Анализ на задача `freetime`

Тагове: сегментно дърво, lazy propagation, офлайн заявки, алгоритъм на Мо, sweep line, дърво на Фенуик, амортизирана сложност

Задачата беше замислена като по-лесна задача за темата при опит за по-нестандартни заявки, които да не могат директно да се поддържат със сегментно дърво. Крайният резултат се оказа задача по-близка до средна трудност, макар и да се ползват сравнително стандартни техники и номера.

В анализа ще наричаме милисекундите точки, а броят работни милисекунди – големина на покритието от интервали. Възможно е да компресираме точките и така винаги максималния десен край да е от порядъка на N, но това почти няма смисъл в задачата, защото в повечето подзадачи, така е по условие. Също вместо D ограничения по дни, ще говорим за Q заявки от подмасиви на началните интервали.

Решение на първа подзадача – 4 точки

Това беше подзадача, така че всеки да може да изкара много лесни точки по задачата. В един масив за всеки интервал отбелязваме маркираните точки и накрая ги преброяваме.

Сложност: $O(QN^2)$.

Решение на втора подзадача – 8 точки

Има различни начини да смятаме покритието на всички интервали. Един начин, полезен за следващите решения, е чрез сегментно дърво върху точките. Нека да използваме статично *lazy propagation* – единствено ще маркираме за всеки връх в сегментното дърво дали целият сегмент, за който отговаря върхът, е покрит. Авторовата реализация пази за всеки връх в дървото броят покрити точки. Така за забързване и улесняване на имплементацията, може да прекъсваме рекурсивното обхождане при добавяне на нов интервал, ако стигнем връх в сегментното дърво, за който всички точки са покрити.

Сложност: $O(QN \log N)$.

Решение на трета подзадача – 8 точки (=4+0+4)

В тази подзадача е достатъчно да пресметнем предварително отговорите на всички заявки. Така фиксираме левия край на подмасивите и малко по-малко добавяме всички останали интервали. Алтернативен начин на алгоритъма със сегментно дърво (което трябва да чистим всеки път при смяна на левия край) е да поддържаме явно покритието от несвързани интервали със *set* (дори може и да изискваме да има поне една точка между интервалите, защото ако са $[1, 3]$ и $[4, 7]$, то за целите на нашата задача това е еквивалентно на $[1, 7]$). Този подход също ще ни е полезен за накрая.

Достатъчно е като се добави нов интервал $[l, r]$, да обходим в *set*-а всички интервали, които се пресичат с него, при което може да се наложи да изтрием интервалите, съдържащи се в новия, както и да променим някой интервал, ако например само частично се припокрива с нашия отляво или отдясно. Съответно при всички тези промени, трябва да поддържаме броя точки на интервалите в *set*-а. Може да видим, че амортизирано ще гледаме само константен брой интервали всеки път, защото всеки интервал в *set*-а се появява веднъж и най-много да се премахне веднъж. А на всяка стъпка, макар да е възможно много интервали да се премахнат, то се добавя най-много един интервал (в авторовата имплементация винаги е един, за удобство).

Сложност: $O(N^2 \log N + Q)$.

Решение на четвърта подзадача – 28 точки (=4+4+0+20)

Това е една от страничните подзадачи, които ни дават повече точки. Предвиденото решение е с класическата офлайн техника за такава задача – коренова декомпозиция на заявки или алгоритъм на Мо. За да може да го приложим е важно да използваме подход, който има гарантирано добра сложност при добавяне на интервал (описаният амортизиран не е такъв). Затова ще използваме подхода със сегментното дърво. Един проблем при него е, че той е добър за добавяне на интервали, но не може да се справи с премахване на интервали. Ще приложим модифицирана версия на алгоритъма на Мо, която се използва за структури от данни, които поддържат добавяне и *undo/revert*.

Идеята накратко е да подредим заявките класически по първи приоритет – нарастващ бъкет на левия край, а втория ни приоритет да е нарастващ десен край. Така, ако бъкетите ни са с размери K и в момента разглеждаме бъкет с леви краища в $[tK, (t+1)K-1]$, то десните краища искаме да са поне $(t+1)K$ и да нарастват. За да гарантираме това, преди да почнем заявките, намираме отговора на всички заявки, чиито краища са в един бъкет, или на които дължината е до K. След това за всяка заявка използваме натрупаната информация от $(t+1)K$ до стария десен край, добавяме интервалите до новия десен край и накрая добавяме интервалите с индекси от $(t+1)K-1$ до новия ляв край. За да може този процес да работи, след като сме намерили отговора правим *revert* на промените на сегментното дърво при добавянето на интервалите за левия край. Това най-лесно се прави, като пазим в списък всички промени, които стават в сегментното дърво при добавяне на интервалите за левия край, и след това само връщаме старото състояние. Допълнително, когато се сменя бъкетът на левите краища, вместо да нулираме дървото, можем просто да поддържаме време на всеки връх в сегментното дърво, което да съответства на бъкета. Така ако видим, че времето на връх в сегментното дърво е старо, приемаме, че този връх е празен.

