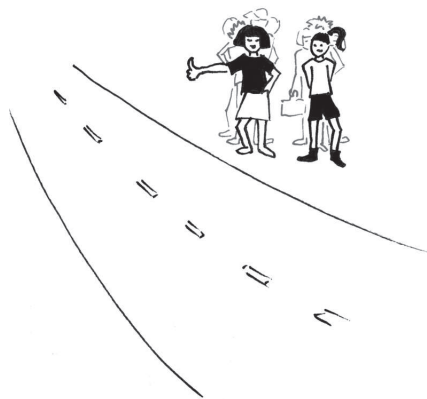


**Słowniki.** Dany jest pewien duży zbiór  $D = \{0, 1, \dots, N - 1\}$ . Chcielibyśmy zaprojektować *słownik* do przechowywania wybranych elementów z  $D$ . Taki słownik to struktura danych, do której można wstawiać elementy, z której można elementy usuwać i dla której można sprawdzać, czy dana liczba  $x \in D$  się w niej znajduje. Implementuje się go zazwyczaj za pomocą zrównoważonych drzew poszukiwań binarnych, których nazwa chyba dobrze oddaje ich poziom skomplikowania. Ale po co się męczyć, jeśli możemy troszkę oszukać?



Do wykonania naszego zadania wykorzystamy *haszowanie*, którego główną ideą jest to, żeby z czegoś dużego zrobić coś małego o „odpowiednich” właściwościach. Obierzmy pewną funkcję  $h$  (zwaną funkcją haszującą, funkcją mieszającą lub funkcją skrótu), która elementy ze zbioru  $D$  przekształca na elementy pewnego niedużego zbioru  $M = \{0, 1, \dots, n - 1\}$  i której wartości można jakoś łatwo obliczać. Zauważmy teraz, że słownik dla elementów z  $D$  możemy zaimplementować za pomocą słownika dla elementów z  $M$ , jeżeli tylko przed wykonaniem każdej operacji słownikowej występujący w niej element z  $D$  będziemy przekształcać za pomocą  $h$  (wynik tego przekształcenia nazywamy *haszem* dla elementu). Z kolei słownik dla elementów z  $M$  implementuje się prościutko, chociażby za pomocą  $n$ -elementowej tablicy zliczającej.

Gdzie zatem tkwi oszustwo? Ponieważ  $n$  jest istotnie mniejsze niż  $N$ , to na pewno funkcja  $h$  nie będzie różnowartościowa, a dowolne dwa elementy  $x \neq y$ , dla których  $h(x) = h(y)$ , w naszym słowniku potraktujemy jako takie same – to niedobrze. Zaradzić można temu na dwa sposoby. Po pierwsze, można założyć, że będziemy mieli szczęście, tzn. że na żadne dwa takie elementy nie natrafimy, i starać się wybrać  $h$  tak, żeby jak najlepiej „rozrzuciła” elementy z  $D$  (do tego celu świetnie nadaje się funkcja  $h(x) = x \bmod n$  i różne jej odmiany). Takie założenie jest jednak zazwyczaj nazbyt optymistyczne, gdyż niezależnie od wyboru  $h$ , średnio po  $O(\sqrt{n})$  operacjach z dużym prawdopodobieństwem zdarzy się jakaś kolizja (jest to tzw. paradoks urodzin, opisany w IKO w *Delcie* 9/2009). Drugi sposób polega na zastosowaniu jakiejś metody radzenia sobie z takimi kolizjami, na przykład przez zastąpienie tablicy zliczającej tablicą list, do których będą trafiały elementy o tych samych wartościach haszy; wówczas znalezienie elementu będzie wymagało zawsze przejścia po liście odpowiadającej jego haszowi. Przy odpowiednim doborze  $h$  można z dużym prawdopodobieństwem zapewnić, aby listy te nadmiernie się nie wydłużały – dokładne oszacowania pozostawiamy teoretykom.

**Słowa.** Niniejszy artykuł nie jest jednak poświęcony liczbom, ale słowom, czyli skończonym ciągom liter z pewnego ustalonego alfabetu  $A$  (np. alfabetu angielskiego), którego rozmiar oznaczamy przez  $|A|$ . Zakładając, że  $D$  jest tym razem zbiorem takich słów, jak w tym przypadku dobrać odpowiednią funkcję haszującą? Dla danego słowa  $s = a_1 a_2 \dots a_m$  przyjmijmy mianowicie:

$$h(s) = (a_1 + a_2 x + a_3 x^2 + \dots + a_m x^{m-1}) \bmod p.$$

W powyższym zapisie  $x$  jest dowolną liczbą całkowitą dodatnią,  $p$  – liczbą pierwszą (tak na wszelki wypadek – zapewne teoretycy mają jakieś dobre argumenty uzasadniające ten wybór), a litery  $a_i$  utożsamiamy z liczbami całkowitymi, tj.  $a = 0, b = 1 \dots$ . Ze względów praktycznych warto dobrać zawsze  $x > 2|A|$ , gdyż dla bardzo małych  $x$  (np.  $x = 2$ ) zazwyczaj łatwo znaleźć dwa krótkie słowa tej samej długości i o tych samych wartościach haszy. (Czy potrafisz, Czytelniku, podać jakiś taki przykład?)

