, u), zacznie ona liczyć się do wyniku, gdy k będzie co najmniej takie, jak maksymalna wartość na ścieżce między nimi. Jest tak, ponieważ drugi gracz może wybrać owy wierzchołek o maksymalnym indeksie i prowadzić nas jedynie do niego, a jeśli k jest mniejsze niż indeks tego wierzchołka, to nie będziemy mogli w niego kliknąć.

Jeśli zaś grupa jest meduzą, to chciałoby się powiedzieć coś bardzo podobnego. Dokładniej, że dla dobrej pary wierzchołków (v, u) zacznie się ona liczyć do wyniku od chwili równej maksymalnej wadze krawędzi na ścieżce od v do u w meduzie. Niestety, pojawia się pewien problem. Spójrzmy na poniższy przykład:



W zależności od tego, czy czołem zostanie wierzchołek 2, 3 czy 4, możemy w trakcie obliczania meduzy otrzymać różne drzewa (i później meduzy) wyznaczane przez wartości *next*:



Jak łatwo zauważyć, środkowa meduza jest niepoprawna i opisana wyżej metoda liczenia nie zwróci poprawnego wyniku. Dzieje się tak, ponieważ dla wierzchołka nieleżącego na cyklu nie musimy dokładnie wiedzieć, na który wierzchołek na cyklu musimy natrafić najpierw. Innymi słowy, musimy natrafić na każdy wierzchołek na cyklu, ale nie musimy mieć pewności co do tego, na który natrafimy jako pierwszy. Co jeśli natrafimy na złą meduzę i jak w ogóle to wykryć? Po pierwsze zauważmy, że sam cykl nigdy się nie zmieni – zawsze kolejność wierzchołków oraz wagi krawędzi na nim pozostaną takie same – dla każdego wierzchołka na cyklu wiemy, jaki inny wierzchołek z cyklu musimy napotkać jako pierwszy. Być może więc jesteśmy w stanie ustanowić inny wierzchołek leżący na cyklu czołem i przeliczyć całą meduzę od nowa? Okazuje się, że tak! Poprawnym czołem musi być wierzchołek leżący na cyklu, z którego wychodzi krawędź o maksymalnej wadze (spośród krawędzi leżących na cyklu w dowolnie policzonej meduzie).

Dlaczego tak jest? Wybierzmy jakikolwiek wierzchołek należący do meduzy, który nie leży na cyklu, ale krawędź bezpośrednio z niego wchodzi w wierzchołek leżący na cyklu. Nazwijmy go v. Nazwijmy też kolejne wierzchołki leżące na cyklu $c_1, c_2, \ldots, c_d$, gdzie c_d jest czołem. Oznaczmy też wagę krawędzi prowadzącej z c_i do c_{i+1} (lub z c_d do c_1) jako w_i. Powiedzmy, że znamy rosnący ciąg indeksów $j_1, j_2, \ldots, j_p$ taki, że wierzchołki $c_{j_1}, c_{j_2}, \ldots, c_{j_p}$ mogą być pierwszymi wierzchołkami leżącymi na cyklu, które napotkamy po wyjściu z v. Dodatkowo powiedzmy, że maksymalnym indeksem wierzchołka, który musimy minąć, aby dojść do cyklu (czyli zalicza się do tego również v oraz każdy z wierzchołków $c_{j_1}, c_{j_2}, \ldots, c_{j_p}$), jest q. Jeśli czołem ustanowiliśmy właśnie wierzchołek c_d, to wagą krawędzi, która w meduzie prowadzi z v do c_{j_p}, jest większa z dwóch wartości: q oraz maksimum z wag $w_{j_1}, w_{j_1+1}, \ldots, w_{j_p-2}, w_{j_p-1}$. Okazuje się, że to dobrze! Aby dojść do wierzchołka c_{j_p} możemy być zmuszeni przejść przez wierzchołek o właśnie takim indeksie, za to nie możemy być zmuszeni przejść przez nic innego. Aby dojść do c_{j_i} dla $i \neq p$, możemy po pierwsze być zmuszeni przejść przez wierzchołek q, ale również by przejść przez krawędź o wadze w_d (ponieważ możemy być zmuszeni wejść na cykl w wierzchołku c_{j_p}). Jako, że waga w_d jest największa ze wszystkich, to uwzględnione wartości są poprawne.

Zastanowienie się czemu uwzględnione maksimum będzie poprawne dla wierzchołków nienależących do $c_{j_1}, c_{j_2}, \ldots, c_{j_p}$ oraz dokończenie dowodu pozostawiamy jako ćwiczenie dla czytelnika.

Zostajemy zatem z następującym zadaniem: mając dane skierowane drzewo z wagami na krawędziach lub skierowaną meduzę z wagami na krawędziach, należy dla każdej liczby powiedzieć, ile jest par wierzchołków takich, że maksimum na ścieżce między tymi wierzchołkami jest równe właśnie tej liczbie. Sumy prefiksowe, użyte na wspomnianym ciągu, będą ostatecznym wynikiem w zadaniu. Możemy oczywiście łatwo sprowadzić przypadek drzewa do przypadku meduzy, poprzez dodanie cyklu długości 1 w korzeniu drzewa. To zadanie jest dość standardowym ćwiczeniem na pracę z drzewami oraz meduzami i da się je rozwiązać na wiele sposobów, jednak przedstawimy jedno z podejść przyjętych przez autorów zadania.

