Jan Kanty Milczek
Treść zadania, Opracowanie

Michał Zgliczyński
Program

Dostępna pamięć: 128 MB. *OI, etap II, dzień pierwszy, 8.02.2012*

Szatnia

W Bajtocji odbywa się doroczny zlot bogatych mieszkańców. Zbierają się oni, by chwalić się swoimi zarobkami, butami Lebajtina i innymi luksusowymi rzeczami. Naturalnie, nie wszystkie rzeczy wnoszą na bankiet — płaszcze, kurtki czy parasole zostawiają w szatni, by odebrać je, wychodząc.

Na nieszczęście bogaczy szajka bajtockich złodziei planuje włamać się do szatni i ukraść część pozostawionych tam przedmiotów. Póki co szef szajki przegląda zaproponowane przez gangsterów plany skoku. Każdy plan wygląda następująco: złodzieje pojawiają się w szatni w chwili m_j, zabierają przedmioty o wartości **dokładnie** *k_j i uciekają, przy czym cały skok zajmuje im czas s_j. Szef gangu chciałby przede wszystkim wiedzieć, które z planów mają szansę się udać, a które nie. Plan ma szansę się udać, jeśli w chwili m_j da się uzbierać przedmioty o łącznej wartości dokładnie k_j, w taki sposób, żeby do momentu $m_j + s_j$ włącznie nikt nie przyszedł po żaden z kradzionych przedmiotów (w takim wypadku zawiadomiłby ochronę i ucieczka nie udałaby się). W szczególności, jeśli w chwili m_j w ogóle nie da się dobrać przedmiotów o łącznej wartości k_j, plan zostaje odrzucony. Znając moment przyniesienia i zabrania każdego przedmiotu, określ, które plany mają szansę powodzenia, a które skazane są na porażkę. Zakładamy, że jeśli przedmiot zostaje przyniesiony w momencie, w którym złodzieje mają dokonać skoku, mogą go już ukraść (patrz test przykładowy).*

Wejście

W pierwszym wierszu standardowego wejścia znajduje się liczba całkowita n ($1 \leqslant n \leqslant 1\,000$), reprezentująca liczbę przedmiotów, które zostaną pozostawione w szatni. W kolejnych n wierszach znajdują się opisy przedmiotów. Każdy z nich składa się z trzech liczb całkowitych c_i, a_i i b_i ($1 \leqslant c_i \leqslant 1\,000$, $1 \leqslant a_i < b_i \leqslant 10^9$), pooddzielanych pojedynczymi odstępami, oznaczających kolejno: wartość przedmiotu, moment, w którym przedmiot zostanie przyniesiony do szatni, oraz moment, w którym zostanie z niej zabrany.

Następny wiersz zawiera liczbę p ($1 \leqslant p \leqslant 1\,000\,000$) — liczbę planów przedstawionych przez szajkę. Każdy z nich jest opisany w jednym wierszu przez trzy liczby całkowite m_j, k_j i s_j ($1 \leqslant m_j \leqslant 10^9$, $1 \leqslant k_j \leqslant 100\,000$, $0 \leqslant s_j \leqslant 10^9$), pooddzielane pojedynczymi odstępami, oznaczające kolejno: moment, w którym złodzieje weszliby do szatni, wartość, jaką chcą uzyskać, oraz czas, jaki zajmie im kradzież.

W testach wartych 16% punktów zachodzi dodatkowy warunek $p \leqslant 10$.

W innych testach, również wartych 16% punktów, wszystkie przedmioty mają równe a_i.

W jeszcze innych testach, wartych 24% punktów, wszystkie zapytania mają równe s_j.

Wyjście

Dla każdego planu szajki określ, czy plan da się zrealizować, tj. ukraść przedmioty o wartości dokładnie k_j i uciec, zanim ktoś upomni się o swoją rzecz. Jeśli plan jest wykonalny, Twój program powinien wypisać na standardowe wyjście słowo `TAK`*, a w przeciwnym wypadku słowo* `NIE`*.*

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
5	TAK
6 2 7	NIE
5 4 9	TAK
1 2 4	TAK
2 5 8	NIE
1 3 9	
5	
2 7 1	
2 7 2	
3 2 0	
5 7 2	
4 1 5	

Rozwiązanie

Wprowadzenie

Mamy zadany zbiór elementów, z których każdy jest scharakteryzowany przez trzy parametry c_i, a_i, b_i, oznaczające odpowiednio wartość elementu, moment pojawienia się oraz moment zniknięcia. Dla takiego zbioru elementów mamy określone zapytania postaci (m, k, s): czy w chwili m istnieje podzbiór elementów, które sumują się do k oraz będą istniały przez cały okres czasu $[m, m + s]$.

Rozwiązanie

Nasze zadanie jest wariacją na temat *problemu pakowania plecaka* (patrz np. [23]), możemy więc spodziewać się rozwiązania korzystającego z programowania dynamicznego. W klasycznej wersji problemu każdy przedmiot miał jednak dwie charakterystyczne dla siebie cechy — cenę c i wagę w. U nas przedmioty mają jedną cechę — wartość, a w dodatku określone są dla nich momenty pojawienia się i zniknięcia. Będziemy próbować poradzić sobie z kolejnymi trudnościami.

