Анализ

Темпото в анализа ще е такова, че решението да е максимално достъпно, защото на много места долният подход не е обяснен достатъчно добре. Той се среща в не една задача.

Подзадача №0

Както винаги, оставих подзадачата с тестовите примери за обратна връзка от системата.

Подзадача №1

Тази подзадача е основата за всяко следващо решение. За целта трябва да видим каква е оптималната стратегия за почистване с една торба.

Все някога ще трябва да стигнем до последната позиция в редицата. Същевременно, за да стигнем до нея, ние вече трябва да сме събрали всички листа преди нея. Така, каквото и да правим, в момента, в който стигаме последната позиция, ще трябва да имаме $W = V_1 + V_2 + \cdots + V_{N-1} + C$. С този аргумент можем да докажем, че винаги е оптимално при $K = 1$ да следваме стратегията, в която почистваме всички листа от ляво надясно, като изначално ходим към края на редицата, почистваме подред купчините с листа и впоследствие, след като сме почистили тротоара, се връщаме към източника на торби. Самото доказателство на стратегията е оставено за упражнение на читателя.

Сега остава да изчислим колко струва тази стратегия. За целта ще третираме всяко листо от една и съща купчина по един и същи начин (защото ще се движи по един и същи начин с другите листа) и ще пресметнем движението им до N, като после ще добавим и движението им до източника:

$$C \cdot 2 \cdot N + \sum_{i=1}^{N} V_i \cdot (N - i) + \sum_{i=1}^{N} V_i \cdot N =$$

$$C \cdot 2 \cdot N + \sum_{i=1}^{N} V_i \cdot N - \sum_{i=1}^{N} V_i \cdot i + \sum_{i=1}^{N} V_i \cdot N =$$

$$2 \cdot N \cdot \left(C + \sum_{i=1}^{N} V_i - \sum_{i=1}^{N} V_i \cdot i \right.$$

Постигната сложност: $O(N)$

Имплементация: `cleanup_first_5p.cpp`

Подзадача №2

Можем да генерализираме идеята от Подзадача 1, като забележим, че при наличието на K торби ще почистим тротоара, като разбием редицата на подредици от съседни купчини листа, като за всяка подредица приложим стратегията от Подзадача 2. По-точно, с първата торба ще почистим всичко до определена позиция P_1, с втората ще почистим всичко до друга позиция P_2, ⋯, с M-тата торба ще почистим до P_M-тата позиция $(M \leq K)$. Така, за да улесним пресмятането на отговора, ще дефинираме $cost'(L, R)$ да бъде равно на цената за почистване на листата до R-та позиция, ако всички купчини преди L-тата са останали без листа. По подобие на разписването на първата подзадача можем да заключим, че:

$$cost'(L, R) = 2 \cdot R \cdot \left(C + \sum_{i=L}^{R} V_i \right) - \sum_{i=L}^{R} V_i \cdot i$$

Наистина теглото на торбата изминава $2 \cdot R$ единици разстояние, а теглото на листата от купчина i изминава $(R - i) + R = 2R - i$ разстояние до крайната си дестинация. Така може да пресметнем колко

е сумарният $cost'$ за всяко възможно разбиване на редицата на подмасиви. Броят на разбиванията е 2^{N-1}, откъдето идва сложността:

Постигната сложност: $O(2^N \cdot N)$

Имплементация: `cleanup_2n_8p.cpp`

Подзадача №3

Забележете, че при всяко едно разбиване на редицата всеки елемент участва в интервалите точно веднъж. Оттам членът $\sum_{i=L}^{R} V_i \cdot i$ в $cost'$ ще мине през всеки един член на редицата точно веднъж. Така, ако въведем нова $cost$ функция:

$$cost(L, R) = 2 \cdot R \cdot \left(C + \sum_{i=L}^{R} V_i \right)$$

и сметнем цената на минималното разбиване с новата $cost$ функция, равна на X,

то отговорът на задачата е $A = X - \sum_{i=1}^{N} V_i \cdot i$.

Така оттук насетне в анализа ще разглеждаме решение, занимаващо се с изчисление по новия $cost$, с презумпцията, че винаги след това го свеждаме до истинския отговор.

Нека построим префиксните суми на редицата $P_i = \sum_{j=1}^{i} V_j$. Тогава нашата $cost$ функция става равна на

$$cost(L, R) = 2 \cdot R \cdot (C + P_R - P_{L-1})$$

Това ще ни улесни пресмятането занапред.