Meduzę możemy potraktować jako cykliczny ciąg ukorzenionych drzew, gdzie każdy korzeń ma wychodzącą skierowaną krawędź do korzenia następnego drzewa. Zacznijmy od policzenia wyniku dla par wierzchołków należących do tego samego drzewa. Narzucającą się techniką jest użycie dekompozycji centroidowej. Znajdźmy centroid drzewa i policzmy maksimum na wszystkich ścieżkach przechodzących przez niego. Oczywiście wszystkie ścieżki, które uwzględnimy, muszą prowadzić od jakiegoś wierzchołka do jego przodka – w końcu drzewo jest ukorzenione. Mając w ręku centroid, możemy policzyć maksima na ścieżkach od niego do każdego z jego potomków (będących w tym samym wywołaniu rekurencyjnym dekompozycji centroidowej) oraz maksima na ścieżkach od niego do każdego z jego przodków (również zawartych w tym samym wywołaniu rekurencyjnym). Mając dwie listy liczb, chcemy policzyć dla każdej wartości, dla ilu różnych par występuje ona jako max(liczba z pierwszej listy, liczba z drugiej listy). Łatwo to zrobić sortując obie listy oraz używając gąsieniczki.

Następnie, dla każdego drzewa, policzmy listę maksimów na ścieżkach od korzenia tego drzewa do każdego wierzchołka w nim. Możemy teraz użyć techniki dziel i zwyciężaj na ciągu drzew. Znając przedział drzew, chcemy podzielić go na pół i rekurencyjnie policzyć wyniki w obu z nich. Brakującym składnikiem jest to, by mając dane rozłączne przedziały $[a_1, b_1]$ oraz $[a_2, b_2]$, policzyć wyniki dla ścieżek zaczynających się w drzewach z przedziału $[a_1, b_1]$, a kończących w przedziale $[a_2, b_2]$, oraz policzyć wyniki dla ścieżek zaczynających się w drzewach z przedziału $[a_2, b_2]$, a kończących w przedziale $[a_1, b_1]$. Oba podproblemy również możemy łatwo rozwiązać używając odpowiednich gąsieniczek, jednak pozwolimy sobie pominąć dokładne szczegóły techniczne.

Opisanej procedury używamy oczywiście na każdej grupie w całym grafie. Używając technik opisanych wyżej, otrzymamy finalną złożoność $\mathcal{O}(m \cdot \log^2(n))$, która nie powinna mieć problemów z uzyskaniem kompletu punktów. Starannie unikając w niej sortowania, lub wybierając nieco inne sposoby radzenia sobie z opisanym podzadaniem na meduzie, możemy skończyć również z algorytmem działającym w złożoności $\mathcal{O}(m \cdot \log(n))$.

5C – Butelki

autor zadania: Mateusz Radecki

Treść

Dane są pojemności trzech butelek oraz objętości płynów znajdujących się w nich na starcie. W jednym ruchu możemy wybrać dwie butelki i przelewać płyn z pierwszej do drugiej tak długo, aż w pierwszej skończy się płyn, lub w drugiej skończy się miejsce, którekolwiek pierwsze się stanie. Dla każdej objętości należy stwierdzić, po jakiej minimalnej liczbie przelań możemy mieć butelkę, w której znajduje się dokładnie tyle płynu.

Rozwiązanie

Oznaczmy pojemności butelek przez A, B i C, gdzie $A \leq B \leq C$. Zauważmy, że sumaryczna objętość płynu we wszystkich trzech butelkach jest stała. Zastanówmy się, co możemy wywnioskować z faktu, że wykonaliśmy chociaż jedno przelanie. Oznacza to, że przynajmniej jedna z butelek jest pełna lub pusta. Mamy zatem 6 możliwych przypadków. Zauważmy, że w każdym z nich, znamy dokładną sumaryczną objętość w płynu w pozostałych dwóch butelkach. Powiedzmy, że pojemności tych butelek to X oraz Y, a sumaryczna objętość płynu w nich to s. Ile istnieje sposobów na rozmieszczenie tylu litrów płynu w obu butelkach z założeniem, że liczba litrów w każdej butelce musi być liczbą całkowitą? Dokładny wzór mógłby nieco odstraszać, jednak ważnym jest, że jest to co najwyżej $X + 1$. Oznacza to, że możliwych kombinacji mówiących ile litrów płynu jest w danym momencie w każdej butelce, jest co najwyżej liniowo wiele! Możemy zatem użyć algorytmu BFS, gdzie stanem jest liczba litrów płynu w każdej butelce. Nie musimy się martwić o odpowiednią reprezentację stanu, czyli opisywanie go poprzez mówienie która butelka jest pusta lub pełna i tym podobne. Wiedząc to, co wcześniej wykazaliśmy wiemy, że algorytm BFS odwiedzi co najwyżej liniowo wiele różnych stanów. Używając odpowiedniej struktury danych (na przykład haszmapy), możemy spamiętywać w których stanach już byliśmy i wykonać poprawne przeszukiwanie grafu wszerz. Znając liczbę ruchów potrzebnych na dojście do każdego osiągalnego stanu, znamy również liczbę ruchów potrzebnych do otrzymania butelki z konkretną liczbą litrów płynu w środku.

Zależnie od wybranej struktury danych nasz algorytm BFS może działać w złożoności $\mathcal{O}(C \cdot \log(C))$ lub nawet $\mathcal{O}(C)$.

5C – Drzewo czerwono-czarne

autor zadania: Mateusz Radecki

Treść

Dane jest drzewo, którego wierzchołki pomalowane są na dwa kolory – czerwony i czarny. Dodatkowo, dla każdego wierzchołka wiemy, jakiego koloru ma on być finalnie. W jednym ruchu możemy wybrać dwa wierzchołki połączone krawędzią, i przemalować dowolny z nich na kolor drugiego. Należy stwierdzić, czy wykonując dowolny (być może pusty) ciąg ruchów, jesteśmy w stanie doprowadzić drzewo do docelowego stanu.

Rozwiązanie

Zacznijmy od rozpatrzenia najprostszych przypadków.

