Piotr Stańczyk
Treść zadania, Opracowanie

Piotr Stańczyk
Program

OI, Etap II, dzień pierwszy, 7.02.2008

Pociągi

W Bajtocji odbędzie się Parada Kolorowych Pociągów. Na torach technicznych bajtockiego dworca trwają intensywne przygotowania. Na dworcu jest n równoległych torów, ponumerowanych od 1 do n. Na i-tym torze ustawiono pociąg o numerze i. Każdy pociąg składa się z l wagonów, z których każdy jest pomalowany na jeden z 26 kolorów (oznaczonych małymi literami alfabetu angielskiego). Mówimy, że dwa pociągi **wyglądają identycznie**, *jeśli ich kolejne wagony są tego samego koloru.*

Parada będzie polegać na tym, że co minutę stacyjny dźwig zamieni miejscami pewną parę wagonów. Prawdziwa parada odbędzie się jednak dopiero jutro. Dziś dyżurny ruchu Bajtazar bacznie przyglądał się próbie generalnej. Dokładnie zapisał sobie ciąg kolejno zamienianych par wagonów.

Bajtazar nie lubi, gdy zbyt wiele pociągów wygląda identycznie. Chciałby, żebyś dla każdego pociągu p policzył maksymalną liczbę pociągów, które w pewnym momencie wyglądają identycznie jak pociąg p w owym momencie.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia opisy pociągów stojących na torach oraz ciąg wykonywanych zamian wagonów,*
- *dla każdego pociągu wyznaczy maksymalną liczbę pociągów, które w pewnym momencie wyglądają tak samo jak on,*
- *wypisze wynik na standardowe wyjście.*

Wejście

Pierwszy wiersz wejścia zawiera trzy liczby naturalne n, l oraz m ($2 \leqslant n \leqslant 1\,000$, $1 \leqslant l \leqslant 100$, $0 \leqslant m \leqslant 100\,000$), oznaczające odpowiednio liczbę pociągów, ich długość oraz liczbę wykonywanych zamian wagonów. W kolejnych n wierszach znajdują się opisy kolejnych pociągów stojących na torach. k-ty z tych wierszy składa się z l małych liter alfabetu angielskiego reprezentujących kolory kolejnych wagonów k-tego pociągu. Za opisami pociągów znajduje się m wierszy zawierających opisy kolejnych zamian, w kolejności ich wykonywania. W r-tym z tych wierszy znajdują się cztery liczby całkowite p_1, w_1, p_2, w_2 ($1 \leqslant p_1, p_2 \leqslant n$, $1 \leqslant w_1, w_2 \leqslant l$, $p_1 \neq p_2$ lub $w_1 \neq w_2$) i opisują one r-tą operację wykonywaną przez dźwig — zamianę wagonu numer w_1 z pociągu p_1 z wagonem w_2 pociągu p_2.

Wyjście

Twój program powinien wypisać dokładnie n wierszy. k-ty wiersz powinien zawierać jedną liczbę całkowitą — liczbę pociągów wyglądających tak samo jak pociąg numer k w pewnym momencie czasu.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
5 6 7	3
ababbd	3
abbbbd	3
aaabad	2
caabbd	3
cabaad	
2 3 5 4	
5 3 5 5	
3 5 2 2	
1 2 4 3	
2 2 5 1	
1 1 3 3	
4 1 5 6	

Oto wygląd pociągów w kolejnych fazach próby generalnej:

```
tor 1:  ababbd    ababbd    ababbd    ababbd    aaabbd    aaabbd    aaabbd    aaabbd
tor 2:  abbbbd    ababbd    ababbd    aaabbd    aaabbd    acabbd    acabbd    acabbd
tor 3:  aaabad -> aaabad -> aaabad -> aaabbd -> aaabbd -> aaabbd -> aaabbd -> aaabbd
tor 4:  caabbd    caabbd    caabbd    caabbd    cabbbd    cabbbd    cabbbd    dabbbd
tor 5:  cabaad    cabbad    caabbd    caabbd    caabbd    aaabbd    aaabbd    aaabbc

           (0)       (1)       (2)       (3)       (4)       (5)       (6)       (7)
```

