SuperMario
(решение)

Може би най-отличителното за задачата е входът, зададен чрез генератор. Предполагам не много от учениците са свикнали да се сблъскват с *толкова* голям обем данни в алгоритмична задача (особено на IOI-подобно състезание). За да се реши задачата за 100 точки трябваше да се измисли много ефективен алгоритъм. Както ще видите след малко, оказва се, че самото решение не е особено сложно. По тази причина задачата беше предвидена с минимални разлики както за група A, така и за група C. Реално за да хванете 100 точки трябва да сте хитри, а не да знаете сложни алгоритми или структури данни.

Ако не друго, задачата предлагаше много на брой възможни подходи. Основната идея беше една и съща – динамично оптимиране – но самата имплементация можеше да бъде много различна. Това, което трябваше да забележим, е че от всяка клетка (състояние) оптималният отговор до края на дъската не зависи от предходните движения – независимо как сме стигнали до текущата клетка, от тук нататък отговорът (за текущата клетка) е еднозначно определен.

Най-простият вариант би решил задачата ако ограничението на **N** беше до около 5000. Идеята е да ползваме едномерна динамична таблица dyn(N), като за всяко състояние пробваме всяко от **K**-те възможни продължения и избираме това от тях, което води до най-добър отговор. Нищо сложно в този вариант – напълно стандартно едномерно динамично. То, обаче, за съжаление е със сложност O(N*K) и би хванал едва около 35 точки в група A и 50 точки в група C.

И трите други решения, които ще покажем нататък, имплементират така наречената "оптимизация на вътрешния цикъл". Това е подход в динамичното оптимиране (а и не само), при който линейна част от алгоритъма се свежда до логаритмична или дори константа такава, ползвайки друг алгоритъм или структура данни.

Нека запълваме динамичната таблица отзад напред (тоест от клетката N-1 към 0). Отговорът за последните K клетки е стойността на самата клетка, тъй като от всяка от тези клетки можем да скочим направо извън дъската. За клетката в индекс N-K вече не можем да направим това, като имаме избор измежду следващите K клетки. Забележете, че която и от тях да изберем, отговорът е C_{N-K} + dyn(i) (за клетка i > N-K). Логично е, тогава, да изберем тази стойност от динамичната таблица (от следваите K позиции), която е най-малка. По аналогичен начин можем да видим, че това е вярно не само за клетката с индекс N-K, ами за всички клетки с индекси ≤ N-K.

Тогава защо не ползваме някоя структура данни, която позволява бързо намиране на най-малкия елемент? Възможни кандидати са STL-ските set или priority_queue, като второто е създадено именно за това. В нашето второ решение ще ползваме priority_queue, тъй като има константен достъп до най-малкия елемент (а и като цяло се държи по-бързо от set). В приоритетната опашка ще вкарваме двойки от типа (минимална_цена_до_края, индекс). Индексът ни е нужен за да знаем дали можем да достигнем до клетката с дадената минимална цена. В момента, в който тази клетка стане недостижима я махаме от приоритетната опашка (тъй като обхождаме индексите в намаляващ ред, то знаем, че никога повече няма да можем да "скочим" там). Тоест алгоритъмът е следния:

1. Вкарваме последните **K** клетки в приоритетната опашка като двойки (**Ci**, i), за всяко i = **N**-**K**+1 … **N**.
2. За всяко *i* от **N**-**K** до 1:

   a. Премахваме клетките (ако има такива) от върха на приоритетната опашка, за които индексът е по-голям от i + **K**.
   b. След изпълнението на *a)* на върха на приоритетната опашка е достижима клетка с минимална цена. Оптималната цена за текущата клетка е dyn(i) = priority_queue.top().first + $\mathbf{C_i}$ (тоест цената на текущата клетка + цената за достигане до края от най-изгодното достижимо продължение).
3. Избираме и връщаме най-ниската цена измежду индексите 1 … **K** (тях можем да достигнем "безплатно" с първия скок).

Това решение е огромно подобрение спрямо предходното. Неговата сложносте O(N*logN) и вече ще свърши работа за **N** ≤ 10,000,000. Очакваше се то да вземе около 60 точки в група A и 80 в група C.

Проблем при горното решение е, че няма как по лесен начин да държим само **K** клетки в приоритетната опашка. Ако държим O(N) двойки от тип (long long, int) паметта (в група A) бързо ще свърши. Реално решението може и да не успее да хване предвидените за него 60 точки, макар и да е достатъчно бързо, тъй като надхвърли позволената му памет.

Това можем да подобрим, като забележим, че реално можем да ползваме и друга структура данни, която да има сходни свойства, но пък позволява лесното премахване на елемент. Един вариант е вече споменатия set, но неговата константа е достатъчно висока да не води до голямо подобрение. Друг вариант е прост вариант на индексно дърво за минимум (RMQ). В него можем да пазим **K** стойности, като RMQ-то да връща минималната от тях. Забележете, че даже в случая не е нужно да имплементираме query() – винаги питаме за *целия* интервал, който можем да вземем константно. Така нужната ни памет за динамичната "таблица" е O(K), като цялото решение става O(N*logK). Така бихме взели около 75 точки в група A и 90 в група C. При по-добра имплементация (break-ване на цикъла в update()-а на RMQ-то) можем да докараме точките до 85 в A.