- Jeśli początkowy i finalny stan drzewa są identyczne, to odpowiedź oczywiście jest twierdząca – nie musimy nic robić.
- Jeśli początkowy stan zawiera tylko wierzchołki jednego koloru, to ponieważ finalny nie jest identyczny, to wymaga pojawienia się innego koloru, czego nie możemy zrobić – odpowiedź jest zatem przecząca.
- Jeśli w finalnym stanie każda para sąsiadujących wierzchołków jest różnych kolorów, to odpowiedź jest przecząca – nie jesteśmy w stanie wskazać pary wierzchołków, na której mogliśmy wykonać ostatni ruch, ponieważ taka para wierzchołków musi być tego samego koloru.
- Jeśli drzewo zawiera wierzchołek o stopniu co najmniej 3, to odpowiedź jest twierdząca.

Dowód ostatniego przypadku jest nieco skomplikowany. Powiedzmy, że wierzchołek o stopniu co najmniej 3 to t, a trójka jakichś jego sąsiadów to v, u i w. Niech pierwszą czynnością, którą wykonamy, będzie sprawienie, żeby nie wszystkie z wierzchołków v, u, w miały ten sam kolor. Jest to bardzo łatwo do osiągnięcia. Jeśli na początku wszystkie mają ten sam kolor, to „przyciągamy" drugi kolor z dowolnego innego wierzchołka do najbliższego mu z tych trzech wierzchołków.

Jeśli gdzieś w drzewie istnieje para wierzchołków (x, y) połączonych krawędzią taka, że oba te wierzchołki mają finalnie być tego samego koloru, oraz $x \neq t$ i $y \neq t$, to załóżmy bez straty ogólności, że wierzchołkiem bliższym do t jest wierzchołek x, a ścieżka z t do x przechodzi przez v. Oznaczmy tę ścieżkę przez $c_1, c_2, \ldots, c_d$, gdzie $c_1 = t$, $c_2 = v$ i $c_d = x$. Oznaczmy początkowy kolor wierzchołka i przez s_i, a docelowy kolor wierzchołka i przez k_i. Zamiast uzyskać zadany układ, uzyskamy układ w którym wierzchołek c_i malujemy na kolor $k_{c_{i-1}}$ dla każdego i z przedziału $[2, d]$. Sprawi to, że wśród wierzchołków $\{t, v, u, w\}$ będą istnieć dwa wierzchołki połączone krawędzią, które powinny finalnie mieć ten sam kolor. Zauważmy, że jeśli nie jesteśmy w stanie znaleźć takiej pary wierzchołków (x, y), to znaczy, że już na początku wśród wierzchołków $\{t, v, u, w\}$ istnieją dwa połączone krawędzią wierzchołki wymagające tego samego koloru i możemy ominąć cały ten etap. Jak na koniec, z osiągniętego układu, uzyskać docelowy układ? Wystarczy kolejno, dla każdego i od 2 do $d-1$, przemalować wierzchołek c_i na kolor wierzchołka c_{i+1}, a na koniec przemalować x na kolor wierzchołka y. Łatwo zauważyć, że da nam to docelowe rozwiązanie. Możemy zatem od tego momentu zakładać, że finalnie wierzchołek t oraz wierzchołek v mają mieć ten sam kolor.

Możemy rozwiązać aktualne zadanie malując odpowiednim kolorem kolejne liście drzewa i zapominając o nich, aż zostaną w nim jedynie wierzchołki t, v, u i w. Zauważmy, że łatwo jest „przeciągnąć" kolor z wierzchołków t, v, u i w do dowolnego innego wierzchołka nie tracąc faktu, że wśród tych wierzchołków znajdują się oba kolory.

Na koniec musimy doprowadzić do sytuacji, w której każdy z wierzchołków t, v, u i w ma odpowiedni kolor, przy czym wiemy, że wierzchołki t oraz v muszą mieć ten sam kolor. Tutaj również łatwo jest się przekonać, że zawsze da się to zrobić, zostawiamy to więc jako ćwiczenie dla czytelnika.

Powyższe kończy przypadek, gdy w drzewie istnieje wierzchołek o stopniu co najmniej 3. Co jeśli tak nie jest? Oznacza to, że drzewo jest ścieżką. Patrząc na ścieżkę jesteśmy w stanie podzielić ją na spójne przedziały koloru czerwonego oraz czarnego. Taki ciąg kolorów ma jakąś długość oraz kolor od którego się zaczyna. Oczywiście oba kolory w takim ciągu występują na zmianę. Możemy policzyć taki ciąg zarówno dla układu początkowego, jak i końcowego. Korzystając z faktu, że w układzie końcowym istnieją co najmniej dwa sąsiednie wierzchołki tego samego koloru, możemy poczynić ważną obserwację: jeżeli oba ciągi kolorów są równe, to da się przekształcić jeden układ w drugi! Wykazanie tego faktu również nie jest trudne i zostawiamy je jako ćwiczenie.

Co jednak, gdy ciągi nie są równe? Zastanówmy się, jak może się zmieniać ciąg kolorów odpowiadający początkowemu układowi. Przedziały tych samych kolorów mogą się łączyć oraz skrajne przedziały mogą znikać. Oznacza to, że możliwymi ruchami są: usunięcie pierwszego elementu ciągu; usunięcie ostatniego elementu ciągu; usunięcie dwóch sąsiednich elementów ciągu. Dość łatwo jest zatem ustalić warunki na to, że ciąg początkowy da się przekształcić w końcowy. Jeśli ciągi zaczynają się od tego samego koloru, to ciąg startowy musi być co najmniej tak długi, jak końcowy. Jeśli zaś zaczynają się od różnych kolorów, to ciąg początkowy musi być ściśle dłuższy niż końcowy. Możemy zatem wyznaczyć początkowy oraz końcowy ciąg dla drzewa i sprawdzić, czy spełniają one odpowiednie nierówności.

Całe zadanie opiera się zatem na sprawdzeniu kilku prostych warunków, co oczywiście możemy zrobić w czasie $\mathcal{O}(n)$.

