АНАЛИЗ НА РЕШЕНИЕТО НА ЗАДАЧА КАРНАВАЛ

- ✓ **Тагове**: структури от данни, геометрия, заявки, аритметична прогресия, коренова декомпозиция, convex hull trick, сегментни дървета, lazy propagation

Подзадача 1

Решението на първата подзадача за 9 точки е тривиално. В един масив ще поддържаме стойностите на щуростта за всеки зрител. Когато получим заявка за намиране на максимум в интервал, ще обходим въпросните стойности и ще вземем най-голямата. Когато получим заявка за промяна на стойностите, ще променим всяка една поотделно съгласно формулата за аритметична прогресия. Сложността на решението е $O(N \times Q)$.

Подзадача 2

Тъй като броят на заявките за промяна е само 200, нека помислим как да отговаряме по-бързо на заявките от първи тип. Това, разбира се, е класическата RMQ задача, начините за решаване на която би трябвало да са добре отработени от състезателите в група В. Ако се използва сегментно дърво сложността ще бъде $O(Q_1 \times \log_2 N + Q_2 \times N)$, ако се използва sparse таблица сложността ще бъде $O(Q_1 + Q_2 \times N \times \log_2 N)$, а ако се използва коренова декомпозиция – $O(Q_1\sqrt{N} + Q_2 \times N)$. При постъпване на заявка за промяна, съответната структура от данни се построява наново.

Подзадача 3

Тъй като броят на заявките за намиране на максимум е само 200, нека помислим как да обработваме по-бързо заявките от втори тип. Един от възможните начини е да разбием интервала на всяка заявка на две части – начало (индекс L_j) и край (индекс $R_j + 1$). Така в началото на интервала на промяната може да отбележим, че числата оттук нататък нарастват еднократно с A_j и освен това на всяка следваща позиция с D_j. След края на интервала добавяме еднократно нарастване с $(L_j - R_j) \times D_j - A_j$ и на всяка позиция с $-D_j$, което елиминира въздействието на заявката след позиция R_j. По този начин, прилагайки подход подобен на метода на замитащата права, ще можем да отговаряме на заявките от втори тип с линейна сложност. Общата сложност на това решение е $O(Q_1 \times N + Q_2)$.

Подзадача 4

В тази подзадача $D_j = 0$, което ни позволява да я решим чрез стандартен lazy propagation в сегментно дърво. Сложността е $O(Q \times \log_2 N)$.

Подзадача 5

Ще разширим идеята за lazy propagation, за да можем да добавяме аритметична прогресия в интервал и да намираме стойността на определен индекс от редицата. За целта във всеки връх ще пазим не една, а две lazy-стойности (сбора на първите членове и сбора на разликите на всички аритметични прогресии, които са приложени върху интервала на върха). Както знаем, при ъпдейт в сегментното дърво интервалът на заявката се разбива на $O(\log_2 N)$ интервала с дължина степен на двойката, за всеки от които отговаря конкретен връх в дървото. Нека сега сме в един такъв връх v, отговарящ за интервала от l до r в редицата. Тогава промяната в този интервал ще е еквивалентна на добавяне на

