АНАЛИЗ НА РЕШЕНИЕТО НА ЗАДАЧА
Frog

Задачата **Frog** беше предвидена като относително сложна в темата. Въпреки това, за нейното решаване за пълен брой точки се изискваше от състезателите да са запознати с една много добре позната техника и една широко разпространена структура данни.

На пръв поглед задачата изглежда стандартна, тъй като мнозинството от състезателите, които са правили поне няколко състезания, вече сигурно са срещали задача със жаба, скачаща по водни лилии. Всъщност задачата изглежда като много тривиално динамично оптимиране, ако изключим големите ограничения. Всъщност задачата се решава именно чрез тази техника, въпросът е как да я оптимизираме да върви в даденото времево ограничение. Дори начинаещи програмисти, запознати с техниката, биха забелязали, че не се нуждаят от нищо повече освен една едномерна таблица за да запазят стейта на задачата – а именно, на коя лилия се намира жабата. След това трябва да направят цикъл по възможните продължения (къде може да скочи Ели), за да определят оптималния отговор за дадената клетка. За съжаление това решение е твърде бавно (неговата сложност е O(N^2)), но въпреки това за него са предвидени около 30-40 точки (дори над 50, ако се имплементира хитро, с известно рязане на търсенето).

Как обаче можем да оптимизираме скоростта на решението? Трябва да приложим една от стандартните оптимизации на такъв тип задачи – оптимизация на вътрешния цикъл. Вътрешния цикъл в случая очевидно е цикълът, който проверява всички възможни продължения. Поради променливата му дължина не можем да го оптимизираме до константа, но поне можем да го сведем до O(logN).

Първо ще направим динамичното итеративно (което ще ни помогне и да избегнем stack overflow поради потенциалната голяма дълбочина на рекурсията), започвайки от последния елемент. На позиция N – 1 няма какво да правим освен да вземем стойността на лилията там. За позиция N – 2 също нямаме голям избор, освен да вземем стойността на лилията там и да отидем на клетка N – 1. За третата отзад-напред клетка обаче имаме две (потенциални) възможности – или да отидем на клетка N – 2 или на N – 1. Нека за сега разгледаме задачата ако скоковете бяха „безплатни", тоест Ели не губеше ЖУ от тях. Очевидно, по-добрата възможност за Ели е да отиде на клетката, чиито отговор е по-голям. Това важи и за повече от две клетки – ако Ели има D клетки, на които може да отиде, за нея най-добре би било да отиде на тази с най-голям отговор. Забележете, че тези D клетки са последователни и вече изчислени, следователно можем да използваме структурата данни RMQ (Range Maximum Query) с която да намираме клетката с максимален отговор за O(LOG(N)) време. В случая нейната имплементация е сравнително лесна (не повече от 20-тина реда общо за update() и query()), тъй като търсим максимума винаги между някаква клетка и единия край на таблицата.

Да се върнем на оригиналната задача, обаче, където скоковете отнемат ЖУ. Тя е малко по-сложна, но реално не изисква почти никакви промени в кода, който решава горната задача, стига да се направи следното наблюдение. Нека сме на клетка i и искаме да намерим клетката с максимален отговор от следващите Di. Ако стойностите им пазим в динамична таблица DP, то търсим максималната стойност от стойностите $DP_{i+1} – T_i$, $DP_{i+2} – 2*T_i$, $DP_{i+3} – 3*T_i$, …, $DP_{i+Di} - D_i*T_i$. Ако към всяка от тези клетки прибавим произволна константа, индексът на

клетката с максимален отговор няма да се промени. Нека прибавим константата N * Ti. Така получаваме стойностите DP_{i+1} + (N–1)*T_i, DP_{i+2} + (N–2)*T_i, DP_{i+3} + (N-3)*T_i, …, DP_{i+Di} + (N–D_i)*T_i. И както казахме, клетката с максимална стойност е същата, която би била и ако не бяхме добавяли константата. Така можем да забележим, че реално можем да прибавяме всяка стойност в RMQ-то, като я модифицираме за да изпълнява даденото свойство. Ако искаме да добавим клетка със стойност X на позиция j, то я добяваме с променена стойност X + (N – j) * Ti. Обратно, за да вземем реалната й стойност, след query-то от нея изваждаме (N – i) * Ti и получаваме X + (N – j) * Ti - (N – i) * Ti = X + Ti * (N – j – N + i) = X + Ti * (i – j) = X – Ti * (j – i), тоест точно стойността, която искахме. Единственият проблем е, че N * Ti всъщност не е константа, тъй като ако N можем да считаме за такава, то Ti се мени от клетка на клетка. Добрата новина е, че максималната стойност за Ti е много малка – едва 5 - и можем просто да пазим MAX(Ti) (тоест най-много 5) различни RMQ-та за различните стойности. Следователно след като сме намерили отговора за дадена клетка от динамичната таблица, я вкарваме във всяко от петте RMQ-та, като модифицираме стойността по подходящ начин за всяко от тях. Когато правим query пък го правим само в RMQ-то, което ни интересува (тоест това, отговарящо за Ti).

Каква е сложността на описаното решение? За всяка клетка от таблицата правим MAX(Ti) update-а и едно query, като всеки update и query е със сложност O(LOG(N)). Тоест общата сложност на алгоритъма е O(N * LOG(N) * MAX(Ti)).

Автор: Александър Георгиев