Marek Biskup
Treść zadania, Opracowanie

Piotr Niedźwiedź
Program

Dostępna pamięć: 64 MB.

OI, Etap III, dzień drugi, 02.04.2009

Kod

Bajtocki Instytut Telekomunikacyjny (BIT) zajmuje się ustalaniem standardów przesyłania danych w sieciach telekomunikacyjnych Bajtocji. Bajtazar, jeden z informatyków pracujących w BIT, zajmuje się **kodami prefiksowymi** *— specjalnym sposobem reprezentowania znaków. Każdemu znakowi bajtockiego alfabetu odpowiada pewien ciąg bitów, nazywany kodem tego znaku. Kody znaków mają następujące własności:*

- *Kod żadnego znaku nie jest prefiksem (tj. początkowym fragmentem) kodu żadnego innego znaku. Na przykład, jeśli* `010010` *jest kodem litery* `A`*, to żaden z ciągów bitów:* `0`*,* `01`*,* `010`*,* `0100` *ani* `01001` *nie może być kodem żadnego znaku. Podobnie,* `0100100`*,* `0100101` *oraz dłuższe ciągi zaczynające się od* `010010` *nie mogą być kodami znaków.*
- *Jeśli dany ciąg bitów* w *jest prefiksem kodu pewnego znaku, ale nie jest całym kodem, to ciągi bitów postaci* w`0` *i* w`1` *(czyli* w *z dopisanym na końcu zerem lub jedynką) są prefiksami kodów (lub całymi kodami) pewnych znaków. Na przykład, jeśli* `0100` *jest prefiksem kodu litery* `A`*, to* `01000` *oraz* `01001` *muszą być prefiksami kodów (lub kodami) pewnych znaków.*

Rozważmy następujący, przykładowy kod prefiksowy dla alfabetu złożonego ze znaków `A`*,* `B`*,* `C`*,* `D` *i* `E`*:*

znak	kod znaku
A	00
B	10
C	11
D	010
E	011

Kodowanie ciągu znaków za pomocą kodu prefiksowego polega na połączeniu kodów jego kolejnych znaków. Na przykład zakodowany ciąg `BACAEBABAE` *ma postać* `10001100011100010000011`.

Bajtazar zauważył, że jeśli pewna liczba początkowych bitów zostanie utracona, to zakodowana informacja może być odkodowana niepoprawnie albo wcale. Na przykład, jeśli usuniemy pięć pierwszych bitów z ciągu podanego powyżej, to powstały ciąg `10001110001000011` *zostanie odkodowany jako* `BACBABAE`*. Pięć ostatnich liter (*`BABAE`*) jest poprawnych, jednak trzy pierwsze (*`BAC`*) nie. Bajtazar zauważył, że wszystkie litery po pierwszym* `E` *zostały odkodowane poprawnie. Doszedł do wniosku, że zawsze, jeśli żadne bity kodu* `E` *nie zostaną utracone, to wszystkie kolejne znaki za* `E` *zostaną odkodowane poprawnie. Tak będzie dla dowolnego kodowanego ciągu znaków, w którym występuje* `E`*. Zauważył on, że litera* `D` *też ma tę własność, ale litery* `A`*,* `B` *i* `C` *tej własności nie mają.*

Opisaną własność kodów znaków `E` *i* `D` *Bajtazar określił jako bycie* **kodem synchronizującym**. *Bajtazar powierzył Ci zadanie napisania programu znajdującego, dla danego kodu prefiksowego, wszystkich kodów synchronizujących. Aby zaoszczędzić czas, wymyślił sobie, że pokaże Ci wszystkie kody znaków na swoim binarnym monitorze. Monitor ma cztery przyciski:*

- `0` — *dopisz 0*
- `1` — *dopisz 1*
- `B` — *backspace, usuwa ostatnio dopisaną cyfrę*
- `X` — *po naciśnięciu tego przycisku monitor wydaje charakterystyczny dźwięk „beep”.*

Na początku wyświetlacz jest pusty; Bajtazar kolejno naciska przyciski i gdy na monitorze pojawia się kod kolejnego znaku, Bajtazar naciska przycisk `X`*. Po pokazaniu ostatniego znaku Bajtazar czyści ekran (naciskając odpowiednią liczbę razy przycisk* `B`*). Wiesz, że Bajtazar pokaże Ci cały kod prefiksowy, naciskając najmniejszą możliwą dla tego kodu liczbę przycisków.*

Wejście

W pierwszym wierszu standardowego wejścia znajduje się jedna liczba całkowita n *(*$6 \leqslant n \leqslant 3\,000\,000$*) oznaczająca liczbę przycisków naciśniętych przez Bajtazara. W kolejnym wierszu znajduje się* n*-literowy napis złożony ze znaków '*`0`*', '*`1`*', '*`B`*' oraz '*`X`*', oznaczających poszczególne przyciski. Każde naciśnięcie przycisku* `X` *oznacza kolejny znak (znaki numerujemy od 1). Suma długości wszystkich kodów nie przekroczy* 10^8*.*

Wyjście

W pierwszym wierszu standardowego wyjścia należy wypisać liczbę k *— liczbę kodów synchronizujących. W kolejnych* k *wierszach należy wypisać, w porządku rosnącym, numery znaków, które są kodami synchronizującymi, po jednym w wierszu. Jeżeli dla danego kodu prefiksowego nie ma żadnych kodów synchronizujących, to należy wypisać tylko jeden wiersz zawierający liczbę 0.*

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`21`	`2`
`11XB0XBB00XB11XB0XBBB`	`4`
	`5`