Вместо чрез пълно изчерпване, ние можем да решим задачата чрез динамично програмиране със стейт `[достигната позиция][брой използвани торби]`. Пресмятането го правим за $O(N)$ време чрез вътрешен for цикъл, който да пресметне:

$$dp[reached][bags] = MIN_{prev=1}^{reached} dp[prev - 1][bags - 1] + cost(prev, reached)$$

Постигната сложност: $O(N^2K)$

Имплементация: `cleanup_nnk_18p.cpp`

Подзадача №4

За следващите подзадачи е нужен опит с оптимизирането на динамични като тази задача. Ако нямате опит в това, бих ви посъветвал да се упражните по нея.

Най-често динамични се оптимизират с по-бързо изчисление на вътрешния цикъл. Например, в немалко задачи, където се търси MIN от вътрешния цикъл в рекурентната зависимост, нерядко може да се използва сегментно дърво за по-бързо пресмятане на стойността, която намираме с вложеното обхождане. Това би било възможно за нашата задача, когато се изпълняват две условия:

1. Стойностите на динамичните спрямо $(L, bags_L)$, от които ще търсим минимум по определената от нас рекурентна зависимост, трябва да са наредени по „хубав“ начин. Това е необходимо, защото масово структурите от данни в състезателната информатика работят добре само със заявки, които са относително прости. Например, едно сегментно дърво много лесно би намерило сбора на числата на позиции x, за които $L \leq x \leq R$ (тоест, сбор в интервала), и много по-трудно някоя структура ще се справи с това да поддържа сбора на числата на позиции x в интервал $[L, R]$, такива че $x - L + 1$ е просто число. В немалко динамични наредбата на стейтовете, търсени от

рекурентната зависимост, е твърде неудобна, за да се използва структура, и трябва да се търси друг вид оптимизация. В тази задача е хубаво че се интересуваме само от префикса на редицата $dp[0][bags-1], dp[1][bags-1], \cdots$, което е може би най-удобната структура, която може да ни бъде възложена от рекурентната зависимост.

2. Сегментното дърво да е достатъчно силно, за да поддържа минимум спрямо формулата на рекурентната зависимост.

За да видим дали 2. е изпълнено, можем да си поиграем със сметката на рекурентната зависимост, за да видим какво точно се изисква от нас да поддържаме:

$$dp[R][bags] = MIN_{L=1}^{R} dp[L-1][bags-1] + cost(L,R)$$
$$dp[R][bags] = MIN_{L=1}^{R} dp[L-1][bags-1] + 2 \cdot R \cdot (C + P_R - P_L)$$
$$dp[R][bags] = MIN_{L=1}^{R} dp[L-1][bags-1] + 2 \cdot R \cdot (C + P_R) - 2 \cdot R \cdot P_L$$
$$dp[R][bags] = 2 \cdot R \cdot (C + P_R) + MIN_{L=1}^{R} dp[L-1][bags-1] - 2 \cdot R \cdot P_L$$

Може да забележим, че каквото и да правим със сегментно дърво, то няма да може да се справи с ефикасното пресмятане на $2 \cdot R \cdot P_L$. Заради това би ни трябвала друга структура. Оттук би дошъл следващият въпрос — какъв обект представлява рекурентната зависимост. Нека фиксираме $(L-1, bags-1)$ и разгледаме как би изглеждала горната сметка спрямо всички $(R, bags)$, които биха взели предвид $(L-1, bags-1)$ като възможен предишен стейт в горната рекурентна зависимост. Тогава може да я пренапишем в:

$$dp[R][bags] = 2 \cdot R \cdot (C + P_R) + MIN_{L=1}^{R} R \cdot A_L + B_L$$

Където $A_L = -2 \cdot P_L$, $B_L = dp[L-1][bags-1]$.

Така нещата, от които ще избираме ляв край, всъщност са геометрични прави. Сега въпросът е как да боравим с тези прави.

По-формално, можем да дефинираме $f_L(R) = R \cdot A_L + B_L$, където (A_L, B_L) са константи, определени по горните изчисления. Така рекурентната ни зависимост се превръща в:

$$dp[R][bags] = 2 \cdot R \cdot (C + P_R) + MIN_{L=1}^{R} f_L(R)$$

Ако за всеки R трябва да намерим коя права $f_L(R)$ (за кой L) има минимална стойност, то ние на практика си създаваме следната задача при изчисление на $dp[0][bags], dp[1][bags], \cdots, dp[N][bags]$ — да започнем с празно множество от линейни функции и итеративно да повтаряме следните две операции (интуитивно подобно на построяване на префиксни суми):

- Преди изчислението на $dp[R][bags]$ да добавим правата за $R, bags-1$ към множеството.
- За изчислението на $dp[R][bags]$ чрез подходяща структура да намираме минималната стойност на права в множеството за $x = R$.