Brak wagi

Oczywiście w praktyce nie jest to problem, wręcz przeciwnie. Gdybyśmy mogli zupełnie pominąć problem czasu i po prostu odpowiadać na pytania, czy z zestawu n przedmiotów możemy uzyskać sumę k, wystarczyłoby przeglądać kolejne przedmioty i zapisywać odpowiedź, czy z pierwszych i przedmiotów możemy uzyskać sumę k. Zależność jest prosta — albo mogliśmy tę sumę uzyskać już za pomocą $i-1$ przedmiotów, albo uzyskamy ją poprzez dodanie i-tego przedmiotu do zbioru uzyskiwalnego $i - 1$ przedmiotami. Podejście to realizuje następujący pseudokod (zakładamy, że wartości przedmiotów są zapisane w tablicy C).

```
 1: function CzySieDa(k, n, C)
 2: begin
 3:   DaSie[0] := true;
 4:   for j := 1 to k do
 5:     DaSie[j] := false;
 6:   for i := 1 to n do
 7:     for j := k downto C[i] do
 8:       if DaSie[j − C[i]] then
 9:         DaSie[j] := true;
10:   return DaSie[k];
11: end
```

Czas dodania i czas zabrania

Te parametry wydają się symetryczne, więc zapewne warto traktować je w podobny sposób. Spróbujmy na początek skonstruować dowolne (być może za wolne) poprawne rozwiązanie. Możemy w tym celu wykorzystać naszą funkcję *CzySieDa*. Zauważmy, że w przypadku zapytania postaci (m, k, s) nie możemy użyć żadnego przedmiotu o czasie dodania a większym niż m, jak i żadnego, którego czas zabrania b jest mniejszy niż $m + s + 1$. Wszystkie inne przedmioty wykorzystać możemy — będą już w szatni i nikt nie zorientuje się na czas, że zostały zabrane. Następujący pseudokod w sposób siłowy wykorzystuje tę obserwację (tablice A i B przechowują czasy dodania i zabrania poszczególnych przedmiotów).

```
 1: function Plan(m, k, s, n, C, A, B)
 2: begin
 3:   IleOdpowiednich := 0;
 4:   for i := 1 to n do
 5:     if (A[i] ⩽ m) and (B[i] ⩾ m + s + 1) then begin
 6:       IleOdpowiednich := IleOdpowiednich + 1;
 7:       OdpowiednieC[IleOdpowiednich] := C[i];
 8:     end
 9:   return CzySieDa(k, IleOdpowiednich, OdpowiednieC);
10: end
```

Ten algorytm działa w czasie $O(n \cdot k)$ dla każdego zapytania, łącznie w czasie $O(p \cdot n \cdot k_{max})$. Treść zadania sugeruje, że ma on szansę dostać 16% punktów. Aby poprawić czas działania naszego algorytmu, musimy przestać rozpatrywać wszystkie przedmioty osobno dla każdego zapytania. Spróbujmy rozwiązać zadanie symulacyjnie — dokładać przedmioty do szatni według ich a, odpowiadać na zapytania według ich m i zabierać przedmioty według ich b. Niestety pozostawalibyśmy w momencie zapytania bez informacji, kiedy wybierane przez nas przedmioty znikną z szatni. Nie trzeba się zrażać — w testach wartych 24% punktów wszystkie plany mają równe s. Jest to równoważne sytuacji, w której dla wszystkich planów zachodzi $s = 0$ — bo wystarczy zmniejszyć o s czas zniknięcia b każdego przedmiotu. Załóżmy teraz, że każdy plan ma faktycznie $s = 0$, co sprowadzi nasze zadanie do odpowiadania, czy z przedmiotów znajdujących się w szatni w momencie m da się wybrać zbiór o łącznej

wartości k. Zauważmy, że o ile kod wewnętrznej pętli z funkcji *CzySieDa* można by bez modyfikacji przyjąć jako kod dodania nowej rzeczy do szatni, to powstałby problem przy zabieraniu przedmiotu — czy wartość **true** w $DaSie[j]$ była uzyskana tylko z wykorzystaniem tego przedmiotu, czy może bez niego, a może na oba sposoby? Aby rozwiązać ten dylemat, możemy przechowywać tam nieco inną wartość. Zamiast zapisywać, czy w ogóle da się uzyskać daną sumę, zapiszemy tam, **na ile sposobów** da się ją uzyskać. Otrzymamy następujący kod dodania przedmiotu:

```
1: procedure DodajPrzedmiot(c)
2: begin
3:   for i := k_max downto c do
4:     Sposobow[i] := Sposobow[i] + Sposobow[i − c];
5: end
```