5B – Autostrada

autor zadania: Kamil Dębowski

Treść

Dany jest opis trzypasmowej autostrady. Dla każdego pasa znamy pozycje oraz prędkość samochodów, które się na nim znajdują. Oznaczamy te prędkości przez v_1, v_2 oraz v_3, dla odpowiednio pierwszego, drugiego oraz trzeciego pasa. Znamy również maksymalną prędkość v_0 naszego samochodu. Jego prędkość możemy dowolnie zmieniać, jednak musi ona pozostać w przedziale $[0, v_0]$. Możemy również w pomijalnym czasie zmieniać pas ruchu na sąsiedni. Wszystkie samochody, włącznie z naszym, mają tę samą długość. Odległości między samochodami w jednym pasie są wielokrotnościami długości samochodu. Oczywiście nie możemy zderzyć się z żadnym innym samochodem. Jesteśmy pytani o pierwszy moment, w którym możemy wyprzedzić wszystkie samochody.

Rozwiązanie

Do rozwiązania zadania użyjemy programowania dynamicznego. Dla każdego pasa będziemy określać pozycję względem samochodów, które na nim jadą, a nie względem początku trasy. Będziemy stopniowo zwiększać czas t, zaczynając z $t = 0$. Dla każdego pasa będziemy pamiętać najbardziej wysuniętą na prawo pozycję na tym pasie, na której możemy się znaleźć w czasie t. Oznaczmy te pozycje przez p_1, p_2 oraz p_3. Zwróćmy uwagę, że wartości te nie muszą być całkowitoliczbowe.

Każda z tych wartości może zmieniać się z pochodną $v_0 - v_i$ albo z pochodną 0 względem czasu. Oznacza to, że na każdym pasie albo jedziemy ze swoją maksymalną prędkością (i przez czas t' zmieniamy swoją pozycję o $t' \cdot (v_0 - v_i)$) albo jedziemy równo z innymi samochodami, ponieważ nie możemy wjechać w samochód jadący bezpośrednio przed nami. Musimy zatem umieć wykrywać kiedy będziemy musieli zwolnić ze względu na następny samochód przed nami.

Inną rzeczą, którą musimy umieć wykrywać, jest możliwość zmiany pasa. Jeśli dla uporządkowanej pary sąsiednich pasów (i, j) na pasie i możemy być dalej niż na pasie j (względem początku trasy) i możemy zjechać z pasa i na pas j, to musimy rozważyć tę opcję. W celu aktualizacji czasu t oraz wartości p_i wybieramy oczywiście wydarzenie, które nastąpi jako pierwsze.

W celu wykrywania momentów, które musimy wziąć pod uwagę, możemy użyć odpowiednich struktur danych, odpowiednio dostosowanego wyszukiwania binarnego, lub jeśli jesteśmy wystarczająco ostrożni, znaleźć je w amortyzowanym czasie liniowym.

Gdy na jakimś pasie znajdziemy się przed wszystkimi innymi samochodami, opłacalnym jest do końca jechać ze swoją maksymalną prędkością aż wyprzedzimy wszystkie inne samochody. Wtedy znamy dokładny wynik.

Alternatywne podejście płynie z zauważenia, że w miarę opłaca nam się trzymać środkowego pasa i jechać na nim do przodu. Jeśli jesteśmy na środkowym pasie i możemy przejechać o pole do przodu, to robimy to z naszą maksymalną prędkością. W przeciwnym wypadku chcemy dostać się na następne wolne pole na środkowym pasie. Mamy tutaj dwie możliwości – użyć do tego pierwszego lub trzeciego pasa. Dla każdego z nich możemy wykryć najbliższy moment, w którym jest to możliwe. W przypadku pierwszego – szybszego pasa, interesuje nas jedynie czy możemy od razu na niego zjechać. Jeśli tak, natychmiastowo to robimy i jedziemy nim (albo z naszą maksymalną prędkością albo tuż za samochodem z jego prędkością) tak długo, aż będziemy mogli zjechać na środkowy pas. W przypadku trzeciego – wolniejszego pasa, interesuje nas najbliższy przedział przed nami, którego długość pozwoli nam dotrzeć do następnego wolnego pola na środkowym pasie – zakładając, że przejedziemy ten przedział z naszą maksymalną prędkością.

W przypadku pierwszego pasa wygodnie jest użyć wyszukiwania binarnego, a w przypadku trzeciego pasa – drzewa przedziałowego. Również w tym rozwiązaniu dostateczna ostrożność pozwala obyć się bez struktur danych i rozwiązać zadanie w amortyzowanym czasie liniowym.

By nie musieć walczyć z dokładnością obliczeń zmiennoprzecinkowych, warto zauważyć, że wszystkie interesujące wydarzenia mają miejsca w momentach czasu, które po zapisaniu jako liczba wymierna mają mianownik będący dzielnikiem $(v_0-v_1)(v_0-v_2)(v_0-v_3)(v_1-v_2)(v_1-v_3)(v_2-v_3)$. Oznaczmy wspomniany iloczyn przez M. Warto zatem nie trzymać w ogóle liczb zmiennoprzecinkowych, a pomnożyć liczony czas przez M. Alternatywnie, można trzymać momenty w czasie jako ułamki o mianowniku równym M. Ze względu na małe limity na wartości v_i nie powinno to sprawiać problemów ze zmieszczeniem się w 64-bitowych typach zmiennych.

Jeśli poprawnie zaimplementowane, otrzymujemy rozwiązanie działające w złożoności $\mathcal{O}(L \cdot \log(L))$ lub nawet $\mathcal{O}(L)$.