Оказва се, че не трябва да откриваме топлата вода и такава структура вече съществува. Тя е Convex hull trick[1]. Разгледайте материалите по-долу, ако не сте запознати с нея, защото останалата част на анализа ще разчита, че сте.

Директната имплементация на CHT дава сложност:

Постигната сложност: $O(NKlog_2N)$

Имплементация: `cleanup_nklog_41p.cpp`

[1]Материали на английски за техниката: CP-algorithms, USACO guide. Ако не разбирате добре английския, използвайте подходящ преводач (или изкуствен интелект).

Подзадача №5

За тази подзадача е предвидено да се оптимизира CHT да стане линейно. Същевременно, решение на база Разделяй и владей оптимизация[2] е с много лек лог и не изглежда разграничимо от NK решението. Същевременно, едно решение с $NK \log N$ сложност също изглежда невъзможно за разграничаване, защото log *частта е по-бърза от останалите. В резултат е напълно възможно да сте изкарали и тази подзадача с различна сложност от $O(NK)$.*

Едната оптимизация на CHT-то е да намираме пресичанията на прави с $O(1)$ сметка. Интересен факт е, че дробното деление е много по-бързо от нормалното. Другата е да забележим, че във всеки един момент ние правим заявки за нарастващи X-ове, та вместо всеки път да правим двоично търсене, за да намерим оптималната права, можем да направим еквивалента на показалки в тази структура и да правим следното — всеки път да проверяваме дали втората права е по-добра от първата и в такъв случай да премахваме първата, до момента в който не стигнем до най-добрата права.

Постигната сложност: $O(NK)$

Имплементация: `cleanup_nklog_50p.cpp`

Подзадача №6 и №7

Останалата част от решението изисква познанието на `Aliens trick`[3].

В такива задачи се налага да докажем, че dp е изпъкнало или вдлъбнато. Интересното е, че има статия[4], която казва, че ако $cost$ функцията изпълнява четириъгълното неравенство[5] в някоя посока, то динамичното по функцията е задължително изпъкнало или вдлъбнато спрямо посоката на неравенството. Оттам е изключително лесно да се провери, че нашата $cost$ функция дава изпъкнало динамично.

Различното от стандартната постановка при `aliens trick` е, че оптималният брой торби не е винаги равен на K, което третият примерен тест показва.

Директно приложеният `aliens trick` би ни дал сложност от $O(N \log_2^3 N)$ или $O(N \log_2^2 N)$. Ако използваме `aliens trick` за намиране на отговора за фиксирано $M \leq K$, можем да се възползваме от изпъкналостта на dp, като правим двоично търсене по отговора, за да намерим оптималния брой торби M, като проверяваме дали $Answer\_for(K = M) > Answer\_for(K = M + 1)$. За щастие или нещастие и двете решения са твърде бавни дори за $N = 10^5$, та трябва да се измисли нещо по-добро.

Чувството, че се правят вложени двоични търсения по отговора, би трябвало да подсети състезателя, че тъй като естеството на двете двоични търсения е изключително сходно, те би трябвало да се овъркилват.

Оттам мисля, че е натурално за човек, който е зациклил на горната идея, да даде крачка назад и да се върне към основната идея на `aliens trick` и да се сети, че когато пресмятаме динамичното с $penalty = 0$, ние всъщност намираме решението за $K = N$. Така, за да се справим със случая, в който оптималният брой торби е $< K$, веднъж извикваме динамичното за решение при $penalty = 0$, като то ще ни даде оптималния брой интервали за $K = N$. Ако той надвишава K, ние ще търсим отговора за брой интервали $= K$.

Разликата между шеста и седма подзадача е в това дали състезателят е написал CHT за линейно или за $O(N \log_2 N)$ време.

Постигната сложност: $O(Nlog\_2N)$

Имплементация: `cleanup_nlog_100p.cpp` *Автор: Борис Михов*

---

[2] Материали: CP algorithms, USACO guide

[3] Материали: Хубав блог, USACO guide

[4] Линк

[5] $cost(a,c) + cost(b,d) \leq cost(a,d) + cost(b,c)$ за всички $1 \leq a \leq b \leq c \leq d \leq N$