Wyjaśnienie przykładu

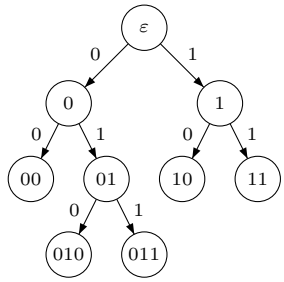
Oto kolejne kody, które pojawiają się na monitorze Bajtazara: `11`, `10`, `00`, `011`, `010`*. Kody* `011` *i* `010` *są kodami synchronizującymi.*

Rozwiązanie

Wczytanie kodu i jego reprezentacja

Na początku zajmiemy się wczytywaniem kodu do pamięci komputera. Jak się za chwilę przekonamy, sposób pokazywania kodu przez Bajtazara jest prosty i wręcz ułatwia jego wczytywanie.

Naturalnym sposobem reprezentacji kodu prefiksowego jest *właściwe* drzewo binarne, czyli takie, w którym każdy wierzchołek w drzewie ma dwóch synów lub jest liściem. Krawędzie do lewego syna są etykietowane wartością `0`, a do prawego syna — wartością `1`. Każdemu wierzchołkowi w takim drzewie odpowiada ciąg binarny złożony z etykiet na drodze od korzenia do tego wierzchołka. Ciąg ten będzie *etykietą* wierzchołka.



Rys. 1: Właściwe drzewo binarne.

Przykładowe drzewo binarne przedstawione jest na rysunku 1. Etykiety liści odpowiadają kodom znaków (czyli tzw. słowom kodowym) z przykładu z treści zadania. Etykiety wierzchołków wewnętrznych odpowiadają prefiksom właściwym (tzn. początkowym fragmentom krótszym niż całe słowo) słów kodowych. Etykieta korzenia to słowo puste, które oznaczone jest przez ε. Okazuje się, że takie drzewo można stworzyć dla każdego kodu prefiksowego, co wynika z dwóch własności kodów prefiksowych opisanych w treści zadania:

- Własność pierwsza mówi, że kod żadnego znaku nie jest prefiksem kodu innego znaku. W reprezentacji drzewowej znaczy to tyle, że wierzchołek wewnętrzny nie może być słowem kodowym.
- Własność druga mówi, że jeśli w jest właściwym prefiksem słowa kodowego to $w\texttt{0}$ oraz $w\texttt{1}$ są prefiksami (właściwymi lub nie) słowa kodowego. Dla drzewa oznacza to, że wierzchołek wewnętrzny ma zawsze dwóch synów, czyli że drzewo jest właściwe.

W tym miejscu warto przytoczyć jedną ważną własność właściwych drzew binarnych, która będzie nam potrzebna w dalszej części opisu, a której prosty dowód (np. przez indukcję względem rozmiaru drzewa) pozostawiamy Czytelnikowi:

Obserwacja 1. Właściwe drzewo binarne o ℓ liściach ma $2\ell - 1$ wierzchołków.

Reprezentacja danych wejściowych umożliwia łatwą konstrukcję drzewa. W tym celu potrzebny nam będzie licznik m przechowujący liczbę odczytanych już słów kodowych. Zaczynamy z pojedynczym wierzchołkiem — korzeniem drzewa — i inicjalizujemy licznik na

0. Następnie czytamy kolejne znaki z wejścia i konstruujemy drzewo, wykonując operacje zależne od przeczytanego znaku.

`0`: Tworzymy nowy wierzchołek, łączymy go z aktualnym wierzchołkiem krawędzią o etykiecie `0` (lewa krawędź) i przechodzimy do nowego wierzchołka.

`1`: Tworzymy nowy wierzchołek, łączymy go z aktualnym wierzchołkiem krawędzią o etykiecie `1` (prawa krawędź) i przechodzimy do nowego wierzchołka.

`B`: Przechodzimy do ojca aktualnego wierzchołka.

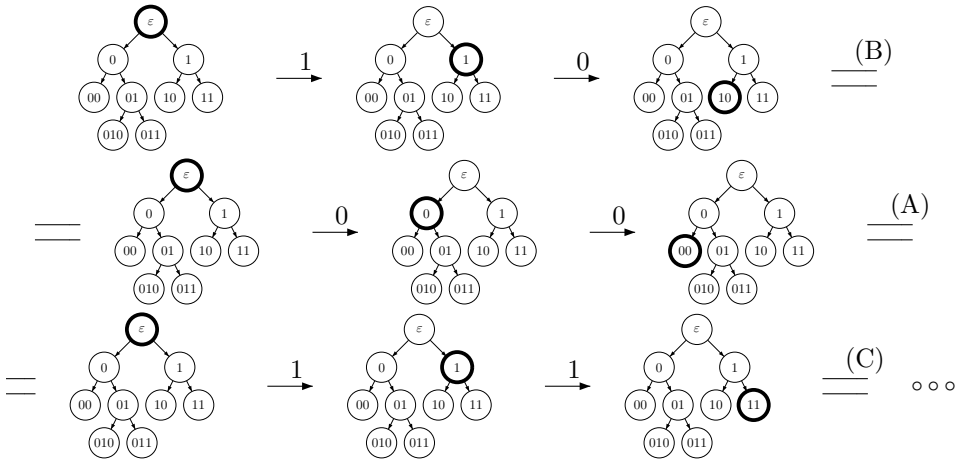
`X`: Zwiększamy m o jeden i zapisujemy nową wartość m w aktualnym wierzchołku.