Dla pociągów 1, 2 i 3 najwięcej podobnych było np. w momencie (4) *(były one nawzajem do siebie podobne). Dla pociągu 5 najwięcej mu podobnych było w momentach* (5) *i* (6)*. Dla pociągu 4 najwięcej podobnych było np. w momencie* (2)*.*

Rozwiązanie

Pierwszy pomysł przychodzący na myśl po przeczytaniu treści zadania to rozwiązanie go za pomocą prostej symulacji. Wystarczy napisać program, który będzie wykonywał przestawienia wagonów zgodnie z opisem zawartym w danych i po każdej zamianie będzie sprawdzał, dla każdego pociągu, ile pociągów wygląda tak samo jak on. Będzie także na bieżąco wyznaczał dla każdego pociągu *wynik* — maksymalną liczbę pociągów do niego podobnych w trakcie dotychczas wykonanych kroków symulacji. Dla różnych sposobów realizacji tego pomysłu otrzymamy algorytmy o różnej złożoności czasowej. Na efektywność obliczeń mają wpływ reprezentacja pociągów (czyli struktura danych) oraz

sposoby wykonywania przestawiania wagonów i wyznaczania liczby podobnych pociągów. Rozpocznijmy od analizy prostej symulacji, którą następnie, w kolejnych podrozdziałach opracowania, będziemy stopniowo ulepszać.

W opisie rozwiązania będziemy stosować takie same oznaczenia, jak w treści zadania, a więc n będzie liczbą pociągów, l — długością pociągu (wszystkie są jednakowej długości), natomiast m — liczbą wykonywanych przestawień wagonów. Wprowadzamy także jedno dodatkowe oznaczenie

$$f = nl + m\log l, \tag{1}$$

z którego będziemy korzystać w trakcie analizy złożoności kilku algorytmów (wtedy wyjaśni się, skąd się ta wartość bierze).

Rozwiązanie o złożoności czasowej $O(n^2 \cdot l \cdot m)$

Najbardziej naturalną reprezentacją danych jest zapisanie pociągu jako ciągu l małych liter alfabetu angielskiego, czyli słowa długości l. Taka reprezentacja jest prosta w implementacji i na dodatek pozwala łatwo i efektywnie przestawiać wagony w pociągach — wystarczy zamienić miejscami dwie litery w słowach. Bardziej czasochłonna okazuje się druga operacja — wyznaczenie dla pociągu liczby pociągów do niego podobnych. Dla konkretnego pociągu możemy porównać opisujące go słowo z pozostałymi w czasie $O(n \cdot l)$ (w pesymistycznym przypadku porównanie dwóch słów wymaga l porównań liter), zatem czas potrzebny dla wszystkich pociągów wynosi $O(n^2 \cdot l)$. Ponieważ porównania musimy przeprowadzać po każdym przestawieniu wagonów, a przestawień wykonujemy łącznie m, to ostateczna złożoność tego algorytmu wynosi $O(n^2 \cdot l \cdot m)$ (jego implementacja znajduje się w pliku `pocs1.cpp`). Porównawszy tę złożoność z zakresem danych wejściowych, można spodziewać się, że rozwiązanie takie niestety nie uzyska zbyt dużej liczby punktów. Zastanówmy się zatem, jak je przyspieszyć.

Rozwiązanie o złożoności czasowej $O(n^2 \cdot l + n \cdot l \cdot m)$

W poprzednim rozwiązaniu dużo czasu zajęło nam porównywanie pociągów. Zauważmy jednak, że nie wszystkie wykonywane porównania są konieczne. Po przestawieniu dwóch wagonów może zmienić się liczba podobnych pociągów tylko dla tych, które uległy zmianie, i dla tych, które są do nich podobne. Z tego wynika, że wystarczy wyznaczyć podobne pociągi tylko dla (co najwyżej) dwóch zmienianych pociągów (przed przestawieniem wagonów i po nim), a następnie uaktualnić wynik dla nich i dla wszystkich pociągów do nich podobnych (tzn. podobnych przed przestawieniem bądź po nim). Symulacja przestawienia dwóch wagonów wymaga więc już tylko czasu $O(n \cdot l)$, czyli symulację wszystkich m kroków wykonujemy w czasie $O(n^2 \cdot l + n \cdot l \cdot m)$, gdzie składnik $O(n^2 \cdot l)$ to czas potrzebny na wyznaczenie liczb podobnych pociągów przed rozpoczęciem symulacji — tę operację wykonujemy tak samo jak poprzednio, przez porównanie wszystkich par pociągów.

