# Анализ

Задачата Ruler беше малко по-нестандартна от повечето, давани по ученически състезания. Все пак, основната "идея" – да се преизчислят локално отговорите за всички възможни входове беше някак интуитивна и не вярвам да убегне на много състезатели. Time Limit-ът от 1 секунда беше даден за да може все пак участници с вярно, макар и бавно решение, които, обаче, не са се сетили да hardcode-нат в кода си отговорите, да хванат някакви точки (така би трябвало да се хванат тестовете до $N \leq 11$ или $N \leq 12$ (в зависимост колко оптимизирано е решението им).

Задачата всъщност е донякъде известен проблем: (https://en.wikipedia.org/wiki/Golomb_ruler). За момента няма измислено ефективно решение, което да намира оптималните дължини или разпределения (макар и да има апроксимационни алгоритми, които дават близки до оптималните стойности). Липсата на интернет и телефони на състезанието позволи все пак да я дадем, тъй като ако такъв има, отговорите могат да бъдат намерени *много* бързо.

Сега нека разгледаме какво е решението тук.

В даденото решение ще наричаме "маркерите" или "деленията" на линийката просто "числа" – интересува ни само къде са те, което са си цели, неотрицателни числа. Разстоянието между деленията е абсолютната стойност на разликата на тези числа.

**Динамично?**

На пръв поглед състезателите могат да се насочат към динамично оптимиране, в което стейтът е до кое число сме стигнали и кои разстояния са вече използвани. Това, обаче, се оказва, че не е приложимо, тъй като има значение кои *числа* сме използвали – добавяйки ново такова, добавяме и нови разстояния, които могат да са различни, в зависимост кои числа сме ползвали на предните стъпки. Възможно е да ползваме два различни сета числа и да получим един и същ сет от използвани разстояния! Така динамичното се "лъже", че е виждало даден стейт и дава грешен отговор.

След като сме длъжни да знаем кои числа сме ползвали на предишните стъпки, обаче, можем да заключим, че динамично оптимиране не може (или по-скоро няма логика) да бъде използвано тук, тъй като е еквивалентно на пълно изчерпване (чрез рекурсия/бектрек), само че харчи памет.

**Бектрек**

Първо трябва да тръгнем от там, че не знаем каква ще е дължината на линията. Очевидно тя трябва да е поне N. Реално, ако имаме N числа имаме $N * (N - 1) / 2$ двойки числа, чиито разстояния искаме да са уникални. Това е значително по-добра долна граница, която можем да използваме. Друга долна граница,

която можем да ползваме, е ans[N – 1] + 1 (отговорът за линийка с едно деление по-малко плюс едно). По-нататък ще видим, че можем лесно да го знаем (виж Наблюдение 2).

**Наблюдение 1**

Можем да изчислим всички отговори и да ги вкараме в кода си, като така времето за изчисление е лимитирано до под 5 часа, вместо дадената 1 секунда. Това вече споменахме в началото, но е доста важно, затова го повтаряме. Тъй има малък брой възможни входове (в тази задача – едва 10), и отговорът за всеки възможен вход е относително кратък, можем да ги преизчислим и hardcode-нем в кода ни, постигайки O(1) сложност при изпълнение на решението ни на системата.

**Наблюдение 2**

За изчислението на N можем да изчислим и ползваме всички отговори за N – 1, N – 2, … 1 без това да направи решението ни много по-бавно. Тъй като имаме експоненциално нарастваща сложност на изчисленията (идваща от пълното изчерпване), за да изчислим N = X ще ние нужно поне двойно време, отколкото за N = X – 1. Реално, за тази задача, времето нараства значително по-бързо от двойно, така че това изобщо не ни забавя, на практика. Ползвайки наблюдение 1 можем да ги ползваме директно без да харчим каквото и да било време за изчисляването им всеки път.

**Оптимизация 1: Използване на побитови операции**

Тъй като както ползваните числа, така и разстояния, са относително малки (както се оказва, до 127), можем да ползваме побитови операции (или пишейки си ги сами, или ползвайки STL-ския bitset) за да пазим и променяме стейта. Така операции, които иначе биха били линейни, стават константни.

Важно е да прилагаме побитовите операции не само за да пазим стейта (кои числа и разстояния са използвани), ами и да го ъпдейтваме. Трябва да измислим как, добавяйки ново число, можем константно да ъпдейтнем стейта.

Представете си, че вече сте сложили 10 числа (има логика да слагате числата в нарастващ ред), като най-голямото е X. Новото число ще пробвате на позиции X + 1, X + 2, … и т.н. Представете си, че изчислите разстоянията от всички досега сложени числа до новото, когато е X + 1. Пробвайки X + 2, всички тези изчислени разстояния до досега сложените числа ще се увеличат с 1. Така можем да ползваме побитовата операция shiftleft (<<=) за да ги ъпдейтнем за O(1). Нещо повече, знаейки тези разстояния, лесно можем да проверим дали числото, което гледаме в момента, може да бъде сложено! Ако го and-нем с вече ползваните разстояния и получим нещо различно от 1, то очевидно вече някое от разстоянията е ползвано, и съответно не можем. Тази операция също е O(1). Единственото линейно нещо е това, че трябва да пробваме X + 1, X + 2 и т.н.