W każdym momencie etykieta aktualnego wierzchołka odpowiada ciągowi aktualnie wyświetlanemu na monitorze. Algorytm ten można zrealizować rekurencyjnie:

```
1:  m := 0;
2:  procedure KONSTRUUJ(q)
3:  begin
4:    while true do
5:      c := KOLEJNYZNAKNAWEJŚCIU();
6:      if c = 0 then
7:        q.left := NOWYWIERZCHOŁEK();
8:        KONSTRUUJ(q.left);
9:      else if c = 1 then
10:       q.right := NOWYWIERZCHOŁEK();
11:       KONSTRUUJ(q.right);
12:     else if c = X then
13:       m := m + 1;
14:       q.number := m;
15:     else { (c = B) lub koniec wejścia }
16:       return;
17: end
18: { Wczytanie drzewa: }
19: Pomiń pierwszy wiersz wejścia;
20: r := NOWYWIERZCHOŁEK();
21: KONSTRUUJ(r);
22: return r;
```

Pierwszy wiersz jest pomijany, gdyż informacja o liczbie znaków w opisie drzewa nie jest nam potrzebna. Z warunku, że Bajtazar czyści ekran, wynika, że program wróci z wywołań rekurencyjnych funkcji KONSTRUUJ po przeczytaniu literek `B`. Powrót z głównego wywołania funkcji KONSTRUUJ będzie rezultatem braku kolejnych znaków na wejściu.

Drzewa binarnego można łatwo użyć do dekodowania zakodowanego ciągu. Na starcie „umieszczamy” dekoder w korzeniu drzewa. Następnie czytamy kolejne bity zakodowanego ciągu, schodząc w dół drzewa po krawędziach odpowiadających przeczytanym bitom. Po dotarciu do liścia wypisujemy literę odpowiadającą etykiecie tego liścia i wracamy do korzenia.



Rys. 2: Dekodowanie ciągu bitów. Etykiety nad strzałkami oznaczają kolejne czytane bity, a symbole w nawiasach — odkodowane znaki.

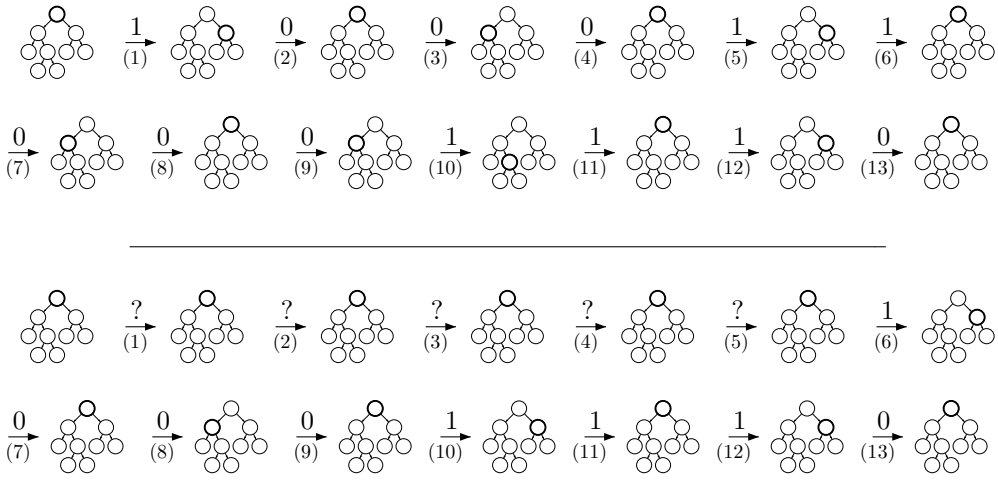
Dla zakodowanego ciągu z treści zadania — `10001100011110001000011` — dekodowanie przebiega następująco (rysunek 2):

1. Startujemy w korzeniu: $q = \varepsilon$ (q oznaczać będzie aktualną pozycję dekodera, czyli jego *stan*, a ε to korzeń drzewa).
2. Czytamy `1` i schodzimy w prawo ($q = $ `1`).
3. Czytamy `0` i schodzimy w lewo ($q = $ `10`). Dotarliśmy do liścia `10`, więc odkodowujemy literę `B`. Następnie przenosimy dekoder do korzenia ($q = \varepsilon$) i kontynuujemy.
4. Czytamy `0` i schodzimy w lewo ($q = $ `0`).
5. Czytamy `0` i schodzimy w lewo ($q = $ `00`). Odkodowujemy `A` i kontynuujemy z korzenia.
6. ...

Rozważmy dekoder umieszczony w pewnym wierzchołku q_1. Po przeczytaniu słowa w trafi on do pewnego wierzchołka q_2. Zakładamy przy tym, że q_2 nie jest liściem, bo — jak pamiętamy — dekoder z liścia trafia natychmiastowo do korzenia (czyli przyjmujemy wtedy $q_2 = \varepsilon$). W tym przypadku powiemy, że dekoder *przechodzi* z q_1 do q_2 po słowie w. Oznaczmy takie przejście $\delta(q_1, w) = q_2$. Zauważmy, że δ jest funkcją ze zbioru par (wierzchołek wewnętrzny, słowo) w zbiór wierzchołków wewnętrznych. Dla każdego słowa kodowego w mamy $\delta(\varepsilon, w) = \varepsilon$, tzn. słowa kodowe zawsze prowadzą z korzenia do korzenia.