To rozwiązanie zostało zaimplementowane w pliku `pocs2.cpp`.

Rozwiązanie o złożoności czasowej $O(n \cdot l + n \cdot l \cdot m) = O(n \cdot l \cdot m)$

Poprzednio udało nam się przyspieszyć fazę aktualizacji wyniku, spróbujmy teraz poprawić efektywność obliczania wartości początkowych. Aby wyznaczyć dla każdego pociągu

liczbę pociągów do niego podobnych, możemy najpierw posortować pociągi (a dokładniej słowa, które je opisują). Sortowanie można zrealizować za pomocą dowolnego algorytmu wykonującego $O(n \log n)$ porównań sortowanych obiektów, czyli działającego w naszym przypadku w czasie $O(l \cdot n \log n)$. Jednak dla słów nad niewielkim alfabetem znacznie bardziej praktyczne jest sortowanie pozycyjne, które działa w czasie $O(n \cdot l + l \cdot |\Sigma|)$, gdzie $|\Sigma|$ to moc alfabetu[1]. Gdy pociągi są posortowane, można łatwo w czasie $O(n \cdot l)$ wyznaczyć dla każdego z nich liczbę pociągów podobnych. W ten sposób uzyskujemy algorytm o całkowitej złożoności $O(n \cdot l + n \cdot l \cdot m) = O(n \cdot l \cdot m)$.

Rozwiązanie o złożoności czasowej $O(n \cdot l + n \cdot m)$?

Skonstruowaliśmy już algorytm n razy szybszy niż wyjściowa symulacja. Patrząc jednak na możliwe zakresy liczb n, m oraz l, należy spodziewać się, że autorzy zadania oczekują jeszcze lepszego rozwiązania. W szczególności warto zastanowić się, czy nie da się szybciej sprawdzać, czy dwa pociągi wyglądają tak samo. Porównań pociągów wykonujemy bardzo dużo — aż $O(n \cdot m)$ — zatem przyspieszenie tej operacji w istotny sposób wpłynie na złożoność całego algorytmu.

Pierwszym z pomysłów jest zastosowanie *haszowania*. Operacja ta polega na przypisaniu obiektom nowych reprezentacji (możemy je nazwać etykietami), o jak najmniejszej wartości. Zazwyczaj jest to funkcja $h : D \mapsto \{0, \ldots, N\}$, gdzie D jest zbiorem obiektów, a N — stosunkowo małą liczbą naturalną. Oczywiście haszowanie (jako funkcja) przypisuje jednakowe wartości jednakowym argumentom. Nie zawsze natomiast, choć jest to pożądane, funkcja haszująca jest różnowartościowa na zbiorze D [2]. W naszym zadaniu wykorzystamy modularną funkcję haszującą, zadaną wzorem

$$h(x) = x \bmod P,$$

gdzie P jest pewną liczbą pierwszą. Aby zastosować h dla naszych obiektów, musimy słowa opisujące pociągi zinterpretować jako liczby. W tym celu przyporządkujemy literom wartości liczbowe zadane funkcją g ($g(a) = 0$, $g(b) = 1$, …, $g(z) = 25$), a następnie rozszerzymy funkcję g na słowa, przypisując słowu $s = c_1 c_2 \ldots c_l$ wartość:

$$g(c_1 \ldots c_l) = \sum_{k=1}^{l} g(c_k) \cdot 26^{k-1}.$$

Z powyższego wzoru widać, że $g(s)$ może być bardzo dużą liczbą, zatem jej wyznaczenie wymaga zaimplementowania własnej arytmetyki na długich liczbach. Ponieważ jednak nie potrzebujemy wartości $g(s)$, a jedynie $h(g(s))$, sprawa znacznie się upraszcza:

$$\begin{aligned} h(g(c_1 \ldots c_k)) &= h\Big(\sum_{k=1}^{l} g(c_k) \cdot 26^{k-1}\Big) \\ &= \Big(\sum_{k=1}^{l} g(c_k) \cdot 26^{k-1}\Big) \bmod P \\ &= \sum_{k=1}^{l} \big((g(c_k) \bmod P) \cdot (26^{k-1} \bmod P)\big) \bmod P. \end{aligned}$$

[1] O wszystkich wspomnianych metodach sortowania można przeczytać w [15], [17] czy [20].

