**АНАЛИЗ НА РЕШЕНИЕТО НА ЗАДАЧА**

**ЧАЛГА**

Disclaimer: задачата Chalga до голяма степен e „открадната“ от Timus: 2107. Oppa Funcan Style. Единствените разлики са, че тук ограниченията бяха по-големи (там са 37, тук бяха 50), и тук беше гарантирано, че ще има решение (което облекчава задачата откъм досадни частни случаи).

**Малко за пермутации**

За решението на задачата първо трябваше да видим какво означава това танцьорките да са на началните си позиции след K секунди. Както беше казано в условието, трябва след прилагането на пермутацията K пъти върху редицата (1, 2, …, N) да получим отново (1, 2, …, N). Състезателите трябваше да знаят, че това става тогава и само тогава, когато всеки от циклите в пермутацията дели K.

Цикъл в пермутация е последователност от нейни елементи, в които всеки сочи към следващия, и последният сочи към първия. В пермутацията от решението на примера в задачата (7, 4, 9, 10, 6, 1, 3, 8, 11, 2, 5) имаме три цикъла:

1. (1 -> 7 -> 3 -> 9 -> 11 -> 5 -> 6 -> 1)
2. (2 -> 4 -> 10 -> 2)
3. (8 -> 8)

Дефиницията на пермутация е такава, че всяка пермутация може да бъде разбита на независими цикли. Намирайки най-малкото общо кратно на дължините на циклите ни дава след колко прилагания получаваме отново началната пермутация. Съответно, ако K е кратно на дължината на всеки от циклите, то имаме пермутация от желания вид.

**Решение чрез изчерпване**

Едно очевидно решение е да пробваме всички възможности как да попълним нулите. Ако имаме Z нули, то имаме Z! варианта, които можем да проверим със сложност O(N \* Z!). В много от тестовете макар и да имаме голямо N, броят нули не е толкова голям. При това, тъй като съществуват много решения, относително бързо стигаме до такова и спираме. Поради тези факти това решение базирано на изчерпване хваща учудващо много от тестовете (малко над половината). Групирането на тестовете, обаче, е така направено, че то да не хване повече от 10-15 точки.

**Решение чрез динамично оптимиране**

Решението на задачата беше … (drumroll) … динамично оптимиране. Което едва ли е учудило много от състезателите. Въпросът беше как да направим стейта достатъчно ефективен.

Вече създадените цикли във входната пермутация не ни интересуват – разбира се, те трябва да делят K, но тъй като е гарантирано, че оригиналната пермутация е изпълнявала това условие, то няма как това да не е така.

Сега оставаме с последователност от навързани числа (различни от нули), които не образуват цикъл – тоест завършват в нула. Тях ще наричаме „сегменти“. Дължина на сегмент ще наричаме броя индекси, които включва. Например, нека разгледаме входа от условието: (0, 0, 9, 10, 6, 1, 3, 0, 11, 2, 0). В него имаме следните сегменти:

1. (3 -> 9 -> 11 -> 0)
2. (6-> 1 -> 0)
3. (10 -> 2 -> 0)

Освен тях имаме и една нула, която не е край на никой сегмент (тази на позиция 8).

В дължина на сегментите ще включваме и „терминиращата“ нула, защото реално нейната позиция е фиксирана от предходното число в сегмента. Така горните три сегмента имат дължини, съответно, 4, 3, и 3. Нула, която не е край на сегмент ще наричаме сегмент с дължина 1 (тази на позиция 8 е такава).

За да получим отговор, трябва да „навържем“ сегментите в цикли, така че дължината на всеки от циклите да дели К. Реално можем да свържем и един единствен сегмент със себе си, като „насочим“ терминиращата му нула да сочи в началото на сегмента. Така, например, можем да сложим 7-ца на мястото на терминиращата нула в първия сегмент, като получим цикъла (3 -> 9 -> 11 -> 7 -> 3) с дължина 4. Друг вариант би бил да свържем първия и третия сегмент, получавайки (3 -> 9 -> 11 -> 4 -> 10 -> 2 -> 7 -> 3), получавайки цикъл с дължина 4 + 3 = 7.

Имаме най-много 50 сегмента с дължина 1, 25 сегмента с дължина 2, 16 сегмента с дължина 3 и, in general, 50/L сегмента с дължина L.