Odkodowanie ciągu z pominięciem prefiksu

Jak zachowa się dekoder, jeśli pewien początkowy fragment zakodowanej wiadomości zostanie utracony? Aby odpowiedzieć na to pytanie, porównajmy pracę dekodera na pełnych



Rys. 3: Przejścia dla dekodera zaczynającego od pierwszego bitu (na górze) i od szóstego bitu (na dole). Na strzałkach zaznaczona jest wartość aktualnie czytanego bitu (nad strzałką) oraz jego numer (pod strzałką). Pierwsze pięć bitów dolnego dekodera jest oznaczone znakiem zapytania, bo dekoder te bity pomija. Z rysunku wynika, że synchronizacja dolnego dekodera następuje po przeczytaniu jedenastego bitu.

danych (oznaczany jako dekoder D_0) oraz na danych bez pięciu początkowych bitów (dekoder D_5), używając przykładu z treści zadania.

Rysunek 3 pokazuje działanie dekodera D_0 (na górze) oraz D_5 (na dole). Dekodery w liściach zostały na tym rysunku pominięte, gdyż trafiają one natychmiast do korzenia drzewa (porównaj z rysunkiem 2). Pominięte zostały też odkodowywane znaki. Dekoder D_5 opuszcza pięć początkowych bitów, więc bity te zostały na rysunku oznaczone znakiem zapytania.

Widzimy, że po bitach 5-10 stany dekoderów są różne. Po jedenastym bicie stany dekoderów stają się identyczne. Dekodery odkodowały jednak w tym momencie różne litery, ponieważ ich poprzednie stany były różne. Następnie dekodery czytają taki sam zakodowany ciąg. Biorąc pod uwagę, że stan dekoderów po 11. bicie jest taki sam oraz że dekodery podlegają takim samym regułom przejścia, wnioskujemy, że ich kolejne stany muszą być identyczne. Odkodują one zatem identyczne kolejne litery.

W treści zadania podano, że synchronizacja wiąże się w tym przypadku z wystąpieniem litery `E`. W dodatku każde wystąpienie litery `E` synchronizuje dekoder, niezależnie od tego, jaka wiadomość jest przesyłana oraz jaki prefiks tej wiadomości został pominięty przy dekodowaniu. Synchronizacja polega na tym, że po odczytaniu kodu `E` stan dekodera D_k, czyli dekodera pomijającego pierwszych k bitów, staje się identyczny ze stanem dekodera D_0, zaczynającego wiadomość od początku. Tuż po odczytaniu `E` dekoder D_0 będzie w stanie ε, gdyż po przeczytaniu pełnego słowa kodowego dekoder zawsze trafia do tego stanu (porównaj z rysunkiem 2). Dekoder D_k musi więc po przeczytaniu `E` trafić także do stanu ε.

Popatrzmy, w jakim stanie może znaleźć się dekoder D_k *tuż przed* przeczytaniem litery `E`. Fragment poprzedzający literę `E` może być dowolnym ciągiem słów kodowych. (Nie znaczy to, że jest to dowolny ciąg bitów, np. `000` nie jest ciągiem słów kodowych!) Dowolny prefiks (początkowy fragment) tego ciągu może zostać utracony. Dekoder zawsze zaczyna w stanie

ε. Zatem dekoder tuż przed przeczytaniem `E` może być w dowolnym stanie, do którego da się przejść z korzenia po sufiksie (końcowym fragmencie) ciągu słów kodowych. Na potrzeby tego zadania nazwijmy zbiór takich wierzchołków zbiorem wierzchołków *ważnych*. Reasumując:

Definicja 1. Zbiór wierzchołków *ważnych* dla drzewa kodu prefiksowego to zbiór wierzchołków wewnętrznych drzewa, do których da się dojść z korzenia po pewnym sufiksie pewnego ciągu słów kodowych.

Nasze zadanie polega na znalezieniu kodów synchronizujących, czyli takich, że dekoder po napotkaniu takiego kodu zawsze dobrze odczyta kolejne znaki. W świetle analizy przedstawionej powyżej, równoważnym sformułowaniem zadania jest:

Znaleźć wszystkie słowa kodowe, synchronizujące każdy ważny wierzchołek.

Synchronizacja oznacza sprowadzenie wierzchołka do ε, bo taki jest stan dekodera D_0 po zakończeniu czytania pełnego słowa kodowego. Rozwiązanie w naturalny sposób dzieli się na dwie części:

1. znalezienie zbioru wierzchołków ważnych,
2. znalezienie słów kodowych synchronizujących wszystkie wierzchołki ważne.

Zadanie wyszukania wierzchołków ważnych da się również rozbić następująco:

1. wyznaczenie wierzchołków wewnętrznych, do których da się dojść z korzenia po sufiksach pojedynczych słów kodowych (nazwijmy je wierzchołkami *bardzo ważnymi*),
2. wyznaczenie wierzchołków wewnętrznych, do których da się dojść z wierzchołków bardzo ważnych po ciągach słów kodowych.

Aby nigdzie nam nie zaginęła w tekście, poniżej ponawiamy dokładną definicję wierzchołków bardzo ważnych:

Definicja 2. Zbiór wierzchołków *bardzo ważnych* dla drzewa kodu prefiksowego to zbiór wierzchołków wewnętrznych drzewa, do których da się dojść z korzenia po pewnym sufiksie pewnego słowa kodowego.

Na rysunku 1 wierzchołki bardzo ważne to ε, `0` oraz `1`. Rzeczywiście, etykiety tych wierzchołków to sufiksy słów kodowych. Są to również jedyne wierzchołki ważne w tym drzewie, bo startując z dowolnego wierzchołka bardzo ważnego i przechodząc po dowolnym słowie kodowym, trafiamy z powrotem do jednego z wierzchołków bardzo ważnych.