[2] Więcej na temat haszowania, w szczególności o dobrych funkcjach haszujących, można przeczytać np. w [20].

Co więcej, raz obliczoną wartość $h(g(s))$ można łatwo aktualizować po zamianie wagonu w pociągu. Załóżmy, że w pociągu $s = c_1 \ldots c_l$ wagon na pozycji k został zmieniony na wagon c'_k; niech s' będzie słowem opisującym pociąg po zmianie. Wówczas

$$h(g(s')) = (h(g(s)) + (c'_k - c_k) \cdot 26^{k-1}) \bmod P.$$

Zauważmy, że jeżeli na samym początku spamiętamy reszty z dzielenia potęg $26^0, 26^1, \ldots, 26^{l-1}$ przez P, to $h(g(s'))$ będziemy mogli wyznaczyć na podstawie $h(g(s))$ w czasie stałym.

Aby porównać dwa pociągi, można wstępnie sprawdzić, czy wartości ich funkcji haszujących są równe. Jeśli nie są, to pociągi na pewno są różne. Jeśli natomiast wartości są równe, to być może pociągi są takie same, ale nie mamy co do tego pewności.

W tym momencie trzeba podjąć decyzję. Albo chcemy mieć program, którego wynikom zawsze ufamy, albo też jesteśmy gotowi podjąć ryzyko błędu. W pierwszym przypadku, po pozytywnym wyniku porównania etykiet słów (czyli wartości funkcji haszującej) należy wykonać jeszcze dokładne porównanie samych słów w czasie $O(l)$. W drugim przypadku możemy pominąć sprawdzenie, uznając, że słowa o równych etykietach są równe. Wówczas nasz program może zwracać błędne wyniki, gdy zawiedzie przyjęte założenie (ma to miejsce zazwyczaj, gdy dane są dobrane „złośliwie" do funkcji lub funkcja haszująca jest po prostu źle wybrana).

W pierwszym przypadku otrzymujemy algorytm, który w najgorszym razie działa tak samo długo jak poprzednie rozwiązanie (to rozwiązanie zostało zaimplementowane w pliku `pocs3.cpp`). Sytuacja taka ma miejsce, gdy występuje wiele par podobnych pociągów. Jeśli natomiast zaryzykujemy pominięcie dokładnego sprawdzenia, to wykonujemy każde porównanie pociągów w czasie $O(1)$, czyli obliczamy rozwiązanie w sumarycznym czasie $O(n \cdot l + m \cdot n)$. Czas $O(n \cdot l)$ jest nam tym razem potrzebny na wyznaczenie początkowych wartości funkcji haszujących oraz, jak poprzednio, na policzenie początkowych wyników. Niestety otrzymujemy program *de facto* niepoprawny i testy do zadania, jeśli są odpowiednio dobrane, mogą to wykazać. Implementacja tego rozwiązania znajduje się w pliku `pocb1.cpp`.

Rozwiązanie o złożoności czasowej $O(f \log f + n \cdot m)$

Przedstawione przed chwilą rozwiązanie „ryzykowne", pomimo że szybkie i łatwe w implementacji, nie może oczywiście być rozwiązaniem wzorcowym. Możemy jednak spróbować je zmodyfikować, przypisując słowom-pociągom etykiety w sposób różnowartościowy. Jeśli zadbamy, by etykiety były krótkie i łatwe do modyfikacji przy zamianie wagonów, to będziemy mogli szybko porównywać pociągi w czasie symulacji.

Rozpatrzmy pociąg reprezentowany przez ciąg liter $s_0 = c_1 c_2 \ldots c_l$. By uzyskać etykietę słowa, będziemy wielokrotnie skracać tę reprezentację, za każdym razem zastępując pary liter pojedynczymi znakami. Do tego celu potrzebna nam będzie funkcja różnowartościowa

$$F : Id \times Id \to Id,$$

gdzie Id jest zbiorem identyfikatorów zawierającym alfabet małych liter angielskich (a raczej przypisanych im kodów $0, 1, \ldots, 25$). Teraz możemy pogrupować litery słowa w pary:

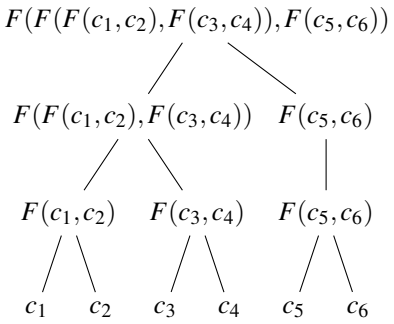
$(c_1,c_2),(c_3,c_4),\ldots$ i zastąpić słowo l-literowe przez słowo złożone z $\lceil l/2 \rceil$ liter (jeśli słowo wyjściowe było nieparzystej długości, to ostatnią literę pozostawiamy niezmienioną):

$$s_1 \;=\; F(c_1,c_2)F(c_3,c_4)\ldots.$$

Powyższą metodę możemy następnie zastosować do słowa s_1, otrzymując s_2, potem do s_2 itd., aż uzyskamy reprezentację s_k o długości 1 (oczywiście $k = O(\log l)$ i wszystkie słowa s_i są nad alfabetem Id). Z różnowartościowości F wynika od razu, że pociągom opisywanym różnymi słowami przypiszemy w ten sposób różne etykiety. Wstępne wyznaczenie etykiety dla jednego pociągu o długości l wymaga wyliczenia

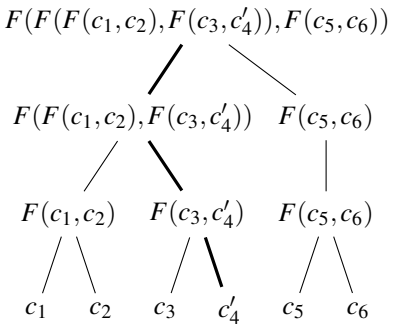
$$\left\lceil \frac{l}{2} \right\rceil + \left\lceil \frac{l}{4} \right\rceil + \ldots = O(l)$$

razy wartości funkcji F.



Rys. 1: Sposób wyznaczania etykiety dla słowa $s_0 = c_1c_2\ldots c_6$ za pomocą funkcji F.

Następnie, po zamianie pojedynczego wagonu w pociągu, wystarczy zaktualizować po jednej literze w każdym ze słów $s_0, s_1, \ldots, s_k$ — to wymaga zaledwie $O(\log l)$ wywołań F.



Rys. 2: Sposób aktualizacji słów s_i po zmianie czwartej litery wyjściowego słowa.

W całym algorytmie będziemy musieli zatem wyliczyć wartość F łącznie

$$O(nl + m\log l) = O(f)$$

razy (patrz wzór (1)).

Pozostaje zatem ostatnie pytanie — w jaki sposób zdefiniować funkcję F. Jedynym wymaganiem, jakie sobie postawiliśmy, jest jej różnowartościowość. W tej sytuacji najprostszym rozwiązaniem jest definiowanie wartości F dla kolejnych par argumentów, które się pojawiają, jako kolejnych liczb naturalnych. Możemy to zrobić, przechowując w strukturze słownikowej wszystkie napotkane dotychczas pary wraz z przypisanymi im wartościami funkcji F. Niech M oznacza maksymalną wartość funkcji przechowywaną w danym momencie w słowniku. Gdy pojawi się kolejna para (c,d), wówczas szukamy jej w słowniku. Jeśli tam się znajduje, to znamy wartość $F(c,d)$. W przeciwnym przypadku nadajemy wartość funkcji $F(c,d) = M$, dodajemy parę (c,d) wraz z wyznaczoną wartością do słownika i zwiększamy M o jeden. Słownik możemy zaimplementować w postaci zrównoważonego drzewa poszukiwań binarnych[3]. Czas wyszukiwania i wstawania elementów wynosi wówczas $O(\log r)$, gdzie r to liczba elementów w słowniku[4]. Maksymalna wielkość słownika w trakcie algorytmu jest rzędu takiego samego, jak liczba wywołań funkcji F, czyli $O(f)$. Stąd złożoność czasowa algorytmu wynosi

$$O(f \log f + n \cdot l + n \cdot m) = O((nl + m \log l) \log(nl + m \log l) + n \cdot m),$$

gdzie pierwszy składnik to sumaryczny czas odwołań do słownika w celu wyznaczenia funkcji F, drugi — $O(n \cdot l)$ — to czas policzenia wyników dla pociągów w początkowym ustawieniu, wreszcie trzeci — $O(n \cdot m)$ — to sumaryczny czas wszystkich wykonywanych porównań etykiet zmienianych pociągów i aktualizacji wyników. Zauważmy, że $n \cdot l = O(f)$, więc drugi składnik nie pojawia się w ostatecznym wzorze na złożoność.