За да пазим всички тези е твърде много, но можем да видим, че сегментите с дължина 1 (нули, които не са терминиращи за никой по-дълъг сегмент) са много полезни – можем да ги ползваме както сами по себе си (1 винаги дели K), така и да ги ползваме да довършим по-дълги цикли. Вместо да ги пазим в стейта на динамичното, ще направим динамичното по такъв начин, че да оптимизира колко най-малко такива сегменти ползваме! Така, ако динамичното върне X, и ние имаме поне X сегмента с дължина 1, то ще можем да образуваме отговор. Ако пък нямаме X, ще знаем, че задачата няма решение. В случая винаги имаме решение, но така караме динамичното да построи решение, което минимизира ползваните сегменти с дължина 1, което ни гарантира намиране на отговор. Нещо повече, така не трябва да ги пазим в самия стейт на динамичното!

След като направихме динамичното да смята най-малкия брой сегменти с дължина 1, то трябва в стейта му да пазим колко с дължина 2, 3, … N имаме. С дадените ограничения не можем да пазим битова маска кои сегменти са ползвани и кои не – тъй като най-много можем да имаме 25 сегмента с дължина 2, то тази битова маска би била твърде голяма. Реално не ни интересува кои точно сегменти сме ползвали, а само по колко от всяка дължина. Това, обаче, също не можем да направим (поне не по лесен начин), тъй като имаме 26 \* 17 \* 13 \* 11 \* 9 \* 8 \* 7 \* … 2 \* 2 варианта.

Можем, обаче, да направим нещо между двете – в едно измерение ще пазим колко сегмента с дължина 2 имаме (максимум 26), а в друго – битова маска на сегментите с дължина 3 или повече, които вече сме ползвали. Тъй като имаме най-много 50/3 = 16 такива сегмента, то като размерности на динамичната таблица това е [26][2^16].

Тъй като ще искаме да образуваме циклите сегмент по сегмент, ще ни трябва и още едно измерение, което казва „цикълът, който образуваме в момента, за сега е с дължина X”. Добавяйки и него получаваме динамична таблица с размери [26][2^16][51] – тоест 86,900,736 клетки.

Тъй като отговорът не може да е повече от 51 (имаме най-много 50 сегмента с дължина едно, всяко по-голямо число не ни интересува и считаме за INF), то можем да ползваме по един байт за клетка – правейки динамичната таблица около 87 мегабайта. За да възстановим отговора ще ни трябва и още един масив със същия размер, който пък казва дали сме ползвали сегмент с дължина 1, сегмент с дължина 2, или някой от сегментите с размер 3 или повече (и кой точно). Всяка клетка от този масив също ни стига да е с размер един байт: имаме до 16 възможности за размер 3+, още една стойност, индикираща сегмент с дължина 2, и още една, индикираща довършване на цикъла (потенциално ползвайки сегменти с дължина 1) – тоест имаме само 16 + 1 + 1 = 18 възможни стойности. Реално, можем да обединим двата масива в един от тип short, като най-младшите 5 бита пазят стойността от втория масив, а останалите пазят отговора, но така не печелим нищо.

Тъй като стейтът ни не е оптимален, много от тези клетки остават празни и не се изчисляват (наистина, няма как хем да имаме 20 сегмента с дължина 2, хем още 16 сегмента с дължина 3 или повече). Това е разхитително откъм памет, но пък ни казва, че сложността ни изобщо не е толкова висока, колкото изглежда. Най-лошият случай е когато имаме един сегмент с дължина 2 и 16 сегмента с дължина 3, за общо 131072 състояния.

Вътре в самото динамично имаме цикъл по сегментите с дължина 3 и нагоре, тоест максимум 16 операции; допълнително, имаме опцията да завършим цикъла веднага или пък до го продължим със сегмент с дължина 2. Тоест, извършваме нещо от порядъка на 131072 \* 18 = 2,359,296 function call-a, което е достатъчно малко решението да върви почти мигновено.

Малкият брой състояния позволява много различни представяния на стейта (някои по-ефективни, други не толкова – като е възможно дори с мап). Предполагам, че ако състезателите се сетят да оптимизират ползването на сегменти с дължина 1 (а не да го пазят като част от стейта), то те ще успеят да решат и цялата задача.

*Автор: Александър Георгиев*