5B – Desant 2

autor zadania: Mateusz Radecki

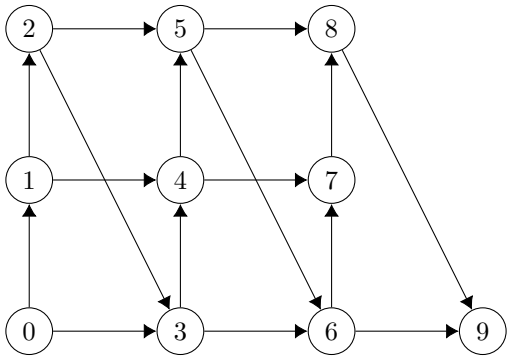
Treść

Dany jest ciąg liczb całkowitych $a_1, a_2, \ldots, a_n$ oraz liczba k. Naszym zadaniem jest odpowiadać na zapytania o podprzedziały zadanego ciągu. Dla każdego zapytania należy wyznaczyć maksymalną możliwą sumę liczb pokrytych rozłącznymi przedziałami długości k zawartymi w całości w podprzedziale z zapytania.

Rozwiązanie

Zinterpretujmy zadanie grafowo. Niech graf zawiera $n+1$ wierzchołków, ponumerowanych liczbami całkowitymi od 0 do n. Dla każdego i z przedziału $[0, n-1]$ dodajmy do niego krawędź skierowaną z wierzchołka i do wierzchołka $i+1$ o wadze 0. Dla każdego i z przedziału $[0, n-k]$ dodajmy krawędź skierowaną z wierzchołka i do wierzchołka $i+k$ o wadze będącej sumą wartości a_j dla j z przedziału $[i+1, i+k]$. Nietrudno zauważyć, że odpowiedzią na zapytanie o przedział $[\ell, r]$ jest długość najdłuższej ścieżki z wierzchołka $\ell-1$ do wierzchołka r.

Wyobraźmy sobie nasz graf narysowany na płaszczyźnie, gdzie wierzchołek i narysujemy w punkcie o współrzędnych $(\lfloor \frac{i}{k} \rfloor, i \mod k)$. Przykładowo, dla $n = 9$ oraz $k = 3$, nasz graf wygląda następująco:



Nasz graf ma bardzo regularny kształt i łatwo jest szukać w nim separatorów. Spróbujmy zatem, w pewien sposób używając na nim techniki dziel i zwyciężaj, policzyć odpowiedzi dla wszystkich zapytań.

Jeśli nasz graf jest bardziej szeroki niż wysoki (czyli liczba kolumn w kracie jest większa niż liczba wierszy) możemy skorzystać z faktu, że każde zapytanie albo ma jeden wierzchołek na lewo od środkowej kolumny, a drugi na prawo, albo w całości zawiera się po którejś ze stron. Jeśli zachodzi pierwszy przypadek, to na pewno optymalna ścieżka przechodzi przez środkową kolumnę, w której to znajduje się nie więcej niż $\sqrt{n}$ wierzchołków. Możemy zatem dla każdego z nich (nazwijmy go v) policzyć długości najdłuższych ścieżek zaczynających się w dowolnym wierzchołku i kończących w v, oraz długości najdłuższych ścieżek zaczynających się w v i kończących w dowolnym innym wierzchołku. Jeśli jakaś ścieżka nie istnieje, to jej długość traktujemy jako $-\infty$. Całość zajmie nam oczywiście czas $\mathcal{O}(n\sqrt{n})$. Następnie, możemy założyć dla każdego zapytania, że optymalna ścieżka przebiega właśnie przez v. Wtedy oczywiście długość ścieżki z wierzchołka startowego s do wierzchołka końcowego t wynosiłaby $dist(s, v) + dist(v, t)$. Jako, że każda rozważana ścieżka musi przechodzić przez któryś z wierzchołków w środkowej kolumnie, to na pewno rozważymy też optymalną z nich. Następnie, możemy rekurencyjnie wywołać się na lewej oraz prawej części kraty i tam niezależnie policzyć wyniki.

Jeśli zaś nasz graf jest bardziej wysoki niż szeroki, to możemy zrobić coś bardzo podobnego, jednak będąc bardziej uważnym. Gdyby nie było krawędzi z ostatniego rzędu do pierwszego, to moglibyśmy po prostu zrobić to samo dla środkowego wiersza. Jeśli jednak w naszym wywołaniu rekurencyjnym nadal dostaliśmy wszystkie wiersze, to musimy najpierw założyć, dla każdego zapytania, że optymalna ścieżka dla niego przechodzi przez najniższy rząd. Możemy w ten sposób w złożoności $\mathcal{O}(n\sqrt{n})$ zrelaksować wyniki dla wszystkich zapytań i wywołać się na kracie bez pierwszego rzędu. Nie podzieliliśmy wierszy na dwie grupy, ale na szczęście we wszystkich głębszych wywołaniach rekurencyjnych, gdy nasz graf będzie bardziej wysoki niż szeroki, będziemy już mogli rozbić się rekurencyjnie wokół środkowego rzędu.

Wybierając za każdym razem lepszy wymiar do rekurencyjnego rozbicia się, zgodnie z twierdzeniem mistrza, uzyskamy sumaryczną złożoność $\mathcal{O}(n\sqrt{n})$.

5A – Fiolki 2

autor zadania: Mateusz Radecki

Treść

Dany jest skierowany graf acykliczny o n wierzchołkach, w którym pierwsze k wierzchołków ma zerowy stopniu wejściowy. Dla każdego przedziału wierzchołków $[\ell, r]$, spełniającego $k < \ell \leq r \leq n$, należy policzyć maksymalną liczbę rozłącznych wierzchołkowo ścieżek, które mogą zaczynać się w pierwszych k wierzchołkach, a kończyć w wierzchołkach o indeksach w tym przedziale. Na wyjściu należy wypisać, dla każdego x z przedziału $[0, k]$, dla ilu przedziałów odpowiedź jest równa x.