Rozwiązanie wzorcowe o złożoności czasowej $O(f \log f)$

Czy można uzyskać jeszcze szybszy algorytm? Nie, jeśli będziemy nadal trzymać się pomysłu z bezpośrednią symulacją. Wymaga ona $n \cdot m$ operacji, ponieważ w każdym kroku liczba pociągów podobnych do zmienianych może być równa nawet n, a dla wszystkich tych pociągów musimy przynajmniej uaktualnić wynik. Ale jeśli przestaniemy traktować każdy pociąg „indywidualnie" i przyjrzymy się grupom jednakowych pociągów, to pojawia się szansa na przyspieszenie obliczeń.

Załóżmy więc, że wszystkie pociągi występujące w trakcie symulacji podzielimy na *grupy* jednakowych pociągów, które będziemy identyfikować etykietami pociągów-reprezentantów, obliczonymi tak, jak w poprzednim rozdziale. Po przestawieniu dwóch wagonów musimy zmienić przynależność do grupy (co najwyżej) dwóch zmienionych pociągów. Jeśli prześledzimy, do jakich grup należał określony pociąg w trakcie całej symulacji, to wynikiem dla niego będzie oczywiście moc najliczniejszej spośród tych grup. Trzeba więc znaleźć sposób, by śledzić, jak zmieniają się moce grup w czasie, i dodatkowo, a może przede wszystkim, wykonywać na nich operacje dodawania oraz usuwania elementów-pociągów. Zamiast dotychczasowego prostego podejścia do symulacji, odrębnie prześledzimy zmiany zachodzące w grupach i odrębnie przeanalizujemy historię poszczególnych pociągów.

[3] Najbardziej znanymi przykładami drzew zrównoważonych są drzewa AVL, które są opisane w [17], i drzewa czerwono-czarne, o których można przeczytać w [20]. W języku C++ można także skorzystać z gotowej implementacji, a mianowicie z kontenera `set` z biblioteki STL.

[4] Można także zastosować … funkcję haszującą. Jednak tym razem ewentualne „błędy" przełożyłyby się jedynie na dłuższy czas wyszukiwania i/lub wstawiania elementów do słownika. Oczekiwany czas tych operacji to jednak $O(1)$. O zastosowaniu funkcji haszujących do implementacji słownika także można przeczytać więcej w [20].

Pierwszy przebieg symulacji — śledzimy całe grupy. Moc grupy będziemy śledzić, zapisując na specjalnej liście wszystkie wykonane dla niej akcje dodania/usunięcia pociągu. Dla grupy g taką listę oznaczymy przez L_g. Dodanie nowego pociągu w kroku t zapiszemy w postaci pary $(t,1)$, natomiast usunięcie pociągu w kroku t — w postaci pary $(t,-1)$. Na podstawie listy L_g możemy łatwo wyznaczyć moc grupy g w dowolnym momencie t (oznaczmy ją $f(g,t)$) — jest ona równa sumie drugich składowych par $(t_i,x_i) \in L_g$, dla których $t_i \leqslant t$. Aby efektywnie wyznaczać wartości $f(g,t)$, trzeba zadbać, by lista

$$L_g = ((t_1,x_1),(t_2,x_2),(t_3,x_3),\ldots)$$

była posortowana rosnąco według pierwszej składowej, a następnie zastąpić drugie składowe elementów ich sumami częściowymi[5]. W ten sposób otrzymujemy ciąg par

$$M_g = ((t_1,f(g,t_1)),(t_2,f(g,t_2)),\ldots).$$

Ponieważ w trakcie śledzenia historii określonego pociągu podczas symulacji interesuje nas maksymalna moc grupy g w okresie, gdy pociąg ten do niej należał, musimy jeszcze umieć sprawnie odpowiadać na pytanie o maksymalną moc grupy g w zadanym przedziale czasowym. Do tego celu możemy wykorzystać *drzewo przedziałowe*. Jest to zrównoważone, statyczne drzewo binarne, zawierające w liściach pary $(t_i,f(g,t_i))$, a w węzłach wewnętrznych minimum i maksimum z pierwszego elementu par oraz maksimum z drugiego elementu par z poddrzewa tego węzła. Taka implementacja pozwala wyznaczać maksymalną moc grupy w zadanym przedziale czasowym w czasie logarytmicznym ze względu na liczbę operacji dotyczących danej grupy.