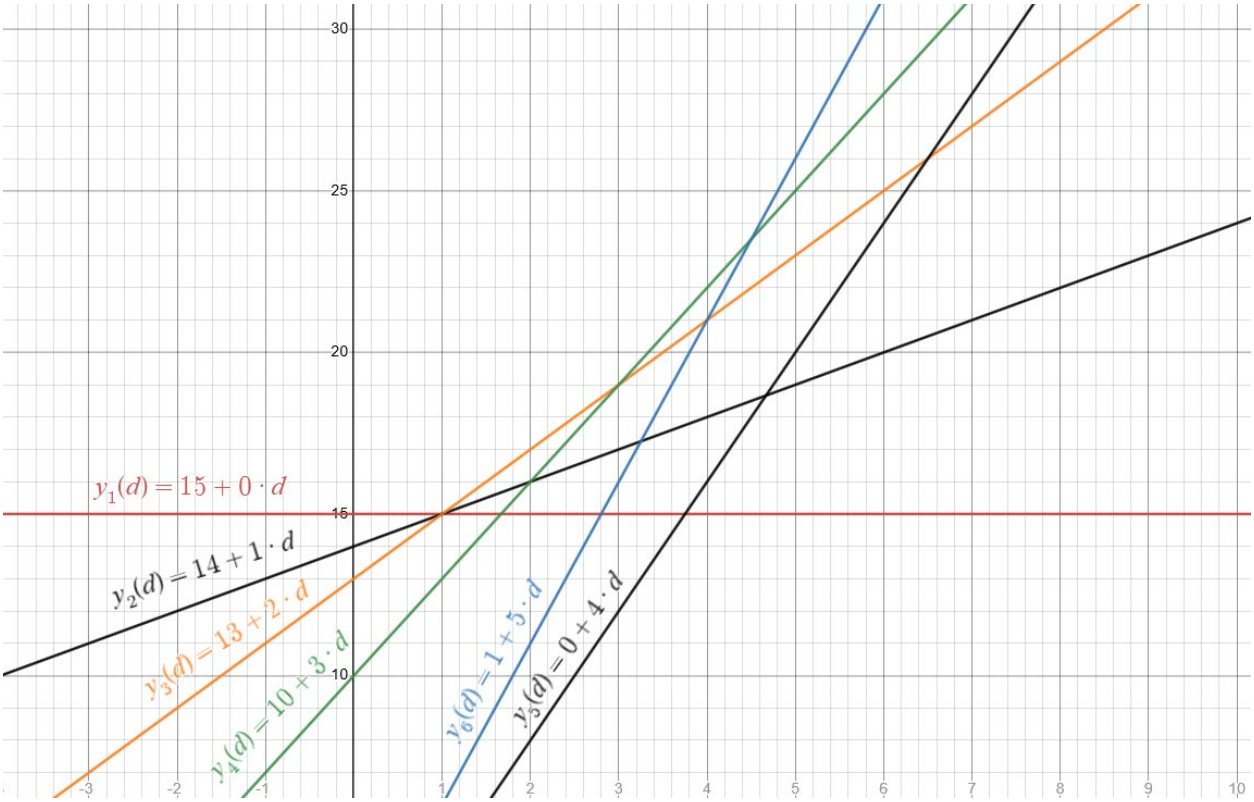
аритметична прогресия с първи член $A_j + (l - L_j) \times D_j$ и разлика D_j. Поддържайки кумулативно тези две стойности във всеки връх, лесно може да разберем каква е добавената стойност за всяка от стойностите в интервала. За да намерим стойността на даден индекс i от редицата, е необходимо да съберем началната ѝ стойност x_i със добавените стойности от прогресиите, съхранени във всеки връх, в чийто интервал попада индекса i. Сложността отново е $O(Q \times \log_2 N)$.

Подзадача 6

Решението на тази подзадача до голяма степен отваря вратите към решението на цялата задача. Тъй като заявките от втори тип се отнасят за целия интервал, то е ясно, че отговорите на заявките от първи тип ще се получават от индекс в редицата, който с течение на времето ще нараства (или поне е сигурно, че няма да намалява).

Да разгледаме следната редица: $15, 14, 13, 10, 0, 1$. В началото максимумът се получава от елемента x_1. При последователното извършване на ъпдейти това може да се промени. Например, след ъпдейт с разлика на прогресията 2, редицата ще бъде $15, 16, 17, 16, 8, 11$ и максимумът ще се получи от x_3. Забележете, че тук няма да разглеждаме стойността на първия член на аритметичната прогресия, която се добавя, а ще я приемем за 0, понеже тя променя всеки елемент от редицата с една и съща стойност и като цяло не влияе на отношенията между числата.

За да анализираме по-подробно тези промени, нека разгледаме функциите от вида $y_i(d) = x_i + (i - 1) \times d$. Тези функции показват каква ще е новата стойност y_i на елемента от редицата x_i след извършване на заявки за ъпдейт със сумарна разлика d.