За жалост, при изготвяне и тестване на задачата, не беше разгледано алтернативно решение с алгоритъма на Мо, което леко изненадващо се оказа значително по-бавно от описаното. Възможно е да използваме класическия алгоритъм на Мо. За тази цел ще направим сегментното дърво да може да поддържа и премахване на интервали. Променяме малко информацията в сегментното дърво и вече вместо да маркираме покритите точки, ще отбелязваме от колко интервала са покрити, т.е. добавяне на нов интервал е като добавяне на $+1$ в съответния сегмент на сегментното дърво. Съответно премахването на интервал е добавяне на -1. За да знаем колко точки са покрити, сегментното дърво може да пази във всеки връх просто колко нули има в съответстващия сегмент, което най-лесно можем да интерпретираме като колко пъти се среща минималната стойност в сегмента, ако тя е 0 (все пак броя покриващи интервали за дадена точка няма как да е под 0). Възможно е да се напише по-оптимизирано този подход, да се избере подходящ размер на бъкетите, но пак това решение е съвсем леко по-бавно и не може да хване подзадачата. Оказва се, че модифицираният алгоритъм на Мо е просто е по-бърз. Една оптимизация, с която може да хванем подзадачата, е компресиране на точките и така се намаля малко логаритъмът.

Сложност: $O((N+Q)\sqrt{N}\log N)$.

Решение на пета подзадача – 7 точки (=0+0+0+0+7)

Може да забележим, че допълнителното ограничение за интервалите прави решение със сегментно дърво, в което поддържаме покритието, валидно. Всъщност ако обединим интервалите на последователни позиции, то винаги получаваме един интервал, като левият му край е равен на минималният ляв край на интервалите, а десният му край е равен на максималния десен край на интервалите. Затова решението е просто да поддържаме тази информация - за минимален ляв край и максимален десен край, което описва интервала на покритието, в сегментно дърво и директно да отговаряме на заявките.

Сложност: $O(N + Q \log N)$.

Решение на шеста подзадача – 30 точки (=0+0+0+0+0+30)

Още една странична подзадача, която макар и да носи много точки, не е насочваща за крайното решение. В случая многото точки са, защото решението само за тази задача е сравнително сложно (дори за писане е по-сложно от пълното). Възползвайки се от допълнителното ограничение може да си мислим за следното. Даден интервал е активен, т.е. допринася за покритието, само за заявки, които не включват интервали, които го съдържат. Затова нека за всеки интервал намерим първия вляво и първия вдясно от него (спрямо началната подредба), които го съдържат.

Нека разгледаме общия случай, в който за даден интервал i, $prev$ е индексът на предния съдържащ, а $next$ на следващия съдържащ. Тогава ако си представим, че отговаряме офлайн на заявките чрез *sweep line* като фиксираме последователно възможните леви краища и поддържаме активните интервали, трябва да активираме интервала i, след като приключим с ляв край $prev$ и също така да деактивираме този интервал в $next$ позиция. Може да си мислим, че това става като просто добавяме броя точки, които покрива интервал i, на позиция i и изваждаме този брой на позиция $next$. Така просто една сума на тези стойности в интервала на заявката, ще ни даде точно броя покрити точки (без да се броят по няколко пъти). Можем да поддържаме тази информация с дърво на Фенуик.

Последната част е как намираме съответните $prev$ и $next$. Има различни варианти. Може би най-просто е да направим *sweep line*, като сортираме краищата на интервалите и в един *set* поддържаме индексите на все още отворените интервали. Така като сме на някой интервал трябва да намерим в *set*-а най-близкия по-малък и по-голям индекс (ако съществуват). Ще обърнем внимание, че трябва да обхождаме интервалите с един и същ ляв край от дълъг към къс, т.е. в намаляващ ред на десните краища. Също тук се възползваме от това, че няма съвпадащи интервали, защото иначе трябваше още малко да добавим към логиката. Идеята на това ограничение с липсата на съвпадащи интервали е точно, за да се премахнат досадни имплементационни детайли в повечето реализации.

Сложност: $O((N + Q) \log N)$.

Решение на седма подзадача – 16 точки (=4+4+4+0+0+0+4)

Решението на подзадачата е отново офлайн. Фиксираме левият край на заявките и сортираме заявките по десен край. След това с някои от описаните подходи добавяме един по един интервалите, докато включим всички от дадена заявка и намираме отговора.

Сложност: $O(Q \log Q + XN \log N)$, където X е максималното ограничение за левия край на заявките.