Operacja usunięcia przedmiotu nie może oczywiście być zupełnie analogiczna; przy takim podejściu moglibyśmy otrzymać kod:

```
1: procedure UsunPrzedmiotZle(c)
2: begin
3:   for i := k_max downto c do
4:     Sposobow[i] := Sposobow[i] − Sposobow[i − c];
5: end
```

Ten kod nie działa, ponieważ zmniejsza liczbę sposobów uzyskania podzbioru o wartości i o liczbę sposobów uzyskania wartości $i - c$ z wykorzystaniem zabieranego przedmiotu. Pętla musi zostać odwrócona, a procedura zadziała poprawnie:

```
1: procedure UsunPrzedmiot(c)
2: begin
3:   for i := c to k_max do
4:     Sposobow[i] := Sposobow[i] − Sposobow[i − c];
5: end
```

Dla formalności można też zapisać kod obsługi zapytania:

```
1: function Plan(k)
2: begin
3:   return (Sposobow[k] ≠ 0);
4: end
```

To rozwiązanie powinno wywołać w czytelniku wątpliwość — czy wartości tablicy *Sposobow* nie będą rosły bardzo szybko, wymuszając użycie dużych liczb? Oczywiście tak się stanie, a implementacja dużych liczb mocno spowolniłaby rozwiązanie. Możemy poradzić sobie z tym problemem inaczej — przechowywać w tablicy *Sposobow* wartości reszt z dzielenia liczby sposobów uzyskania sumy przez dużą, ale mieszczącą się (dwukrotnie) w zakresie obliczeń, liczbę pierwszą. Występuje wtedy ryzyko, że nasz program pomyli się (jeśli przy którymś zapytaniu liczba sposobów będzie podzielna przez tę liczbę pierwszą), ale jest to zagrożenie bardzo małe (im większa liczba pierwsza, tym mniejsze). Warto więc podjąć takie ryzyko wobec przyspieszenia, jakie w ten sposób uzyskamy: rozwiązanie działa w czasie $O(n \cdot k_{max} + p)$.

Niestety, cały czas rozważaliśmy sytuację, w której wszystkie wartości s dla zapytań są równe. Gdy tak nie jest, musimy przy dodawaniu elementów przechować informację o tym, w którym momencie przedmiot zostanie zabrany. Oznacza to w praktyce, że operacje dodania i zabrania elementu nie będą symetryczne ani analogiczne. Rozbijmy je więc.

Czas dodania

W poprzednim podejściu próbowaliśmy rozpatrywać kolejne wydarzenia na osi czasu i nie ma powodu, by odejść od tego podejścia w przypadku dodawania przedmiotów do szatni. Będziemy poruszać się po osi, wykonując albo operacje dodania elementu, albo próby zrealizowania planu.

Czas zabrania

Wyobraźmy sobie inną sytuację. Goście imprezy nie przynoszą rzeczy, ale warzywa, a nasi złodzieje nie chcą kraść, ale zrobić sałatkę. Wiadomo, że data przydatności sałatki do spożycia to data przydatności najmniej trwałego składnika. Goście będą więc w momencie a przynosić składnik o wartości c i dacie przydatności w. Złodzieje w momencie m będą próbowali zrobić sałatkę o łącznej wartości k i dacie przydatności co najmniej r. Nie mamy problemu z tym zadaniem. Używając programowania dynamicznego, możemy określać przy dodawaniu każdego następnego składnika datę przydatności najtrwalszej sałatki, jaką da się uzyskać w danym momencie dla danego sumarycznego kosztu. Dodanie nowego składnika realizuje następujący kod:

```
1: procedure NowySkladnik(c, w)
2: begin
3:   for i := k_max downto c do
4:     Najtrwalsza[i] := max(Najtrwalsza[i], min(Najtrwalsza[i − c], w));
5: end
```

A zapytanie jest realizowane przez następującą funkcję:

```
1: function Salatka(k, r)
2: begin
3:   return (Najtrwalsza[k] ⩾ r);
4: end
```

Co to ma wspólnego z naszym zadaniem? Otóż wystarczy podstawić b jako w i $m + s + 1$ jako r i okazało się, że rozwiązaliśmy zadanie. Po posortowaniu wejścia, co zajmie czas $O((n + p) \cdot \log(n + p))$, pozostaje realizować fragmenty *NowySkladnik*, które zajmują czas $O(k_{max})$, i *Salatka*, które działają w czasie stałym. Łączny czas działania algorytmu to zatem $O((n + p) \cdot \log(n + p) + n \cdot k_{max} + p)$. Implementacje rozwiązania można znaleźć w plikach `sza.cpp`, `sza1.cpp`, `sza2.cpp` i `sza3.pas`.

Testy

Zadanie było sprawdzane na 12 zestawach danych testowych, patrz tabela poniżej.

Nazwa	n	p
sza1.in	20	10
sza2.in	500	10
sza3.in	777	12 372
sza4.in	567	54 336
sza5.in	198	354 420
sza6.in	187	228 608
sza7.in	200	123 200
sza8.in	98	712 068
sza9.in	666	123 504
sza10.in	777	300 312
sza11.in	950	499 700
sza12.in	1 000	1 000 000