,

Розбір задачі «Пироголяндія»

Пироголяндія являє собою дерево, вершинами якого є пекарні, а ребрами є дороги. Тоді кожна вершина містить якесь невід'ємне число(значення вершини), а кожне ребро має один з двох станів: заблоковане чи розблоковане. Будемо вважати, що дерево має корінь у вершині 1.

Блоки 1-5:

- Запит типу 1: збережемо масив що відповідає за стан кожного ребра. При виклику запита будемо змінювати стан ребра у цьому масиві.
- Запит типу 2: запустимо DFS із вершини p, та відвідаємо усі вершини які знаходяться у компоненті цієї вершини. При цьому не будемо переходити по заблокованим ребрам. Змінимо значення відвіданих вершин.
- Запит типу 3: можна викликати спочатку запит типу 5 до вершини p, щоб дізнатись її значення після застосування запиту типу 3. Після цього викликаємо запит типу 6, щоб змінити значення кожної вершини з компоненти вершини p на нуль. Замінюємо значення вершини p на підраховану суму.
- Запит типу 4: повертаємо значення вершини p.
- Запит типу 5: запустимо DFS із вершини p, та відвідаємо усі вершини які знаходяться у компоненті цієї вершини. При цьому не будемо переходити по заблокованим ребрам. Додамо до відповіді значення відвіданих вершин.
- Запит типу 6: запустимо DFS із вершини p, та відвідаємо усі вершини які знаходяться у компоненті цієї вершини. При цьому не будемо переходити по заблокованим ребрам. Змінимо значення відвіданих вершин на нуль.
- Запит типу 7: візьмемо будь-яку вершину із ненульовим значенням. Якщо такої не існує, то очевидно що відповідь нуль. Інакше запустимо із неї DFS, що буде повертати 1, якщо у піддереві вершини з якої ми виходимо є хоча б одна ненульова вершина, та 0 інакше. Тоді, якщо ми переходимо до наступної вершини через заблоковане ребро, та DFS з неї повертає 1, то існує пара вершин(вершина з якої ми запустили DFS на початку, та певна вершина з цього піддерева), така що шлях між ними проходить через це заблоковане ребро. Отже, до відповіді слід додати 1.

Блок 6:

- під час ініціалізації розіб'ємо дерево на компоненти, що обмежені заблокованими ребрами, та дамо кожній компоненті свій номер. Оскільки запити типу 1 відсутні, то множина вершин кожної компоненти змінюватись не буде. Також, будемо зберігати значення кожної вершини не зовсім чесно: будемо зберігати її старе значення, а справжнє ми зможемо дізнатись застосувавши певні операції.
- для кожної компоненти будемо зберігати певні значення:
  1. integer sum — сума значень вершин компоненти.
  2. integer toAdd — значення яке треба додати до кожної вершини цієї компоненти, щоб воно було рівним справжньому значенню вершини.
  3. integer lastVertex — остання вершина компоненти до якої було застосовано запит типу 3, або нуль якщо до вершин цієї компоненти ще не застосовувались запити типу 3.
  4. clear — булева змінна, що рівна $true$, якщо до компоненти вже застосовували запит типу 3, або запит типу 6, та нуль інакше.
  5. cnt — кількість вершин у цій компоненті.
- запит типу 2: візьмемо номер компоненти у якій знаходиться вершина із номером p, та додамо до $toAdd$ значення w.
- запит типу 3: можна викликати спочатку запит типу 5 до вершини p, щоб дізнатись її значення після застосування запиту типу 3. Після цього викликаємо запит типу 6, щоб змінити значення кожної вершини з компоненти вершини p на нуль. Замінюємо значення вершини p на підраховану суму. Змінимо значення $lastVertex$.
- запит типу 4: оскільки кожна вершина зберігає своє старе значення, то треба до цього значення додати значення параметру $toAdd$ з її компоненти.
- запит типу 5: оскільки значення вершини рівне $a_p + toAdd(p)$ (де $toAdd(p)$ рівне значенню параметра $toAdd$ компоненти у якій знаходиться вершина із номером p), тоді суму значень вершин у компоненті можна визначити за формулою $sum + cnt \times toAdd$.
- запит типу 6: розглянемо два випадки залежно від значення параметру $clear$.
  1. $clear = false$: зробимо всі операції чесно: пройдемось по кожній вершині, замінимо її значення на нуль. Значення sum та $toAdd$ також замінимо на нуль, а значення $clear$ на $true$.
  2. $clear = true$: у нас є лише одна вершина, нечесне значення якої відмінне від нуля, і це вершина із номером $lastVertex$. Отже, можна зробити теж саме що у випадку $clear = false$, але замінити лише значення вершини із номером $lastVertex$.