И сега – шок. Решението за 100 точки е по-просто като имплементация и структури данни от това с приоритетна опашка и това с индексно дърво. То ползва двустранна опашка (deque в STL) и постига O(N) по време и O(K) по памет за динамичната таблица. Реално то е модификация на решението с приоритетна опашка, като се възползва от структурата на отговорите.

За него ще разгледаме примера от условието: (2, 3, 6, 1, 2, 7) и K = 2 и решението с приоритетната опашка.

Първо сме вкарарали последните K елемента като двойките (7, 6) и (2, 5) - цена 7 от клетка с индекс 6 и цена 2 от клетка с индекс 5. Сега разглеждаме позиция 4 (с число 1). Бихме ли скочили на клетка 6 (със стойност 7), при положение, че имаме по-близка клетка с по-малка стойност до края? Не! Защо тогава ни трябва изобщо да пазим двойката (7, 6)? Еми не ни трябва – можем и да не я пазим. След като обработим клетка 4, за нея вкарваме двойката (3, 4) (цена 3 = 1 + 2, индекс 4). Бихме ли скочили на клетка 5 (със стойност 2)? Все още да, тъй като цената е по-малка от цената на текущата клетка (с цена 3). От клетка с индекс 3 (и цена 6) отговорът е 8, именно минавайки през въпросната клетка. От индекс 3 вкарваме в приоритетната опашка двойката (8, 6). От клетката с индекс 2 (и стойност 3) вече не можем да достигнем до (2, 5), затова трябва да я премахнем от приоритетната опашка. Имаме избор дали да идем до (8, 3) или (3, 4). Очевидно бихме избрали втория вариант, като

отговорът за текущата клетка става (6, 2) (6 = 3 за текущата клетка + 3 до края, минавайки през индекс 4). Има ли смисъл да пазим (8, 3), при положение, че имаме (6, 2)? Отново не – отговорът е по-голям, а индексът е по-далечен – следователно клетката е ненужна. Последно, за клетката с индекс 1 вече (3, 4) е твърде далечна, съответно не я разглеждаме и остава единствено (6, 2). Следователно отговорът за индекс 1 е (7, 1).
От първите K клетки имаме (7, 1) и (6, 2). Избираме по-малкия отговор, който е и оптималният отговор.

Какво направихме? Всеки път, когато изчислим нова клетка от динамичната таблица (реално динамичната опашка), премахваме всички възможни продължения, които никога не бихме взели. Това може лесно да се имплементира с двустранна опашка (deque). Елементите в опашката ще бъдат сортирани по цена в нарастващ ред и индекс в намаляващ (най-отпред на опашката са елементите с най-ниска цена, но пък най-далечен индекс, докато най-отзад са тези с най-близък индекс, но потенциално по-висока цена). След като сме изчислили стойността за дадена клетка я добавяме най-отзад в опашката, като преди това премахваме от края й всички с по-голяма цена. Както казахме, в началото на опашката са най-изгодните (откъм цена) клетки, но рано или късно те стават твърде далечни и трябва да ги премахнем (докато стигнем до клетка на разстояние по-малко или равно на K).

Всяка клетка влиза и излиза точно по веднъж от опашката (което става с O(1)). Намирането на най-изгодното продължение е също O(1) (просто вземаме първата клетка в опашката). Нещо повече, по всяко време в опашката пазим най-много K елемента – в момента, в който дадено продължение стане твърде далеч го премахваме. Така общата сложност става O(N) по време и O(K) по памет (за динамичната "таблица"). Това е много прост вариант на оптимизация на вътрешния цикъл с изпъкнала обвивка – реално стойностите, които пазим в двустранната опашка са "изпъкнали" – увеличават се по едната координата и намаляват по другата.

Нужната памет за предложените решения е твърде много за група A – само масивът с елементите би заел около 200 мегабайта, при ограничение от 64. Там трябва да направим допълнително (относително простото) наблюдение: задачата е симетрична. Би ли имало разлика в отговора, ако тръгвахме от дясно на най-дясната клетка и искахме да стигнем преди най-лявата? Със сигурност можем да постигнем същия отговор – просто като стъпваме на клетките, които сме избрали във варианта от ляво надясно. С подобна логика можем да видим, че това е и оптималният отговор – ако имаше по-добър, то бихме могли да ползваме неговите клетки за варианта от ляво надясно. Ползвайки това, можем да генерираме числата едно по едно и да държим само предходните **K** клетки от динамичната таблица (може би ви е направило впечатление, че ограничението за **K** беше значително по-малко от **N**). Така нужната памет за цялото решение става O(K) вместо O(N).

Последно, не бива да забравяме частния случай, в който K > N и, съответно, отговорът е 0, а също така и че отговорът в най-лошия случай е N * 1000000007, което значително надхвърля възможностите на int (тоест трябва да ползваме long long за стойностите на динамичната таблица).

Автор: Александър Георгиев