Rozwiązanie

Dla każdego wierzchołka, dynamicznie, policzymy sobie pewien wektor rozmiaru k. Będziemy pracować w ciele $\mathbb{Z}_p$, najlepiej dla jakiejś dużej liczby pierwszej, w optymalnym przypadku większej niż 10^{18}. Dobrym pomysłem jest wybrać $p = 2^{61} - 1$, ze względu na to, że dzięki jej reprezentacji w systemie dwójkowym jesteśmy w stanie szybko wykonywać mnożenie dwóch liczb modulo owa liczba pierwsza. Dla i-tego z pierwszych k wierzchołków, ustalonym dla niego wektorem będzie wektor złożony z samych zer oraz jedynki na i-tej pozycji. Dla każdego wierzchołka o indeksie większym niż k, jego wektorem będzie suma wektorów wierzchołków do niego wchodzących, przemnożonych przez losowe wartości z przedziału $[1, p-1]$. Wszystkie wektory możemy obliczyć dynamicznie w złożoności $\mathcal{O}(mk)$. Po co nam one? Zaraz się przekonamy.

Wybierzmy pewne k wierzchołków z przedziału $[k+1, n]$ i zastanówmy się, czy możemy wybrać k rozłącznych ścieżek, zaczynających się w wierzchołkach $1, 2, \ldots, k$, a kończących się w wybranych wierzchołkach. Spójrzmy na ciąg wektorów tych wierzchołków. Łatwo zauważyć, korzystając z wiedzy o przepływach, że takie k ścieżek nie istnieje wtedy i tylko wtedy, gdy możemy zakazać co najwyżej $k-1$ wierzchołków, tak aby każda ścieżka z któregokolwiek z pierwszych k wierzchołków do któregokolwiek z wybranych k wierzchołków przechodziła przez któryś z zakazanych wierzchołków. W szczególności, możemy zakazywać również startowych oraz końcowych wierzchołków. Co jesteśmy wtedy w stanie powiedzieć o wektorach wybranych wierzchołków? Wszystkie są rozpinane przez co najwyżej $k-1$ wektorów zakazanych wierzchołków, a więc nie są one liniowo niezależne.

W tym momencie może pojawić się promyk nadziei, że w przeciwnym przypadku mamy duże prawdopodobieństwo (zależne od wybranej liczby pierwszej p), że owy zbiór wektorów będzie liniowo niezależny. Okazuje się być to prawdą! Co więcej, możemy posuwać obserwacje dalej. Jeśli minimalny rozmiar zbioru, który da się zakazać, aby przeciąć wszystkie ścieżki prowadzące z pierwszych k wierzchołków do wybranego zbioru wierzchołków wynosi q, to wszystkie wektory dla wybranego zbioru wierzchołków na pewno są rozpinane przez pewien zbiór q wektorów, więc rząd macierzy powstałej przez ułożenie obok siebie wektorów wybranych wierzchołków wynosi co najwyżej q. Okazuje się, że rzeczywiście ma on duże prawdopodobieństwo wynosić dokładnie q.

Pominiemy tutaj dowody, jeśli jednak ktoś jest zainteresowany, to polecamy zapoznać się z twierdzeniem Mengera oraz z lematem Schwartza-Zippela.

Zostajemy zatem z zadaniem, gdzie mamy daną macierz M o wymiarach $(n-k) \times k$ i dla każdego $x \in [0, k]$ mamy policzyć liczbę takich przedziałów wierszy tej macierzy, że podmacierz złożona jedynie z tych wierszy M ma rząd dokładnie x.

Zmniejszmy n o k, żeby n było od teraz równe liczbie wierszy w macierzy M. Przeiterujmy się zatem od lewej do prawej po prawym końcu rozważanego przedziału. Nazwijmy aktualnie rozważany prawy koniec r. Istnieje taki ciąg $\ell_1, \ell_2, \ell_3, \ldots$ długości co najwyżej k, że wszystkie jego elementy są dodatnie i zachodzi $r \geq \ell_1 > \ell_2 > \ell_3 \ldots$. Dodatkowo, przedział $[\ell_i, r]$ jest najkrótszym takim przedziałem kończącym się na r, że rząd podmacierzy wyznaczanej przez niego jest równy i. Jak zmienia się ciąg $\ell_1, \ell_2, \ell_3, \ldots$, gdy przechodzimy z $r-1$ do r? Jeśli r-ty wektor jest wektorem zerowym, to ciąg nie zmienia się wcale. W przeciwnym razie mamy dwa przypadki. Albo r-ty wektor jest liniowo niezależny ze wszystkimi wierszami o indeksach z ciągu $\ell_1, \ell_2, \ell_3, \ldots$, albo nie jest. Jeśli jest liniowo niezależny, to chcemy dodać r-tą pozycję do ciągu, jeśli nie jest, to możemy jedną z wartości w ciągu wymienić na r. Opłaca się wymienić najmniejszą wartość z tych, które wymienić możemy, ale jak ustalić, które z nich możemy wymienić, jest trudnym pytaniem.

Trzymajmy zatem częściowo zeschodkowaną macierz odpowiadającą wektorom na pozycjach z ciągu $\ell_1, \ell_2, \ell_3, \ldots$. Dodatkowo, dla każdego wiersza tej macierzy, trzymajmy informację jaką kombinacją liniową wektorów ze wspomnianych pozycji w macierzy M on jest. Dzięki temu, jeśli umiemy zapisać r-ty wektor jako kombinację liniową wierszy zeschodkowanej macierzy, to możemy też ustalić, jaką kombinacją liniową wierszy macierzy M on jest. Możemy wymienić go na te wektory, które pojawiają się w tej kombinacji z niezerowym współczynnikiem. Po takiej zamianie zeschodkowana macierz nie zmienia się, ale jesteśmy w stanie w czasie $\mathcal{O}(k^2)$ przeliczyć kombinacje liniowe wierszy macierzy M, które składają się na wiersze zeschodkowanej macierzy.