Блок 7:

- щоб вирішити цей блок, можна модифікувати рішення до попереднього блоку. З'являється лише проблема із тим, що кількість та склад компонент можуть змінюватись.
- додамо до компонент два параметри:
  1. $allVertexes$ — вектор, що зберігає множину вершин, що знаходяться у цій компоненті.
  2. $activeVertexes$ — вектор, що зберігає множину вершин, що знаходяться у цій компоненті, та мають нечесне значення відмінне від нуля.
- через запит типу 1 треба вміти поєднувати певні компоненти. Для цього застосуємо структуру даних $DisjointSetUnion$ (система неперетинних множин).
- розглянемо операцію $unionSets$, застосовану до компонент із номерами x та y у нашій DSU. Вона повинна поєднати дві компоненти. Будемо вважати, що це лідери відповідних компонент у DSU та вони різні. Ми будемо підвішувати компоненту із номером y до компоненти із номером x. Тоді нам треба якось поєднати їх параметри. Для цього, спочатку очистимо список $activeVertexes(y)$, а потім давайте пройдемось по вершинам компоненти y (список $allVertexes(y)$), якщо значення $clear = true$, тоді спочатку замінимо значення вершини на нуль, а потім замінимо значення кожної з них на $a_p = a_p + toAdd(y) - toAdd(x)$, додамо кожну з них до списку $activeVertexes(y)$, а до $sum(y)$ додамо $cnt(y) \times toAdd(y)$. Далі, перекинемо вершини зі списку $allVertexes(y)$ до списку $allVertexes(x)$, а також вершини зі списку $activeVertexes(y)$ до списку $activeVertexes(x)$. Додамо до $sum(x)$ значення $sum(y)$, а до $cnt(x)$ значення $cnt(y)$. Підвісимо вершину y до вершини x у DSU.
- які зміни відбулися у запитах?
  - запит типу 2: змін немає, лише треба брати номер компоненти з DSU.
  - запит типу 3: після виклику запита типу 6, наш список $activeVertexes$ повинен містити лише вершини p, але він є порожнім, тому треба додати до нього вершину із номером p, якщо її значення після цього запиту є ненульовим.

  - запити типу 4 та 5: відповіді можна знайти так само як у блоці 6.
  - запит типу 6: у рішенні до шостого блоку ми фактично проходились по списку $allVertexes$, але помітимо, що можна проходитись лише по списку $activeVertexes$ та замінювати лише їх значення на нуль. Після цього слід видалити усі вершини зі списку $activeVertexes$.

- асимптотика такого рішення не буде вкладатись в обмеження, бо може перевищувати $O(N^2)$. Проте, можна застосувати техніку *small-to-large*(від меншого до більшого). Перші ніж оброблювати запит типу 1, давайте упорядкуємо x та y так, щоб $cnt(x)$ було не меншим за $cnt(y)$. Тоді асимптотика такого рішення вже буде $O(n\ log\ n)$, що повинно проходити даний блок.