Poszczególne etapy rozwiązania są przedstawione w kolejnych podrozdziałach.

Wyznaczenie zbioru wierzchołków bardzo ważnych

Podamy kilka algorytmów dla problemu wyszukiwania wierzchołków bardzo ważnych. Najprostszy z nich jest naturalny, choć nieefektywny. Drugi algorytm stanowi jego optymalizację. Trzeci algorytm jest najefektywniejszy, jednak działa na innej zasadzie oraz nie poprawia całkowitej złożoności rozwiązania.

Naiwny algorytm dla zbioru wierzchołków bardzo ważnych

Naiwny algorytm znajdowania wierzchołków bardzo ważnych bazuje na definicji tych wierzchołków. Wierzchołek q jest bardzo ważny, jeśli $q = \delta(\varepsilon, u)$ dla pewnego sufiksu u słowa kodowego. W naiwnym algorytmie sprawdzamy wszystkie słowa kodowe po kolei. Dla każdego słowa w rozważamy wszystkie jego sufiksy u i dodajemy do zbioru wierzchołków bardzo ważnych wierzchołek docelowy przejścia z korzenia po rozważanym sufiksie, czyli $\delta(\varepsilon, u)$.

Przyjmijmy następujące oznaczenia:

- K — zbiór wszystkich słów kodowych rozważanego kodu,
- suffixes(w) — zbiór wszystkich sufiksów słowa w; zbiór ten zawiera w szczególności słowo puste i w,
- N — liczba słów kodowych w kodzie,
- $w_1, w_2, \ldots, w_N$ — kolejne słowa kodowe kodu.
- $L = |w_1| + |w_2| + \ldots + |w_N|$ — suma długości wszystkich słów kodowych kodu.

Oto pseudokod naiwnego algorytmu dla zbioru wierzchołków bardzo ważnych:

```
1: W := ∅;
2: forall w in K do
3:    forall u in suffixes(w) do
4:       W := W ∪ {δ(ε, u)};
5: return W;
```

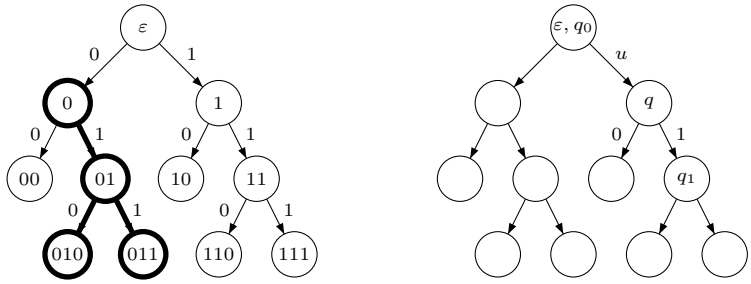
Dla każdego słowa w wykonujemy $|w|+1$ przejść po sufiksie tego słowa. Sufiksy te mają długość $0, 1, 2, \ldots, |w|$, co daje w sumie $O(|w|^2)$ operacji. Złożoność algorytmu naiwnego wynosi więc $O(|w_1|^2 + |w_2|^2 + \ldots + |w_N|^2)$. W pesymistycznym przypadku, jeśli słowa kodowe mają długość rzędu N, czyli dla drzewa niezrównoważonego, złożoność algorytmu to $O(N^3)$. Z kolei gdy drzewo kodu jest zrównoważone, czyli gdy słowa kodowe mają długości $O(\log N)$, złożoność algorytmu wyniesie $O(N \log^2 N)$. Jakkolwiek w optymistycznym przypadku złożoność jest niezła, przypadek pesymistyczny jest wysoce niezadowalający.

Poprawiony algorytm dla zbioru wierzchołków bardzo ważnych

Optymalizacja naiwnego algorytmu będzie opierała się na spostrzeżeniu, że wiele sufiksów słów kodowych ma wspólne fragmenty. Jeśli słowa kodowe różnią się tylko na ostatnim bicie, np. `1010100` oraz `1010101`, to odpowiadające sobie sufiksy tych słów też różnią się tylko na ostatnim bicie. Zamiast więc wykonywać obliczenia dla obu sufiksów oddzielnie, możemy większość obliczeń wykonać wspólnie.

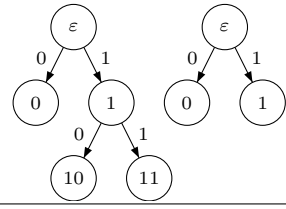
Rozważmy u_0 i u_1 — sufiksy słów kodowych różniące się tylko na ostatnim bicie. Możemy napisać $u_0 = u\mathtt{0}$ oraz $u_1 = u\mathtt{1}$, gdzie u jest wspólną częścią słów. W algorytmie naiwnym obliczaliśmy oddzielnie $q_0 = \delta(\varepsilon, u_0)$ oraz $q_1 = \delta(\varepsilon, u_1)$ i dodawaliśmy oba wierzchołki do zbioru wierzchołków bardzo ważnych. Wartość $\delta(\varepsilon, u_0)$, czyli wierzchołek

docelowy przejścia od korzenia po słowie u_0, możemy znaleźć w dwóch krokach. Najpierw obliczymy przejście po słowie u: $q = \delta(\varepsilon, u)$, a następnie z wierzchołka q przejdziemy po bicie 0: $q_0 = \delta(q, 0)$. Podobnie wyznaczymy $\delta(\varepsilon, u_1)$, jednak w tym przypadku nie ma potrzeby obliczania q ponownie — wystarczy obliczyć $q_1 = \delta(q, 1)$. Przykład opisanych w tym akapicie obliczeń jest przedstawiony na rysunku 4. (Uwaga, rysunek przedstawia inne drzewo niż w treści zadania.)