W ten sposób możemy rozwiązać całe zadanie w złożoności $\mathcal{O}(mk + nk^2)$.

5A – Zbiory niezależne

autor zadania: Marek Sokołowski

Treść

Mamy dane liczby ℓ, r oraz c. Naszym zadaniem jest policzyć ile istnieje różnych nieetykietowanych drzew, których wierzchołki są pomalowane na jeden z c kolorów i w których rozmiar maksymalnego zbioru niezależnego zawiera się w przedziale $[\ell, r]$. Wynik należy podać modulo 998 244 353.

Rozwiązanie

Wspomniana liczba pierwsza dla bardziej doświadczonych zawodników jest sygnałem, że będziemy intensywnie korzystać z szybkiej transformaty Fouriera, służącej do szybkiego mnożenia wielomianów. Mówiąc ściślej, będziemy korzystać z operacji na funkcjach tworzących: będziemy chcieli umieć je dodawać, mnożyć, ale także dla funkcji $F(x)$ będziemy chcieli umieć obliczać $\exp(F(x))$ i $\frac{1}{F(x)}$. Informacje na temat tego jak wykonywać te operacje w czasie $\mathcal{O}(n \cdot \log(n))$, gdzie n maksymalnym interesującym nas współczynnikiem, możliwe są do znalezienia w internecie.

Rozważmy najpierw zadanie, w którym chcemy liczyć drzewa o liczbie wierzchołków równej n, zamiast o rozmiarze zbioru niezależnego należącego do przedziału $[\ell, r]$, oraz interesują nas różne ukorzenione drzewa, a nie dowolne. Użyjmy zatem programowania dynamicznego: niech $L[i]$ będzie liczbą takich różnych nieetykietowanych ukorzenionych lasów, które zawierają dokładnie i wierzchołków. Niech $D[i]$ będzie liczbą ukorzenionych drzew o i wierzchołkach. $D[n]$ będzie zatem interesującym nas wynikiem. Programowanie dynamiczne inicjalizujemy oczywiście jako $L[0] = 1$. Zachodzi też własność $D[i] = c \cdot L[i-1]$ – drzewo możemy utożsamiać z lasem pozostałym po usunięciu jego korzenia, który to mógł mieć jeden z c kolorów. Warto zauważyć, że ciąg (L_n) możemy utożsamiać z jego funkcją tworzącą, którą będziemy oznaczać jako $L(x)$. Warto wprowadzić tutaj pomocnicze ciągi – niech $L_i(x)$ oznacza funkcję tworzącą dla ciągu, który oznacza liczbę lasów o odpowiedniej liczbie wierzchołków, w których każde drzewo ma rozmiar co najwyżej i. Zaczynamy oczywiście z $L_0(x) = 1$.

Jeśli poznamy wartość $D[i]$, to dość łatwo jest, znając również $L_{i-1}(x)$, obliczyć $L_i(x)$. Dla każdego drzewa liczonego w $D[i]$ możemy dowolną liczbę jego kopii dołożyć do dowolnego lasu. Dołożenie dowolnej liczby kopii drzewa o i wierzchołkach możemy wykonać poprzez przemnożenie przez $\sum_{j=0}^{\infty} x^{ij} = \frac{1}{1-x^i}$. Skoro dla każdego drzewa możemy niezależnie zdecydować o liczbie jego dołożonych kopii, to otrzymujemy równość $L_i(x) = L_{i-1}(x) \cdot \frac{1}{(1-x^i)^{D[i]}}$.

Łatwo się domyślić, że zamiast obliczać kolejne $L_i(x)$, będziemy utrzymywać jedną funkcję i ją aktualizować. Mnożenie jest jednak trudne i wolne, co możemy zatem zrobić zamiast niego? Chcąc finalnie obliczyć $\prod_{i=1}^{r} \frac{1}{(1-x^i)^{D[i]}}$, możemy zamiast tego obliczyć $\exp(\sum_{i=1}^{r} \log(\frac{1}{(1-x^i)^{D[i]}}))$. Dodawanie wydaje się już prostsze, jednak by upewnić się, że istnieje jakakolwiek nadzieja na poprawę czasową, należy spojrzeć na rozwinięcie Taylora dla funkcji $\log(\frac{1}{(1-x^i)^{D[i]}})$. Otrzymujemy:

$$\log\left(\frac{1}{(1-x^i)^{D[i]}}\right) = D[i] \cdot \sum_{j=1}^{\infty} \frac{x^{ij}}{j}$$

Jest to dla nas świetna wiadomość. W i-tym kroku, logarytm który dodajemy, jest nie tylko bardzo łatwy do obliczenia, ale również ma tylko $\lfloor \frac{r}{i} \rfloor$ niezerowych współczynników! Możemy zatem dodać wszystkie logarytmy w złożoności czasowej $\mathcal{O}(r \cdot \log(r))$, a następnie w tej samej złożoności przyłożyć do nich eksponentę. Jest jednak pewien problem. By obliczyć $D[i]$ musimy poznać $[x^{i-1}]L_{i-1}(x)$, a w tym celu, musielibyśmy dla każdego i po drodze obliczać eksponentę dość dużej funkcji tworzącej.