**Оптимизация 2: Използване на предходни отговори**

Тъй като в началото фиксираме колко е дължината на линийката (L), то по всяко време знаем колко "разстояние" ни остава до края ѝ (нека бъде някакво число R), и колко още деления трябва да сложим (нека бъде някакво число M). M очевидно ще бъде по-малко от N (тъй като вече ще сме сложили деление в позиция 0), тоест ще имаме изчислен отговора ans[M]. Ако по някое време стигнем до позиция, в която ans[M] > R, то очевидно няма как да "довършим" линийката и можем да игнорираме този клон на търсенето.

**Оптимизация 3: Евристика за нарастващите разстояния**

Както казахме, за да сложим N числа ще ни трябват поне N * (N – 1) / 2 позиции. Така, ако сме в стейт, в който ни остава разстояние до края на линийката R и M числа, то ако R < M * (M – 1) / 2 можем директно да спрем.

**Оптимизация 4: Евристика за нарастващите *неизползвани* разстояния**

Нещо повече, можем да доразвием горната оптимизация, тъй като вече знаем, че част от разстоянията са "използвани" и не могат да се срещат отново. Представете си, че сме ползвали разстоянията 1, 3, 5, 6, 7, 10, 11, 13 (и по-големи, които не ни интересуват за момента), последното ползвано число е 42, и ни остават още 5 деления, които трябва да "сложим". Така най-най-минималната реалистична конфигурация за останалите 5 числа би била:

1. 44 (ползвайки разстояние 2 до 42)
2. 48 (ползвайки разстояние 4 до 44)
3. 56 (ползвайки разстояние 8 до 48)
4. 65 (ползвайки разстояние 9 до 56)
5. 77 (ползвайки разстояние 12 до 65)

Разбира се, тази конфигурация не е валидна (тъй като, например, 48 до 42 дава разстояние 6, което вече е ползвано), но все пак можем да ползваме като евристика за долна граница колко разстояние ще ни трябва още, за да довършим отговора. Така, вместо 5 * (5 – 1) / 2 = 10, можем да видим, че ще ни трябва *поне* 77 – 42 = 35.

Накратко, вместо да ползваме формулата M * (M – 1) / 2, ще ползваме сумата на първите M неизползвани разстояния.

**Оптимизация 5: По-добрата половина**

Ако ще разположим N числа в L позиции, то поне половината от числата ще са или в първата или във втората половина от позициите. Тъй като една линия е валидна както от ляво надясно, така и от дясно наляво (имаме точка в нулевата и в последната позиция, което ни позволява това), то можем да фиксираме поне половината от числата да са в първата половина (проверявайки на (L/2+1)-ва позиция дали вече сме сложили поне N/2 числа).

**Оптимизация 6: Посока на търсене**

Функцията е монотонно нарастваща, защото, ако оптималният отговор за N е X, то отговорът за N + 1 ще се нуждае от поне още една позиция за новото число. Ако не сме ограничени да слагаме деление в края на линията (тоест последното деление може да е, примерно, в L – 2 при линия с дължина L), то функцията, също така е и непрекъсната.

Така, вместо да пробваме ans[N – 1] + 1, после ans[N – 1] + 2, после ans[N – 1] + 3 и т.н., докато стигнем до отговора за N, можем да приложим двоично търсене. За горна граница можем да ползваме ans[N – 1] * 2 + 1 – тъй като сме сигурни, че в отговора за N – 1 не сме ползвали разстоянието ans[N – 1] + 1, то можем да разположим N-тото число на това разстояние от последното, като така гарантираме уникалността на всички новополучени разстояния.

Макар и двоичното търсене значително да забързва нещата, то все пак не е най-доброто, което можем да направим. "WAT!?", може би си мислите? Повечето подобни изчерпвания намират отговор относително бързо (особено ако имат достатъчно "излишно" разстояние), но в същото време правят пълно изчерпване ако няма отговор за фиксираното L. Така "негативните" отговори, както и потенциално първият/първите няколко позитивни са относително бавни, докато позитивните с L по-голямо от оптималното са относително бързи.

Вместо да правим двоично търсене (което, on average, ще мине през поне няколко отрицателни отговора), можем да почнем от горната граница и да слизаме линейно надолу, спирайки при първия негативен отговор. Така ще намираме много бързо отговорите за L-овете, които са по-големи от оптималното (търсенето се забавя чак като стигнем много близо до оптималното).

Допълнително, понякога с L много по-голямо от оптималното успяваме да намерим отговор не с последно число L, ами с L – X. Така, следващата стъпка не трябва да е L – 1, ами L – X – 1 (реално прескачаме няколко стойности, за които сме сигурни, че има отговор).

*Автор: Александър Георгиев*