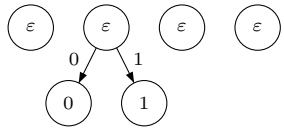
Rys. 4: Przykład wykorzystania wspólnych prefiksów słów do przyspieszenia naiwnego algorytmu znajdowania wierzchołków bardzo ważnych. Rozważamy sufiksy długości 2 kodów 010 i 011, czyli 10 i 11. Sufiksy te są wyznaczone przez etykiety pogrubionych krawędzi na rysunku z lewej strony. Przejścia z korzenia po tych sufiksach ilustruje rysunek z prawej. Wspólną część sufiksów stanowi słowo $u = 1$ i po tym słowie trafiamy do wierzchołka q. Następnie po bitach 0 i 1 trafiamy z q do q_0 i q_1. Zauważmy, że q_0 jest korzeniem, gdyż z wierzchołka 10 przechodzimy bezpośrednio do ε.

Popatrzmy teraz na wszystkie sufiksy słów kodowych powstałe przez obcięcie pierwszego bitu. Sufiksy te są opisane przez drzewa binarne, które są zaczepione w lewym i prawym synu korzenia drzewa kodu. Dla kodu z rysunku 1, poddrzewa odpowiadające tym sufiksom przedstawione są na rysunku 5. Zauważmy, że etykiety w węzłach drzewa zostały poprawione (względem etykiet z rysunku 1), aby odzwierciedlały etykiety na ścieżce od nowego korzenia do węzła.



Rys. 5: Poddrzewa drzewa z rysunku 1 odpowiadające sufiksom słów kodowych powstałych po obcięciu pierwszego bitu.

Tak samo sufiksy powstałe przez obcięcie pierwszych dwóch bitów można opisać za pomocą poddrzew zaczepionych w wierzchołkach w odległości 2 od korzenia (rysunek 6). W ogólności, sufiksy powstałe przez obcięcie pierwszych k bitów można opisać przez poddrzewa zaczepione w wierzchołkach w odległości k od korzenia. Zatem zbiór wszystkich sufiksów słów kodowych jest opisany przez wszystkie poddrzewa zaczepione w wierzchołkach drzewa kodu.



Rys. 6: Poddrzewa drzewa z rysunku 1 odpowiadające sufiksom słów kodowych powstałych po obcięciu pierwszych dwóch bitów.

W nowym algorytmie skorzystamy z tego, że sufiksy tworzą drzewa binarne, aby wykonywać operacje na wielu sufiksach jednocześnie. Dla każdego drzewa sufiksów będziemy obliczać funkcję $\delta(\varepsilon, u')$ dla u' odpowiadających etykietom *wszystkich* wierzchołków tego drzewa, także wierzchołków wewnętrznych (docelowo interesuje nas wartość $\delta(\varepsilon, u)$ dla u będących sufiksami słów kodowych, czyli odpowiadających liściom drzewa sufiksów). Dzięki temu, mając wartość $\delta(\varepsilon, u')$, będziemy mogli w jednym kroku obliczyć wartość $\delta(\varepsilon, x)$ dla x będącego synem wierzchołka u'. W ten sposób każde poddrzewo sufiksów zostanie przetworzone w czasie proporcjonalnym do jego rozmiaru.

Na procedurę przetwarzania drzewa sufiksów możemy spojrzeć jak na synchroniczne przechodzenie po dwóch drzewach. W każdym momencie trzymamy po jednym wierzchołku z każdego drzewa — drzewa kodu (x) i poddrzewa sufiksów (y). Przy przechodzeniu po etykiecie `0`, każdy wierzchołek przechodzi na swojego lewego syna (dla etykiety `1` — na prawego). Wierzchołek x trzyma aktualną wartość funkcji δ, więc jeśli dojdziemy z nim do liścia, musimy przejść do korzenia. Wierzchołek y trzyma aktualny wierzchołek w drzewie sufiksów. Naszym celem jest przetworzenie wszystkich sufiksów z tego drzewa, więc liście tego drzewa ograniczają głębokość przechodzenia.

Pseudokod przetwarzania drzewa sufiksów (pojedynczego!) w celu uzupełnienia zbioru wierzchołków bardzo ważnych B wygląda następująco:

```
1: procedure PRZESZUKIWANIESYNCHRONICZNE1A(x, y)
2: begin
3:   if x jest liściem then
4:     x := ε;
5:   if y jest liściem then
6:     B := B ∪ {x};
7:   else
8:     PRZESZUKIWANIESYNCHRONICZNE1A(x.left, y.left);
9:     PRZESZUKIWANIESYNCHRONICZNE1A(x.right, y.right);
10: end
```

Aby przetworzyć wszystkie sufiksy naszego kodu, musimy wywołać procedurę PRZESZUKIWANIESYNCHRONICZNE1A(ε, y) dla każdego wierzchołka y. Możemy jednak dokonać drobnej optymalizacji. Jeśli w czasie przechodzenia x stanie się liściem, to zostanie ustawione z powrotem na korzeń. Wartości zmiennych x i y są więc wówczas identyczne jak dla bezpośredniego wywołania PRZESZUKIWANIESYNCHRONICZNE1A(ε, y) dla aktualnej wartości wierzchołka y. Takie wywołanie zostanie (lub zostało) wykonane, więc w tym momencie można je pominąć.

