Rain
(Решение)

За решаването на задачата състезателите трябваше да се досетят как да я сведат до по-лесна такава. Доста очевидното свойство, което ни позволява да направим това, е да видим, че функцията на това дали можем да изпълним условията на задачата е монотонна. Тоест ако можем да намерим разположение на саксиите при което в не повече от K от тях да падат капки с фактор по-голям от F, то ще можем да направим същото нещо и с фактор F + 1 – просто като запазим позициите им.

Така използвайки двоично търсене свеждаме задачата до следната: нека текущата стойност на двоичното търсене ни е M. Всички капки със стойност по-голяма от M ни разделят интервала (перваза) на подинтервали с различни големини. Можем ли да поставим саксиите по такъв начин, че не повече от K от тях да не са изцяло в някой от подинтервалите?

Това всъщност е и по-сложната част от задачата – как да определим дали можем да изпълним това? Ще построим граф в който върховете са ни определени от (position, remainingK) – тоест къде по перваза сме и колко най-много още саксии можем да „унищожим". В него трябва да намерим път с дължина N (като всяко ребро ни съответства на поставянето на саксия). Това не би било проблем ако графът ни не беше ИЗКЛЮЧИТЕЛНО голям – до 2 милиона върха! За наше щастие, възползвайки се от това, че той е DAG (насочен, ацикличен граф) при внимателно строене можем да го направим изключително рядък – тоест да имаме по най-много 2 излизащи ребра от всеки връх. За целта ще ползваме нещо средно между динамично оптимиране и greedy – за всяка позиция ще пазим коя е най-дясната саксия с която можем да стигнем до там. Очевидно, ако можем да стигнем до някой връх като сме ползвали X саксии или X+Y саксии, за нас е по-изгодно да вземем пътя, в който сме използвали X+Y саксии. Това именно е greedy частта на нашето решение (всъщност правим нещо подобно на алгоритъма на Dijkstra, но нито ползваме чистия му вариант, нито ползваме приоритетна опашка). Така ако сме в дадена позиция (position, remainingK) разглеждаме два случая. Първият е да сложим текущия интервал директно започвайки от position, като там има два подслучая – ако го пресича киселинна капка или ако не. Ако не – отиваме в позиция (position + len[curPot], remainingK). Иначе отиваме в (position + len[curPot], remainingK-1). Другият вариант е ако искаме със сигурност да запазим текущата саксия е да намерим най-левият интервал, надясно от position, където можем да я поставим без да пада киселинна капка в нея. Така отиваме в позиция (position + D + len[curPot], remainingK), където D ни е някакво естествено число, което би ни позволило да сложим саксията в

някой от подинтервалите. С тази алчна стратегия ограничаваме броя излизащи ребра от всеки връх до 2.

Каква е сложността на това решение? Твърде голяма. За съжаление търсенето на позиция, където можем да сложим дадена саксия без да падат киселинни капки в нея би било линейно, а цялото ни решение – квадратично. При ограничения до 100000 това е твърде много.
Как можем да се справим с този проблем? За всяка стойност от binary search-а ще прекалкулираме едно RMQ (Range Maximum Query), което ще ни дава с каква дължина е най-дългият интервал, който започва между два индекса. Така като искаме да сложим дадена саксия някъде, можем да търсим константно в RMQ-то, вместо линейно както го правихме до сега. Това обаче не е съвсем вярно, защото не винаги най-дългият интервал е този, който ни трябва. Представете си, че искаме да сложим саксия с дължина 3, а интервалите след текущата позиция са с дължини 1, 0, 3, 2, 1, 0, 5, 4, 3, 2, 1, 0, 2 ,1 ,0. Така RMQ-то ще ни върне интервала с дължина 5, като за нас по-изгодно е да вземем по-ранния с дължина 3. Затова вместо директно да взимаме стойността от RMQ-то правим ново двоично търсене – да определим най-ранния индекс, за който има интервал с дължина по-голяма или равна на тази на саксията, която искаме да сложим. Това вече ни гарантира и оптималност на отговора.
Така постигнахме решение със сложност O(L * Log(L)) за строенето на RMQ-то и O(L * K * Log(L)) за намирането на най-дългия път. Тъй като всичко това правим в двоично търсене по L получаваме крайна сложност O(L * K * Log^2(L)). Забележете, че броят и дължината на саксиите, които са най-важните входни данни в задачата, не участват в определянето на сложността в това решение ☺
В тестовете на задачата са включени и доста от частните случаи (когато K е нула, когато N е едно, когато целият перваз е запълнен и т.н.) което би позволило на участници, които не са се сетили за истинското решение да хванат допълнително точки освен малките тестове, или пък биха взели точки на невнимателни състезатели с истинското решение.

Edit:
При тестването Иван Иванов забеляза, че ако се подходи достатъчно хитро можем да разглеждаме интервалите един след друг – използвайки стек или опашка, вместо RMQ. Така най-сложната структура данни, която ползва решението, е стек или опашка, а най-сложният алгоритъм – Binary Search. Обаче за да се измисли как точно да стане това, трябва да се помисли много ☺.
За целта трябва да се реализира динамично оптимиране по K и по N, вместо по K и по L. Построеното динамично позволява дори да изнесем цикълът по K като външен и да съкратим таблицата само до два реда – O(N) по памет! Разгледайте авторовото решение за детайли. Неговата сложност е O(Log(L) * L * N).

Автор: Александър Георгиев