Potrzebujemy zatem sprytnego pomysłu. Powiedzmy, że obliczyliśmy już $\log(L_{2^k}(x))$ dla pewnego k. Możemy zatem pozwolić sobie na obliczenie $L_{2^k}(x)$. Spróbujmy obliczyć jednocześnie wszystkie wartości $D[i]$ dla i należącego do przedziału $[2^k + 1, 2^{k+1}]$. Jak możemy to zrobić? Oznaczmy:

$$P(x) = L_{2^k}(x) \mod x^{2^k} = L(x) \mod x^{2^k}$$

$$S(x) = \frac{L_{2^k}(x) - P(x)}{x^{2^k}} \mod x^{2^k}$$

$$W(x) = \frac{L_{2^{k+1}}(x) - P(x)}{x^{2^k}} \mod x^{2^k} = \frac{L(x) - P(x)}{x^{2^k}} \mod x^{2^k}$$

$P(x)$ i $W(x)$ odpowiadają zatem wszystkim lasom, których liczba wierzchołków mieści się w przedziałach odpowiednio $[0, 2^k)$ i $[2^k, 2^{k+1})$, zaś $S(x)$ odpowiada tym lasom, w których drzewa nie przekraczają rozmiarem 2^k. Korzystając z faktu, że w lesie o sumarycznej liczbie wierzchołków mniejszej niż 2^{k+1} zmieści się co najwyżej jedno drzewo o rozmiarze większym niż 2^k, możemy rozpisać równanie, dzięki któremu będziemy w stanie wyznaczyć $W(x)$:

$$W(x) = S(x) + P(x) \cdot W(x) \cdot x \cdot c \mod x^{2^k}$$

$$W(x) = \frac{S(x)}{1 - P(x) \cdot x \cdot c} \mod x^{2^k}$$

Umiejąc wykonywać operacje na funkcjach tworzących możemy zatem obliczyć $W(x)$, dzięki czemu będziemy w stanie poznać $D[i]$ dla wszystkich i z przedziału $[2^k + 1, 2^{k+1}]$. Dla każdego z nich będziemy zatem mogli zaktualizować $L(x)$ o odpowiedni logarytm.

Ponieważ obliczenie kolejnych wartości wykonujemy w czasie $\mathcal{O}(2^k \cdot k)$, to obliczenie liczby drzew ukorzenionych o n wierzchołkach dla każdego n nie większego niż r możemy wykonać w czasie $\mathcal{O}(r \cdot \log(r))$. Pozostają dwa problemy: interesowały nas rozmiary zbiorów niezależnych, oraz drzewa nie miały być ukorzenione.

Z pierwszym problemem możemy poradzić sobie nieco komplikując obliczenia. Wiemy, że dla ukorzenionego drzewa istnieje algorytm zachłanny, który oblicza rozmiar maksymalnego zbioru niezależnego – wierzchołek należy do zbioru niezależnego wtedy i tylko wtedy, gdy żadnego jego dziecko do niego nie należy.

Zamiast funkcji tworzącej $L(x)$, możemy zdefiniować funkcje tworzące $A(x)$ oraz $B(x)$. $[x^i]A(x)$ oznacza liczbę takich ukorzenionych lasów o sumarycznym zbiorze niezależnym rozmiaru i, w których żaden korzeń nie został wzięty do zbioru niezależnego przez algorytm zachłanny. $[x^i]B(x)$ oznacza liczbę wszystkich ukorzenionych lasów o sumarycznym zbiorze niezależnym rozmiaru i. Analogicznie do $L_i(x)$ możemy zdefiniować $A_i(x)$ i $B_i(x)$ – interesują nas jedynie ukorzenionych lasy, w których żadne drzewo nie ma zbioru niezależnego o rozmiarze większym niż i. Okazuje się, że analogicznie do przypadku z ograniczeniem na liczbę wierzchołków, możemy znając $A_{2^k}(x)$ i $B_{2^k}(x)$ obliczyć $A_{2^{k+1}}(x)$ i $B_{2^{k+1}}(x)$ w czasie $\mathcal{O}(2^k \cdot k)$. Przekształcenia działają na tej samej zasadzie, jednak oczywiście są one bardziej skomplikowane, gdyż musimy rozwiązać układ dwóch równań z dwiema niewiadomymi. Zabawę z dokładnym przekształceniem wzorów, postanowiliśmy pozostawić jako ćwiczenie dla czytelnika.

Z drugim problemem, którym jest obliczenie liczby nieukorzenionych drzew, a nie ukorzenionych, pomoże nam pewien niesławny wzór.

Obserwacja (Otter's formula). *Ustalmy nieukorzenione nieetykietowane drzewo T. Niech v będzie liczbą różnych drzew ukorzenionych w wierzchołku i izomorficznych z T. Niech e będzie liczbą różnych drzew ukorzenionych w krawędzi i izomorficznych z T. Niech s będzie równe* 1*, jeśli T jest symetryczne względem którejkolwiek krawędzi, a* 0 *w przeciwnym przypadku. Wtedy $v - e + s = 1$.*

Korzystając z $A(x)$ oraz $B(x)$ obliczymy $G(x)$ oraz $H(x)$. Niech $[x^i]G(x)$ oznacza liczbę ukorzenionych drzew o rozmiarze zbioru niezależnego równym i, w których korzeń nie został wzięty przez algorytm zachłanny do zbioru niezależnego. Niech $[x^i]H(x)$ oznacza liczbę ukorzenionych drzew o rozmiarze zbioru niezależnego równym i, w których korzeń został wzięty przez algorytm zachłanny do zbioru niezależnego. Niech $[x^i]T(x)$ oznacza liczbę nieukorzenionych drzew o rozmiarze zbioru niezależnego równym i. Sumując powyższy wzór Ottera po wszystkich nieetykietowanych drzewach otrzymujemy:

$$T(x) = G(x) + H(x) - \frac{1}{2}G(x)^2 + \frac{1}{2}G(x^2) - G(x)H(x) - \frac{1}{2x}H(x)^2 + \frac{1}{2x}H(x^2)$$

$T(x)$ okazuje się być dokładnie funkcją tworzącą ciągu, o który jesteśmy pytani w zadaniu! Znając dla każdego n nie większego niż r liczbę drzew o rozmiarze zbioru niezależnego równym n, możemy po prostu zsumować te wartości dla wszystkich n z przedziału $[\ell, r]$. Otrzymujemy dzięki temu rozwiązanie działające w złożoności $\mathcal{O}(r \cdot \log(r))$ z ogromną stałą.