Drugi przebieg symulacji — śledzimy pociągi. Jedyne, co pozostaje zrobić, to przeanalizować „historię" każdego pociągu w czasie symulacji. Przyjmijmy, że zarówno pociągi, jak i grupy będziemy oznaczać etykietami obliczonymi w ten sam sposób, co w poprzednim rozdziale. Wówczas wstępne wyznaczenie etykiety pociągu wymaga $O(l)$ wywołań funkcji F, jej aktualizacja po zmianie jednego wagonu — $O(\log l)$ wywołań F, a porównanie dwóch pociągów lub grup — czasu $O(1)$. Na początku odnotowujemy, że każdy pociąg od chwili 0 należy do swojej grupy. Rozważmy teraz przestawienie wagonów pomiędzy pociągami p_1 i p_2 w chwili t_x. Jeśli wiemy, że p_1 przed przestawieniem należał do grupy g_1 i trafił do niej w momencie t_1 (i było to ostatnie „zdarzenie" w historii tego pociągu), to należy wyznaczyć maksymalną wartość $f(g_1,t)$ w okresie $[t_1,t_x]$ i wykorzystać ją do uaktualnienia wyniku dla p_1. Analogicznie postępujemy dla p_2. Następnie odnotowujemy dla p_1 i p_2, że od tej chwili należą do nowych grup.

Postarajmy się przeanalizować złożoność czasową przedstawionego rozwiązania. Wyznaczenie etykiet wszystkich pociągów wymaga czasu $O(f \log f)$. W trakcie pierwszej symulacji stworzymy listy L_g akcji wykonywanych na poszczególnych grupach. Początkowe listy grup (odpowiadające chwili 0) tworzymy w czasie $O(n+f)$. Następnie symulujemy kolejne przestawienia wagonów i przypisujemy kolejne akcje do poszczególnych grup — złożoność

[5] Dla ciągu (a_n) ciąg jego sum częściowych to (s_n), gdzie $s_n = \sum_{i=1}^{n} a_i$.

czasowa tego kroku to $O(m)$. Gotowe listy L_g mają sumaryczną długość $O(n+m)$ — przekształcenie ich w listy z sumami częściowymi, a następnie w drzewa przedziałowe zajmie więc dokładnie tyle samo czasu.

Mając gotowe drzewa przedziałowe dla grup, możemy przystąpić do drugiego przebiegu symulacji. W trakcie tego przebiegu ponownie śledzimy wszystkie zdarzenia, ale tym razem w każdym kroku aktualizujemy wynik i grupę dla zmienionych pociągów. Ze względu na konieczność wykonywania zapytań w drzewach przedziałowych wymaga to czasu $O((n+m)\log(n+m))$ — składnik $n\log(n+m)$ w tym zapisie wziął się stąd, że na końcu symulacji dla każdego pociągu musimy uwzględnić ostatnią grupę, w której się znalazł.

Sumaryczna złożoność całego rozwiązania wynosi więc

$$T(n) \quad = \quad O(f\log f + n + f + m + (n+m)\log(n+m)),$$

czyli

$$\begin{aligned} T(n) &= O((nl + m\log l)\log(nl + m\log l) + n + f + m + (n+m)\log(n+m)) \\ &= O((nl + m\log l)\log(nl + m\log l)) \\ &= O(f\log f). \end{aligned}$$

Widzimy więc, że nowe podejście do symulacji pozwoliło nam zmniejszyć najbardziej kłopotliwy składnik $n \cdot m$ do $(n+m)\log(n+m)$, wskutek czego „ukrył się" w składniku wynikającym z wyliczania etykiet. Rozwiązanie wzorcowe zostało zaimplementowane w plikach `poc0.cpp` i `poc3.java`.