Aby wyznaczyć hasz dla danego słowa, możemy zastosować tzw. schemat Hornera, czyli obliczyć hasze dla wszystkich kolejnych sufiksów (tj. końcowych fragmentów) słowa  $s$ :

$$h_m = a_m \bmod p, \quad h_i = (a_i + x h_{i+1}) \bmod p \quad \text{dla } i = m - 1, m - 2, \dots, 1.$$

Daje to prosty algorytm o złożoności  $O(m)$ .

Za pomocą opisanej funkcji  $h$  możemy już łatwo skonstruować słownik reprezentujący zbiór słów. Jednak nie jest to bynajmniej jedyne zastosowanie haszy na słowach. Okazuje się, że za ich pomocą można rozwiązać wiele znanych



**Rozwiązanie zadania F 751.**

W chwili, gdy natężenie prądu wynosiło  $I_0$ , ładunek na kondensatorze  $C$  wynosił  $q_0 = CI_0 R$ , a energia pola elektrostatycznego była równa

$$W_0 = \frac{q_0^2}{2C}.$$

Stan równowagi nastąpi, gdy ten ładunek rozdzieli się między kondensatorami. Energia pola elektrostatycznego będzie wtedy równa

$$W_1 = \frac{q_0^2}{2C_{\text{całk}}} = \frac{q_0^2}{6C}.$$

Zatem w układzie wydzielono się ciepło

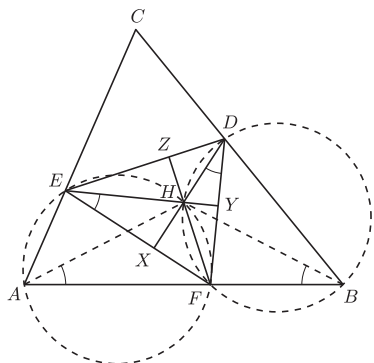
$$Q = W_0 - W_1 = \frac{q_0^2}{3C} = \frac{CI_0^2 R^2}{3}.$$



### Rozwiązanie zadania M 1259.

Z danych w treści zadania równości kątów wynika, że  $\sphericalangle EFD = \sphericalangle BCA$ .

Oznaczmy przez  $H$  punkt przecięcia wysokości trójkąta  $DEF$ .



Przyjmijmy ponadto, że proste  $DH$  i  $EF$  przecinają się w punkcie  $X$ , proste  $EH$  i  $DF$  przecinają się w punkcie  $Y$ , a proste  $FH$  i  $ED$  przecinają się w punkcie  $Z$ . Wówczas

$$\sphericalangle EAF + \sphericalangle EHF = \sphericalangle EDF + \sphericalangle YHZ = 360^\circ - \sphericalangle HYD - \sphericalangle HZD = 180^\circ,$$

skąd wynika, że na czworokącie  $AFHE$  można opisać okrąg. Analogicznie, na czworokącie  $BFHD$  można opisać okrąg. Ponadto

$$\sphericalangle HEF = 90^\circ - \sphericalangle DFE = \sphericalangle HDF.$$

Wobec tego

$$\sphericalangle HAF = \sphericalangle HEF = \sphericalangle HDF = \sphericalangle HBF,$$

skąd wynika, że  $HA = HB$ .

Analogicznie dowodzimy, że  $HB = HC$ , co oznacza, że punkt  $H$  jest środkiem okręgu opisanego na trójkącie  $ABC$ .



problemów z algorytmiki tekstów, zazwyczaj w podobnej złożoności czasowej co w przypadku innych klasycznych metod (takich jak słownik słów bazowych czy dużo bardziej złożone drzewo sufiksowe), ale znacznie prościej.

Jak poprzednio, i tutaj tkwi drobne oszustwo wynikające z niebezpieczeństwa kolizji. Jednakże, ponieważ w dalszych rozważaniach nie będziemy się zajmować strukturami słownikowymi, więc możemy sobie pozwolić na to, żeby  $p$  było stosunkowo duże, np. rzędu  $10^{16}$  (zakładając, że chcemy uniknąć przepełnienia typu 64-bitowego w schemacie Hornera). Można też w ogóle nie przyjmować żadnej wartości  $p$ , czyli wyznaczać wartości haszy z pominięciem operacji modulo! Wówczas będzie to odpowiadać obliczeniom modulo zakres typu, czyli np.  $2^{64}$ . W obu przypadkach prawdopodobieństwo kolizji, przy odpowiednio niedużej liczbie operacji, jest stosunkowo małe, więc można przyjąć wariant optymistyczny, że równe hasze oznaczają równe słowa.

