На пръв поглед формулата от условието може да Ви се стори объркваща, но тя може да се осмисли по много прост начин. От една страна, няма нужда да възприемаме числата като редица – всяко число се определя еднозначно от *a1* и първите n на броя p-та. От друга страна, всяко от числата е просто поредица от цифри, които са напълно независими една от друга. И тъй като в задачата се търси информация за отделни цифри при поредна заявка, нека се фокусираме как всяко p, което участва в определянето на числото, влияе на съответната цифра.

Нека си мислим за p-тата като маркери, които се поставят под някоя цифра. Тогава тези маркери влияят единствено на позициите надясно (както и на цифрата, където е маркерът). И това „влияние“ е просто умножение с 2 по модул 10 – именно това е смисълът на въведената операция ⊕, когато цифрите са равни, а се оказва, че тя ще бъде прилагана само на равни цифри. Всички маркери променят по еднакъв начин цифрата (умножение с 2 по модул 10), т.е. за нас представлява интерес единствено техният брой. След като намерим този брой, отговорът на заявката е просто съответната цифра от *a1*, умножена с 2 на степен този брой.

Преминаваме към съществото на задачата. При зададени *i* и *j* трябва да намерим броят на числата *p* в интервала [*2, j*], които са по-големи или равни на *i*. След това трябва да разменим стойностите на *p1* и *pj*.

За решаването на тази задача ще използваме интервално дърво, където за всеки интервал се поддържа някаква структура, в която се пазят p-тата от интервала. Нека помислим какви изисквания имаме към тази структура. Тя трябва да ни казва колко елемента, по-малки от някакво число, има вътре. Ако си представим, че елементите вътре са сортирани, това е еквивалентно на въпроса какъв е индекса на първия елемент, който има дадената стойност (всички по-малки не ни интересуват). Ако изискванията спираха дотук, задачата би се решавала с обикновен вектор, който да се поддържа сортиран.

В задачата обаче се налага и да се извършва добавяне и премахване на елементи (така се реализира размяната). Така стигаме до (тъжното) прозрение, че ни трябва балансирано двоично дърво за търсене. В авторското решение е използван treap.

След като сме си избрали и написали (или копирали) кода на подходящо дърво за търсене, реализацията на заявките е проста – трябва само да приложим нужните функции.

Сложността на алгоритъма е Q.F.logN, където F е сложността за търсене, добавяне и триене от дървото за търсене.