Блок 8:

- оскільки компоненти не будуть змінюватись у складі, можна побудувати додаткове дерево. Для цього, давайте умовно видалимо із нашого дерева усі заблоковані ребра та стиснемо кожну компоненту що утворилась в одну вершину в новому дереві. Всі нові вершини ми якось пронумеруємо, а також для кожної вершини початкового дерева запам'ятаємо номер вершини до якої вона увійшла в новому дереві. Тепер додамо до цього дерева заблоковані ребра, що ми видалили на початку його побудови. Підвісимо це дерево за будь-яку вершину. Будемо вважати, що вершина в новому дереві має чорний колір, якщо існує хоча б одна вершина із ненульовим значенням у початковому дереві, що належить цій вершині в новому дереві. Переформулюємо запит сьомого типу: треба знайти мінімальну кількість ребер у новому дереві, які потрібно розблокувати, щоб на шляху між кожною парою чорних вершин усі ребра були розблоковані.
- Розглянемо будь-яке ребро з нового дерева. За якої умови його треба розблокувати? Якщо у піддереві нижньої вершини(більш віддаленої від кореня) цього ребра є хоча б одна вершина чорного кольору, та ззовні піддерева верхньої вершини є хоча б одна вершина чорного кольору, то це ребро треба розблокувати. Інакше не існує пари вершин чорного кольору, на шляху між якими лежить це ребро, отже його розблоковувати не потрібно.
- Давайте запустимо DFS з кореня нового дерева, та випишемо вершини чорного кольору у порядку часу входження до них. Нехай це послідовність g довжини k. Введемо нову функцію $dist(x, y)$ — кількість ребер на шляху між парою вершин x та y у новому дереві. Звідси, отримуємо формулу для визначення відповіді на запит сьомого типу: $ans = (dist(g_1, g_2) + dist(g_2, g_3) + \ldots + dist(g_{k-1}, g_k) + dist(g_k, g_1))\ /\ 2$. Чому це вірно? Знову розглянемо будь-яке ребро з нового дерева. Якщо існує вершина чорного кольору у піддереві нижньої вершини цього ребра та вершина чорного кольору ззовні верхньої вершини цього ребра, то це ребро увійде до суми рівно два рази, бо воно увійде в це піддерево, та вийде з нього. Інакше, воно жодного разу не увійде до цієї суми.
- отже, для вирішення сьомого типу запитів треба вміти підтримувати цю суму. Оскільки відсутні запити першого типу, склад нового дерева змінюватись не буде, але вершини можуть змінювати колір. Давайте зберігати таку суму: $query_7 = dist(g_1, g_2) + dist(g_2, g_3) + \ldots + dist(g_{k-1}, g_k)$ Тоді нас цікавлять лише зміни що відбуваються у послідовності g. У нас може або додаватись новий елемент до послідовності, можливо десь у середину, або видалятись.
- випишемо в окремий масив усі вершини нового дерева у порядку часу входження до них при виклику DFS з кореня. Тобто, коли DFS входить у нову вершину, поточний час збільшується на 1. Звідси, $tin(v)$ — час входження до вершини v, а $tout(v)$ — час виходу з вершини v. Особливість цього порядку в тому, що усі вершини з піддерева вершини v зустрічаються на відрізку $tin(v) \ldots tout(v)$ у цій послідовності. Для кожної вершини запам'ятаємо позицію на якій вона зустрічається у цьому масиві. Якщо вершина має чорний колір то в додатковому масиві на її позицію поставимо одиничку, інакше нуль. Як треба оброблювати зміни у новому дереві?
  - зміна кольору вершини з чорного на білий. Припустимо, що ця вершина зустрічалась на позиції u в послідовності g. Тоді у формулу для знаходження значення $query_7$ вона могла увійти максимум у двох доданках.
    1. $u > 1$: $dist(g_{u-1}, g_u)$
    2. $u < k$: $dist(g_u, g_{u+1})$

    Тоді, нам треба перевірити можливі випадки та вміти знаходити відстань між двома вершинами, а потім віднімати їх від суми. Для цього можна скористатись додатковими структурами даних.
  - зміна кольору вершини з білого на чорний аналогічна, але тепер треба додавати числа до нашої суми.
- щоб дізнатись відповідь на запит сьомого типу, до нашої суми треба додати значення $dist(g_1, g_k)$, а потім поділити суму на два. Щоб знайти першу та останню чорну вершинки, звернемось до нашого масиву із нуликів та одиничок. Можемо на цьому масиві побудувати дерево відрізків та швидко знаходити наші вершини(можна зробити set, але дерево відрізків нам знадобиться для вирішення наступних блоків).