Rozwiązanie alternatywne

Zauważmy, że usprawnienie, które doprowadziło nas do rozwiązania wzorcowego, mogłoby zostać równie dobrze zastosowane do rozwiązania używającego haszowanie — zamiast etykiet pojawiłyby się w nim po prostu funkcje haszujące pociągów. W ten sposób otrzymuje się efektywne, lecz niestety niepewne rozwiązanie — jedynie w przypadku wykorzystania stosunkowo dużej liczby pierwszej P (np. rzędu 10^{16}) istniała duża szansa na zdobycie przez nie maksymalnej punktacji. Sposób poradzenia sobie z tym, że dla tak dużej wartości P liczba możliwych etykiet jest stosunkowo duża, oraz oszacowanie złożoności tego rozwiązania pozostawiamy Czytelnikowi jako ćwiczenie. Pomóc w tym mogą jego implementacje zawarte w plikach `poc.pas`, `poc1.cpp` i `poc2.cpp`.

Testy

Testy do zadania zostały wygenerowane losowo, przy użyciu czterech niezależnych metod. Każda z nich została skonstruowana w celu wychwycenia nieprawidłowych bądź zbyt wolnych rozwiązań różnego rodzaju.

- **s** — generator testów, których celem jest wykrywanie programów o nieefektywnie zaimplementowanej fazie aktualizacji wyników dla pociągów. W testach tego typu występuje wiele identycznych pociągów, przez co rozwiązania nieefektywnie aktualizujące wyniki po każdej zamianie wagonów muszą wykonywać wiele operacji.

- **h** — generator testów, których zadaniem jest wykrycie programów wykorzystujących metodę haszowania. Testy tego typu zawierają wiele podobnie wyglądających pociągów, różniących się kolorem tylko kilku wagonów. Istnieje szansa, że programy wykorzystujące metodę haszowania sklasyfikują podobnie wyglądające pociągi jako takie same (oczywiście bardzo dużo zależy od jakości funkcji haszującej, jakiej zawodnik użył w swoim programie).
- **l** — generator testów losowych. Wybiera on kolejno po dwa pociągi i stara się doprowadzić jeden z nich do postaci drugiego, wykorzystując w tym celu wagony ze specjalnie przygotowanego zbioru pociągów zapasowych.
- **p** — generator testów, których zadaniem jest wykrycie drobnych błędów w programach. W testach tych pociągi różnią się na co najwyżej kilku wagonach i wykonywanych jest bardzo dużo zamian właśnie tych wagonów. Odpowiedzi dla różnych pociągów są stosunkowo zróżnicowane, przez co większość błędnych programów daje na nich złe odpowiedzi.

Poniższa lista zawiera podstawowe informacje dotyczące testów. Przez n została oznaczona liczba pociągów, przez l — ich długość, przez m — liczba wykonywanych zamian, natomiast przez r — rodzaj testu (**s**, **h**, **l** lub **p**).

Nazwa	n	l	m	*r*
poc1 .in	10	10	100	*l*
poc2 .in	10	10	1 000	*l*
poc3a.in	110	10	304	*s*
poc3b.in	100	10	300	*l*
poc3c.in	100	10	300	*p*
poc4a.in	100	10	1 204	*s*
poc4b.in	100	10	3 000	*l*
poc4c.in	100	10	3 000	*p*
poc5a.in	990	100	1 009	*s*
poc5b.in	1 000	100	1 000	*l*
poc5c.in	1 000	100	1 000	*l*
poc5d.in	1 000	100	1 000	*p*
poc6a.in	110	100	30 006	*s*
poc6b.in	100	100	9 996	*h*
poc6c.in	1 000	100	5 000	*l*
poc6d.in	100	100	30 000	*p*

Nazwa	n	l	m	*r*
poc7a.in	995	100	10 004	*s*
poc7b.in	1 000	100	9 996	*h*
poc7c.in	1 000	100	10 000	*l*
poc7d.in	100	100	10 000	*p*
poc8a.in	960	10	83 409	*s*
poc8b.in	920	10	72 996	*h*
poc8c.in	990	10	82 000	*l*
poc8d.in	980	10	90 000	*p*
poc9a.in	990	10	80 009	*s*
poc9b.in	100	90	75 000	*h*
poc9c.in	100	90	83 000	*l*
poc9d.in	100	100	80 000	*p*
poc10a.in	1 000	100	99 988	*s*
poc10b.in	1 000	100	99 996	*h*
poc10c.in	1 000	100	100 000	*l*
poc10d.in	1 000	100	100 000	*p*