Пълно решение 1 – 100 точки

Идеята на предната подзадача макар и лесна, беше да подсказва в каква посока е пълното решение, затова е и сложена като предпоследна. Може да се забележи, че е възможно да поддържаме допълнителна информация за покритието, така че да отговаряме на заявки с фиксиран десен край и променящ се ляв край. Проблемът по принцип е, че когато е фиксиран десният край и левият край е променлив, то трябва да можем да разберем какво е покритието, ако се включат само част от интервалите. За тази цел нека да пазим за всяка точка в покритието, индексът на интервала, който я е покрил последно. Ако гледаме по този начин, то за заявка $[l, r]$ броят точки, които се покриват, са всъщност броят точки, чийто индекс на последно покриване е поне l. Разбира се, можем да не пазим за всяка точка отделно индекса на последно покриване, а да групираме покритите точки с равни индекси, т.е. отново да поддържаме непресичащи се интервали в *set* както в трета подзадача, но с допълнителна информация за индекса. Авторовата имплементация използва *map* за интервалите по тази причина.

Аргументът защо амортизирано това е бързо е подобен като в трета подзадача. Отново като добавяме нов интервал към покритието трябва да обходим всички, които се пресичат с него. Единствено сега имаме малко повече случаи. Един такъв е, ако има интервал, който изцяло съдържа нашия и тогава големият интервал се разпада на три части (най-лявата част е със стария индекс на покриването, най-дясната също, и само текущият интервал има новия индекс на покриване). Аналогични са случаите, ако има интервали, които частично се припокриват отляво или отдясно с текущия. Така на една стъпка може да ни се наложи да добавим до три интервала, но отново всеки интервал се премахва най-много веднъж от покритието. Затова амортизирано имаме константен брой интервали на всяка стъпка.

Последният детайл е, че за да отговаряме бързо на заявки $[l, r]$, след като сме добавили всички интервали до r-тия, трябва да можем да кажем броя точки с индекс поне l. Затова в дърво на Фенуик можем да пазим префиксния масив на бройките точки с даден индекс и да го поддържаме актуален при всяка промяна на покритието. Интересно е, че е възможно и първата част да направим само с дърво на Фенуик, като използваме трика в дърво на Фенуки за бързо намиране на първото попълнено число в масив. Но в случая този подход не е особено по-бърз от описания.

Сложност: $O((N + Q) \log N)$.

Пълно решение 2 – 100 точки

Алтернативното решение прилича на предното, но всъщност обобщава подходите със сегментно дърво. Отново искаме да поддържаме информацията за последния индекс на покриване на точките. За тази цел в едно сегментно дърво освен да маркираме покритите точки, ще записваме и индекса на последно покриване. Най-лесно е да имаме и две специални състояния за върховете – съответният сегмент няма маркирани точки, съответният сегмент не е последно покрит от един и същ интервал. Така всяко добавяне на интервал към сегментното ще става с *lazy propagation* за текущия индекс спрямо върховете и съответстващите им сегменти, които попадат в текущия интервал. Интересната част е поддържане на актуалната информация за това колко точки са последно покрити с даден индекс. За тази цел достатъчно е преди да сложим *lazy update* на даден връх, да направим още едно допълнително спускане в сегментното дърво надолу и всъщност ръчно да намерим интервалите на покритите точки с еднакъв индекс и да актуализираме тази информация. Поддържането на броя точки, покрити с даден индекс, може да става отново с дърво на Фенуик.

Това може да изглежда бавно, но всъщност отново амортизирано е бързо. Един начин да го разгледаме е, че подобно на предното решение, поддържаме непресичащи се интервали, но със сегментно дърво. Друг начин е да си представим как при извършване на нормалните операции с *lazy propagation* и предаване надолу на промените, във всеки такъв връх поставяме по един предмет. Допълнителното спускане, която правим, всъщност минава през вече обходени върхове от предни операции и премахва предметите в тях. Може да забележим, че то се спира точно в най-долните върхове, в които сме сложили предмети. Затова понеже поставяме общо $O(N \log N)$ предмети при всички нормални операции, то общо допълнителните спускания няма как да ни отнемат повече време от това.

Сложност: $O((N + Q) \log N)$, но с по-голяма константа.

Интересното на последното решение е, че то е сравнително по-общо. Ако имаме сегментно дърво, в което поддържаме по-сложна информация и *lazy propagation*, то можем всеки път да видим старото състояние преди новата промяна, без да пазим огромна информация във всеки връх, което не забавя особено значително алгоритъма.

Подзадачите на тази задача са добър пример за честно срещан подход и при съставяне на някои задачи на международните състезания. Макар да няма много междинни стъпки за крайните решения, то има много възможности да се изкарат значителен брой точки, без да се реши нещо съществено по крайната задача (в случая 73 точки!). За тези, които не могат да решат задачата, това е добро упражнение за изкарване на точки с различни идеи.

Автор: Илиян Йорданов