Блок 9:

- давайте випишемо усі вершини нашого дерева у порядку часу входження в них у DFS, викликаного із кореня дерева. Тепер, для кожної вершини у додатковому масиві, на позиціях де вони зустрічаються у масиві часу входження, будемо зберігати кількість заблокованих ребер на шляху від кореня до цієї вершини. Тоді, усі вершини з піддерева вершини v, що ще й знаходяться у компоненті вершини v, мають те ж саме значення у додатковому масиві що й значення вершини v. Згадаємо, що вершини піддерева вершини v знаходяться на відрізку $tin(v) \ldots tout(v)$ у масиві обходу DFS. Важливий факт, який треба помітити — значення вершини v на цьому відрізку у додатковому масиві мінімальне. А отже, усі вершини що мають мінімальне значення на цьому відрізку, знаходяться у компоненті вершини v. Тобто, якщо визначити у кожній компоненті її корінь(найближчу вершинку до кореня всього дерева), тоді ця компонента може бути визначена як множина вершинок, що лежать на відрізку $tin(v) \ldots tout(v)$, та мають на ньому мінімальне значення.
- побудуємо на додатковому масиві дерево відрізків. Кожен лист цього дерева буде відповідати за певну вершинку початкового дерева, визначену по номеру вершини у масиві обходу DFS. У кожному листі дерева відрізків будемо зберігати пару параметрів:
  1. значення вершини
  2. значення вершини у додатковому масиві(кількість заблокованих ребер на шляху від кореня дерева до цієї вершини)

  А для інших вершинок дерева відрізків, що відповідають за певні проміжки, будемо зберігати наступні параметри:
  1. сума значень усіх вершин з цього проміжку
  2. мінімальне значення у додатковому масиві усіх вершин з цього проміжку
  3. кількість вершин із мінімальним значенням на цьому проміжку

4. додаткові параметри, які умовно будемо називати обіцянками, щоб оброблювати певні операції на цьому дереві

Перейдемо до розв'язку запитів:

- запити типу 1: подивимось на піддерево нижньої вершини цього ребра. Оскільки це ребро заблоковане, то воно додає 1 до значення кожної вершини у додатковому масиві з цього проміжку. Тобто, у дереві відрізків треба відняти 1 від додаткового значення кожної вершини з цього проміжку, а це можна зробити використовуючи обіцянку $add1$ — число на яке треба змінити додаткове значення кожної вершини з цього проміжку.
- запити типу 2: спочатку, знайдемо корінь компоненти у якій знаходиться ця вершинка(як це зробити дивіться після розв'язку блоку 9). Нехай це вершина v. Тоді, до значення усіх вершин з компоненти вершини v треба додати число w. Оскільки ці вершини знаходяться на відрізку $tin(v) \dots tout(v)$, та мають мінімальне значення на ньому, то можна використати наше дерево відрізків, а саме ще один параметр для обіцянок $add2$ — число на яке треба змінити значення кожної вершини з цього проміжку.
- запити типу 3: викличемо запит типу 5 та 6. Запит типу 6 лише замінив значення кожної вершини нашої компоненти на нуль, але значення додаткового параметру не змінилось. Отже, ми можемо додати до значення нашої вершини p суму підраховану в запиті типу 5, використавши наше дерево відрізків для проміжку $tin(p) \dots tin(p)$.
- запити типу 4: використовуючи обіцянки, ми можемо дізнатись значення окремої вершинки, поступово спускаючись до неї по дереву відрізків та перекидаючи обіцянки до синів поточної вершини.
- запити типу 5: щоб дізнатись відповідь на цей запит, треба взяти суму значень усіх мінімумів на відрізку $tin(v) \dots tout(v)$, де v — корінь компоненти вершини p. Наше дерево відрізків вже зберігає цю суму для часткових проміжків, тому треба обрати головні з них та комбінувати.
- запити типу 6: будемо використовувати ще один параметр обіцянки в дереві відрізків $clear$ — чи було очищення кожної вершинки із мінімальним значенням на проміжку цієї вершинки дерева відрізків.
- слід зауважити, що на відміну від перекидування обіцянок у звичайному дереві відрізків(наприклад звичайне додавання на відрізку), в цьому дереві відрізків ми будемо передавати обіцянку лише до синів із мінімальним значенням. Це стосується усіх обіцянок окрім $add1$, яку слід передавати так само як у звичайному дереві відрізків.
- отже, асимптотика такого рішення буде $O(n\ log\ n)$.