АНАЛИЗ НА РЕШЕНИЕТО НА ЗАДАЧА КАРНАВАЛ

Разглеждайки графиките на тези функции, можем да установим, че няма две с еднакъв наклон, т.е. две функции се пресичат в точно една точка (не задължително с целочислена d-координата). За да отговаряме, на заявките за намиране на максимум, ни трябва най-голямата стойност на функция в това d, което е равно на сумата от разликите на приложените досега прогресии. Разбира се, не е удачно всеки път да преминаваме през всяка от функциите и да изчисляваме стойността им за текущото d. Затова трябва да забележим, че стойностите на d, в които максимумът се получава от дадена функция i, образуват непрекъснат (потенциално празен) интервал. Ако предварително намерим тези интервали, можем да поддържаме текущия максимум и след получаване на заявка за ъпдейт да проверяваме дали нямаме нов максимум и ако да – да го намерим. Така амортизирано имаме $O(N)$ операции по намиране на новия максимум за всички заявки.

Сега да уточним как точно намираме кои са функциите, които в определени непразни интервали дават максимална стойност. За примера това са интервалите: $[0;\ 1)$ за y_1, $[1;\ 3)$ за y_3, $[3; 4{,}5)$ за y_4, $[4{,}5;\ +\infty)$ за y_6. В сила е твърдението, че за $i < j$ интервалът на максималност на $y_i(d)$ е преди интервала на $y_j(d)$, в случай че и двата са непразни. Така че разглеждаме функциите y_i в ред на нарастване на i и поддържаме тези, които до момента имат непразен интервал на максималност. При добавяне на нова функция, тя ще бъде с най-голям коефициент пред d, така че със сигурност ще е максимална от някаква точка до безкрайността. Въпросът е кои от запазените до момента функции ще отпаднат и отговорът ни се дава от изложеното по-горе твърдение – нито една или някакъв суфикс от списъка (приемаме, че са подредени по нарастване на i). Условието за премахване на последната функция от списъка е: ако d-координатата, от която последната функция започва да доминира над предпоследната в списъка е по-голяма или равна на d-координатата, от която новодобавената функция започва да доминира над предпоследната. Тази техника много напомня на алгоритъма на Graham за намиране на изпъкнала обвивка на множество от точки и затова се нарича Convex Hull Trick.

Подзадача 7

Тази подзадача е предназначена за бавни имплементации на авторовата идея.

Подзадача 8

Както вече разбрахме, можем да решим задачата, когато заявките обхващат цялата редица. Това веднага би трябвало да наведе състезателите на мисълта да търсят някаква структура от данни, с която да могат да намалят сложността за изпълнение на заявките, използвайки този факт. Особено подходяща се оказва коренова декомпозиция.

Накратко идеята е да разделим редицата на $\sqrt{N}$ блока с големина $\sqrt{N}$ елемента. Когато получим заявка за промяна, просто отбелязваме промяната (подобно на lazy в сегментно дърво) към изцяло засегнатите блокове и построяваме отново тези, които са засегнати само частично. Когато получим заявка за намиране на максимум, обхождаме елемент по елемент частично обхванатите блокове, а за останалите просто проверяваме за наличие на нов максимум. Сложността е $O(N + Q\sqrt{N})$, като отново обработването на заявките за намиране на максимум има амортизирана сложност (т.е. някои от заявките може да са значително по-бавни от други, но сумарно броят на операциите е ограничен).

АНАЛИЗ НА РЕШЕНИЕТО НА ЗАДАЧА КАРНАВАЛ

Забележка: Асимптотично изборът големината на блоковете да бъде $\sqrt{N}$ е оптимален, но на практика това може да варира с разлика до константен фактор в зависимост от реализацията.

Едно решение за 0 точки

Описаното тук решение няма никаква практическа стойност в конкретната задача, понеже, както ще се убедите сами по-късно, е много бавно и също така използва голямо количество памет. Нещото, което го прави ценно, е, че е със сложност $O(Q \times \log_2^3 N)$, което би било по-добре от $O(N \times \sqrt{N})$ за много големи стойности на N.

От подзадача 6 става ясно, че е доста удобно да се update-ва цял интервал и да се правят заявки за него. Тоест проблемното нещо е това да update-ваме само част от интервал. По-конкретно, тъй като в подзадача 6 се update-ва и query-ва цялата редица, то можем да пазим един Convex Hull Trick за нея и да „отместваме“ правите при update (реално прилагаме транслация на „геометричната обстановка на Convex Hull Trick-а“). Query-то пък в тази ситуация е доста просто. В решението за 100 точки се разширява функционалността на задачата, като се разделя редицата на блокове и всеки блок се update-ва, когато е изцяло вътре в заявката за update и се построява наново, когато само частично е вътре.