Ostateczna postać naszego algorytmu wyznaczającego wierzchołki bardzo ważne to:

```
 1: procedure PRZESZUKIWANIESYNCHRONICZNE1B(x, y)
 2: begin
 3:   if x jest liściem then
 4:     return; { omawiana optymalizacja }
 5:   if y jest liściem then
 6:     B := B ∪ {x};
 7:   else
 8:     PRZESZUKIWANIESYNCHRONICZNE1B(x.left, y.left);
 9:     PRZESZUKIWANIESYNCHRONICZNE1B(x.right, y.right);
10: end
11: { Kod wyznaczający wierzchołki bardzo ważne: }
12: B := ∅;
13: forall q in T do { T — zbiór wszystkich wierzchołków }
14:   PRZESZUKIWANIESYNCHRONICZNE1B(ε, q);
15: return B;
```

Koszt przetworzenia pojedynczego poddrzewa o korzeniu w wierzchołku q jest proporcjonalny do liczby wierzchołków w tym poddrzewie, którą oznaczymy przez S_q. Ten koszt należy przesumować po wszystkich wierzchołkach, aby dostać złożoność całego algorytmu: $O\left(\sum_{q \in T} S_q\right)$.

Obliczenie tej sumy zostawimy na koniec. Zapowiemy teraz jedynie, że złożoność ta wynosi $O(|w_1| + |w_2| + \ldots + |w_N|) = O(L)$, czyli jest znacznie lepsza niż dla algorytmu naiwnego; dodatkowo, warunek z treści zadania gwarantuje, że $L \leqslant 10^8$, czyli że jest to wartość stosunkowo nieduża.

Efektywny algorytm wyznaczania wierzchołków bardzo ważnych

Algorytm poprawiony ma tę zaletę, że zastosowana w nim technika jest identyczna z techniką używaną w pozostałych krokach rozwiązania. Nie jest on jednak ciągle optymalny. Jako ciekawostkę podamy algorytm o pesymistycznej złożoności $O(N)$, używający techniki podobnej do przetwarzania wstępnego w algorytmie Aho–Corasicka ([35]). Ta poprawka nie zmienia jednak pesymistycznej złożoności całego rozwiązania zadania.

Dla każdego wierzchołka q drzewa T wyznaczymy następujące wierzchołki:

- $L[q]$ — najniższy liść odpowiadający pewnemu właściwemu sufiksowi etykiety q,
- $I[q]$ — najniższy wierzchołek wewnętrzny odpowiadający pewnemu właściwemu sufiksowi etykiety q (I pochodzi od angielskiego *internal node*).

Właściwy sufiks słowa to końcowy fragment tego słowa krótszy niż całe słowo, a *najniższy* znaczy będący najdalej od korzenia, czyli mający najdłuższą etykietę. Jeśli obliczenia będziemy wykonywać, poczynając od wierzchołków położonych najwyżej (czyli od wierzchołków o mniejszej odległości od korzenia), wartości L i I dla aktualnego wierzchołka będziemy mogli wyznaczyć w oparciu o wartości L i I ojca.

Faktycznie, niech p będzie ojcem wierzchołka q, a b bitem na krawędzi łączącej te wierzchołki. $I[p]$ daje najniższy wierzchołek wewnętrzny odpowiadający właściwemu sufiksowi etykiety p. Z $I[p]$ można przejść po bicie b do pewnego wierzchołka x. Nietrudno

się przekonać, że etykieta x jest sufiksem etykiety q. W dodatku x jest najniższym wierzchołkiem o tej własności — jeśli istniałby niższy, to jego rodzic odpowiadałby sufiksowi etykiety p, a byłby niżej niż $I[p]$, co prowadzi do sprzeczności. Mamy dwa przypadki:

- Jeśli x jest liściem, to jest najniższym liściem odpowiadającym właściwemu sufiksowi etykiety q. Wtedy sufiks etykiety q odpowiadający najniższemu wierzchołkowi wewnętrznemu musi być też sufiksem etykiety x.
- Jeśli x jest wierzchołkiem wewnętrznym, to jest najniższym wierzchołkiem wewnętrznym odpowiadającym właściwemu sufiksowi etykiety q. Wtedy sufiks etykiety q odpowiadający najniższemu liściowi, jeśli istnieje, musi być też sufiksem etykiety x.

Obliczenia przedstawia poniższy algorytm. Zwróćmy uwagę na szczególny przypadek korzenia drzewa, dla którego każda z wartości $I[\varepsilon]$ oraz $L[\varepsilon]$ jest pusta.

```
1:  L[ε] := nil;
2:  I[ε] := nil;
3:  forall q in T w kolejności BFS[1], oprócz korzenia do
4:      p := q.parent;
5:      b := etykieta krawędzi między p i q;
6:      if I[p] = nil then
7:          x := ε;
8:      else
9:          x := δ(I[p], b);
10:     if x jest liściem then
11:         L[q] := x;
12:         I[q] := I[x];
13:     else
14:         I[q] := x;
15:         L[q] := L[x];
```

Mając obliczone wartości $I[q]$ (wartości $L[q]$ służyły wyłącznie do wyznaczenia $I[q]$), możemy znaleźć wszystkie wierzchołki odpowiadające sufiksom słów kodowych. Są to wszystkie wierzchołki $I[q]$ dla liści, wszystkie wierzchołki $I[I[q]]$ dla liści itd.

```
1:  B := ∅;
2:  forall q in T w odwróconej kolejności BFS do
3:      if q ∈ B or q jest liściem then
4:          B := B ∪ I[q];
5:  return B;
```

Koszt czasowy tego algorytmu jest ewidentnie proporcjonalny do rozmiaru drzewa, a zatem, na mocy Obserwacji 1, wynosi $O(N)$.