Як ефективно знаходити корінь кожної компоненти(із можливістю застосовувати запити першого типу):

- давайте знову випишемо усі вершинки у порядку обходу DFS, а також повернемось до параметрів $tin(v)$ та $tout(v)$. Заблокованими будемо називати ті вершинки, ребро із яких до предка у дереві заблоковане. Отже, давайте зробимо ще один додатковий масив. Якщо вершина заблокована, або це вершинка 1(корінь дерева), то на її позиції будемо зберігати значення $tout(v)$, а інакше будемо на її позиції зберігати нуль.
- як знаходити корінь компоненти: по-перше, усі предки нашої вершини будуть знаходитись на позиціях із номерами менше ніж $tin(p)$, тобто на відрізку $1 \dots tin(p)$. А також, за властивістю tin та $tout$, значення $tout(v)$ повинно бути більше або рівним за $tout(p)$, якщо v предок p. Якщо існує дві вершини, що є кандидатами на корінь нашої компоненти, то серед них треба обрати останню. Тобто, лідер компоненти вершини p це остання вершинка на відрізку $1 \dots tin(p)$, на позиції якої стоїть значення що не менше за $tout(p)$.

Блок 11:

- єдине що ми ще не розглянули, це як одночасно підтримувати запити першого та сьомого типу. Тепер у нас може змінюватись структура нашого нового дерева, що було побудоване на стиснутих компонентах, тому будувати нове дерево, може виявитись занадто незручно. Давайте зробимо копію нашого початкового дерева. І тепер чорними будуть вершини, що є коренями компонент, що містять хоча б одну вершину із ненульовим значенням. Тоді змінилась функція $dist(x, y)$ — кількість заблокованих ребер на шляху між вершинами x та y. Проте, ми можемо дізнатись її значення використовуючи значення наших додаткових параметрів у дереві відрізків, що ми застосовували для вирішення перших 6 типів запитів.
- тепер, при виклику запиту типу 1, треба вміти поєднувати дві компоненти, та ділити одну компоненту на дві. Це ми можемо зробити певною послідовністю фарбувань вершин з білого кольору на чорний, та навпаки, залежно від складу компонент.
- проте, нам треба зауважити, що це ребро могло входити до значення $query_7$. Давайте спочатку змінимо його стан, перерахувавши значення $query_7$, а потім будемо поєднувати або ділити компоненти.

  1. зміна стану ребра з розблокованого до заблокованого: треба дізнатись скільки разів це ребро буде входити до суми. Згадаємо про масив де ми зберігаємо 1 на позиціях чорних вершин, та 0 на позиціях білих. Тоді, якщо на відрізку $tin(p) \dots tout(p)$ зустрічається хоча б одна одиничка, та на відрізку $1 \dots tin(p) - 1$ також зустрічається хоча б одна одиничка, тоді існує доданок $dist(q, p)$, що має входити до значення $query_7$. Тобто до цього значення слід додати 1. Так само, якщо на відрізку $tin(p) \dots tout(p)$ зустрічається хоча б одна одиничка, та на відрізку $tout(p) + 1 \dots n$ також зустрічається хоча б одна одиничка, то існує доданок $dist(p, q)$ що має входити до значенняя $query_7$, тобто це значеня треба знову збільшити на 1.
  2. зміна стану ребра з заблокованого на розблоковане: аналогічно до першого випадку, проте треба віднімати 1 від значення $query_7$.

Асимптотика рішення $O(n\ log\ n)$.