Доста естествено е след като имаме решение с коренова декомпозиция да помислим и за такова със сегментно дърво. Тоест да разделим редицата на интервали в стил на сегментно дърво и да пазим някаква информация, която ни интересува, във всеки връх на дървото. За да поддържаме update-и в тази ситуация, ни трябва поне едно от двете – да можем бързо да променяме информацията във връх, когато той е частично засегнат или да можем бързо да „сливаме“ информацията от две деца на някой връх. Първото нещо изглежда доста отчайващо, пък и също така, ако го имахме, едва ли щяхме да се занимаваме изобщо да правим дърво, понеже можеше директно да си пазим информацията за цялата редица, затова ще помислим върху втория ни вариант.

Бързо сливане на два Convex Hull Trick-а изглежда доста безнадеждно само по себе си. Добрата новина е, че тук не ни трябва да сливаме изцяло произволни такива. Тъй като ние винаги сливаме две деца, то е вярно, че наклоните на правите от „първия“ CHT са строго по-малки от наклоните на всички прави от „втория“. Оказва се, че е вярно, че резултатът е всъщност конкатенация на някакъв префикс от „първия“ CHT и суфикс от „втория“, ако мислим за CHT като списък от прави. Тоест имаме шанс лесно да видим кои части биха влезли в резултатния CHT с двоично търсене. Сега само трябва да можем да проверяваме дали една права е доминирана от някоя друга или не. В момента ще разсъждаваме само за „първия“ CHT, понеже за „втория“ е аналогично. Проверката дали да „попнем“ една права при строенето на CHT е следната – ако „новодошлата“ права се пресича с предишната на последната с по малък x, отколкото последната се пресича с предишната ѝ, то тя е изцяло доминирана от новодошлата. Тъй като не можем да симулираме този процес имаме следния вариант – можем да проверяваме дали от “втория” CHT има права, която в x, в който конкретната права се пресича с предишната, има по-голяма стойност. Повече подробности може да се види в имплементацията на

АНАЛИЗ НА РЕШЕНИЕТО НА ЗАДАЧА КАРНАВАЛ

ConvexHullTrick operator +(const ConvexHullTrick &lhs, const ConvexHullTrick& rhs) от файла **nlog3_prototype_stoyan.cpp.** Може да се докаже, че това условие е необходимо и достатъчно. Тази идея без допълнителни технически усложнения е написана във файла **nlog3_prototype_stoyan.cpp**. Там СНТ е написан с std::vector и на практика решението е $O(N \times Q \times \log_2 N)$, но идеята се вижда много по-ясно.

Остава да се свърши по-досадната и техническа част от задачата. Трябва да намерим структура от данни, която лесно се „дели“ и „слива“. Също така поддържа някаква форма на update-ване с lazy propagation и освен това може да се напише персистентно. Във файла **nlog3_stoyan.cpp** е използвано компресирано сегментно дърво (обектите от тип **LineSet**), но може би по-удачно би било да се използва **Treap**. Така сложността за едно сливане става $O(\log_2^2 N)$, понеже ни трябват $O(\log_2 N)$ операции, за да „слезем по дървото“, тоест да направим двоично търсене (функциите **int LineSet::getFirstOut(const shared_ptr<LineSet> &rhs)** и **int LineSet::getFirstIn(const shared_ptr<LineSet> &lhs)**) и още $O(\log_2 N)$ операции, за да проверим колко е оптималната стойност за някой си x (функцията **double LineSet::get(int x)**).

Така окончателно един update в сегментното дърво **SegmentTreeNode** ни коства $O(\log_2^3 N)$ операции, а всяко query $O(\log_2 N)$ операции. И така получаваме, че задачата става със сложност $O(Q \times \log_2^3 N)$ от гледна точка на време и $O(N \times \log_2 N)$ от гледна точка на памет, понеже във всеки момент не повече от около $2 \times N$ прави общо за всички върхове на сегментното дърво и всяка права от тях може да изисква $O(\log_2 N)$ допълнителни върхове заради факта, че **LineSet** е компресирано сегментно.

Автори: Добрин Башев, Стоян Малинин