**Najdłuższe wspólne pod słowo.** Zauważmy, że na podstawie haszy  $h_i$  dla sufiksów słowa  $s$  możemy łatwo wyznaczać hasze dla poszczególnych pod słów (tj. spójnych fragmentów)  $s$ . Faktycznie, dla słowa  $a_i a_{i+1} \dots a_j$  szukanym haszem jest

$$(a_i + a_{i+1}x + \dots + a_j x^{j-i}) \bmod p = (h_i - x^{j-i+1} h_{j+1}) \bmod p.$$

Jeżeli spamiętamy wartości pierwszych  $m$  potęg liczby  $x$  modulo  $p$ , to wartość tę możemy obliczać w czasie stałym. W takim razie także w  $O(1)$  można sprawdzać, czy dane dwa (równej długości) pod słowa  $s$  są takie same.

Dzięki temu za pomocą haszy można zaimplementować algorytm wyznaczania najdłuższego wspólnego pod słowa (NWP) dwóch słów  $s$  i  $t$  długości  $O(n)$ . Będziemy w nim wyszukiwać binarnie długość  $w$  wyniku (daje to czynnik  $\log m$  w złożoności czasowej), a przy ustalonym  $w$  – wrzucać hasze wszystkich  $w$ -literowych pod słów  $s$  i  $t$  do jednej tablicy, sortować w porządku niemalejącym (koszt czasowy  $O(m \log m)$ ) i sprawdzać, czy jakieś dwa hasze pochodzące z różnych wyjściowych słów są takie same. Ostatecznie złożoność czasowa uzyskanego algorytmu to  $O(m \log^2 m)$ . Czy potrafisz, Czytelniku, uogólnić ten algorytm do wyznaczania NWP kilku (ale wciąż  $O(1)$ ) słów?

**Tablica sufiksowa.** Stwierdziliśmy już, że można wykorzystać haszowanie do sprawdzania równości dwóch słów, ale wydawać by się mogło, że w przypadku, gdy słowa mają różne hasze, nie sposób stwierdzić, które z nich jest wcześniejsze leksykograficznie (tj. alfabetycznie). Otóż nic bardziej mylnego! Wystarczy za pomocą porównywania haszy wyszukać binarnie najdłuższy wspólny prefiks (tj. początkowy fragment) dwóch słów i porównać stojące za nim w tych słowach literki – lub stwierdzić ich brak, jeżeli jedno z wyjściowych słów jest prefiksem drugiego. Koszt całej operacji to zaledwie  $O(\log m)$ .

W ten sposób można wyznaczyć bardzo ważną strukturę danych w algorytmice tekstów, czyli tablicę sufiksową. W tablicy tej wymienione są numery wszystkich sufiksów danego słowa w kolejności leksykograficznej tych sufiksów – np. dla słowa  $abaab$  kolejność ta to:  $aab, ab, abaab, b, baab$ , czyli tablica sufiksowa to  $3, 4, 1, 5, 2$ . W tym celu wystarczy posortować (czynnik  $m \log m$  w złożoności) wszystkie numery sufiksów słowa, jako operator porównania wykorzystując opisane powyżej porównanie leksykograficzne – daje to łącznie koszt czasowy  $O(m \log^2 m)$ .

**Wyszukiwanie wzorca w tekście.** Znany algorytm Karpa–Rabina (KR) wyszukiwania wzorca w tekście polega na porównywaniu haszy kolejnych pod słów tekstu z haszem wzorca (złożoność czasowa  $O(m)$ ). Nie jest to jednak wielkie osiągnięcie, gdyż znanych jest mnóstwo prostych i efektywnych algorytmów wyszukiwania wzorca. Jednakże większość z nich bardzo trudno uogólnia się na przypadek dwuwymiarowy – wyszukiwanie jednej prostokątnej tablicy znaków w drugiej. Problem ten nie występuje w przypadku algorytmu KR, dla którego takie uogólnienie jest stosunkowo proste. Pozostawiamy je Tobie, Drogi Czytelniku, jako ćwiczenie. Drobną podpowiedź: w algorytmie KR hasz dla kolejnego pod słowa tekstu wyznaczany jest bezpośrednio na podstawie hasza poprzedniego pod słowa, a nie na podstawie wartości  $h_i$ .