[1] tzn. w kolejności rosnących odległości od korzenia

Wyznaczenie zbioru wierzchołków ważnych

Zakładamy w tym momencie, że zbiór wierzchołków bardzo ważnych B mamy dany. Jak zostało napisane wcześniej, wierzchołki ważne to wierzchołki, do których da się dojść z wierzchołków bardzo ważnych po ciągach słów kodowych. Takie wierzchołki łatwo wyznaczyć, inicjalizując zbiór W zbiorem wierzchołków bardzo ważnych i rozszerzając go kolejno o wierzchołki, do których da się dojść z pewnego wierzchołka z W po pojedynczym słowie kodowym. Procedurę kończymy, gdy przejścia z wierzchołków ze zbioru W po słowach kodowych trafiają zawsze do W. Otrzymany zbiór to zbiór wierzchołków ważnych.

W algorytmie będziemy trzymać kolejkę wierzchołków do przetworzenia, zainicjalizowaną zbiorem wierzchołków bardzo ważnych, oraz zbiór wierzchołków, do którego będziemy wrzucać wierzchołki ważne, również zawierający na początku wszystkie wierzchołki bardzo ważne. Przetworzenie wierzchołka będzie polegać na sprawdzeniu przejść z tego wierzchołka po wszystkich słowach kodowych. Każdy wierzchołek docelowy takiego przejścia jest wierzchołkiem ważnym. Jeśli natrafimy pierwszy raz na ten wierzchołek, czyli jeśli nie ma go w naszym zbiorze, to dodajemy go do zbioru oraz do kolejki przetwarzania. Obliczenia kończą się po opróżnieniu kolejki.

```
1:  W := zbiór wierzchołków bardzo ważnych;
2:  Q.insertAll(W);
3:  while Q ≠ ∅ do
4:    q := Q.popFront();
5:    forall w in K do
6:      q' := δ(q, w);
7:      if q' ∉ W then
8:        W := W ∪ {q'};
9:        Q.insert(q');
10: return W;
```

W tej postaci algorytm ma złożoność czasową $O(N \cdot L)$.

Podobnie jak poprzednio, możemy skorzystać z tego, że wiele słów kodowych posiada wspólne prefiksy. Znowu wykorzystamy synchroniczne przechodzenie po dwóch drzewach. W pierwszym drzewie (zmienna x) startujemy z wierzchołka q, który przetwarzamy. Drugie drzewo (zmienna y) opisuje wszystkie słowa kodowe, po których przechodzimy z wierzchołka q. Zaczynamy w nim więc z korzenia.

Jeśli przy takim przechodzeniu wierzchołek x stanie się liściem, to musi on przeskoczyć od korzenia. Jeśli z kolei wierzchołek y dojdzie do liścia, to skończyło się słowo kodowe i do aktualnego wierzchołka x da się dojść z wierzchołka q po całym słowie kodowym. Zatem wierzchołek x dodajemy do zbioru wierzchołków ważnych, jeśli go tam jeszcze nie ma, i dodajemy do przyszłego przetwarzania. W tej postaci złożoność algorytmu wynosi $O(N^2)$, gdyż jest maksymalnie $O(N)$ wierzchołków ważnych (patrz Obserwacja 1) i dla każdego z nich wykonujemy $O(N)$ operacji.

Aby przyspieszyć działanie algorytmu (jak się wkrótce okaże, poprawia to również jego złożoność), zapamiętujemy, jakie pary (ε, y) były już przetwarzane. Nie ma sensu przetwarzać jakiejś pary powtórnie, więc po napotkaniu takiej pary kolejny raz, nie kontynuujemy rekurencyjnych wywołań.

```
1:  procedure PRZESZUKIWANIESYNCHRONICZNE2(x, y)
2:  begin
3:    if x jest liściem then
4:      PRZESZUKIWANIESYNCHRONICZNE2(ε, y);
5:    else
6:      if x = ε then
7:        if y.visited then return;
8:        else y.visited := true;
9:      if y jest liściem then
10:       if x ∉ W then
11:         Q.insert(x);
12:         W := W ∪ {x};
13:     else
14:       PRZESZUKIWANIESYNCHRONICZNE2(x.left, y.left);
15:       PRZESZUKIWANIESYNCHRONICZNE2(x.right, y.right);
16: end
17: { Kod wyznaczający wierzchołki ważne: }
18: W := zbiór wierzchołków bardzo ważnych;
19: Q.insertAll(W);
20: forall x in T do
21:   x.visited := false;
22: while Q ≠ ∅ do
23:   PRZESZUKIWANIESYNCHRONICZNE2(Q.popFront(), ε);
24: return W;
```

Aby oszacować złożoność tego algorytmu, zauważmy, że każda para (x,ε) oraz (ε,y) występuje w przeszukiwaniu co najwyżej raz. Jest to zapewnione odpowiednio przez to, że każdy wierzchołek pojawia się na kolejce co najwyżej raz, oraz przez flagę *visited*. Liczba operacji potrzebna do przetworzenia pojedynczej pary (x,ε) lub (ε,y), z pominięciem operacji wykonanych w ramach rekurencyjnego przetwarzania innych par (x,ε) oraz (ε,y), jest proporcjonalna do rozmiaru poddrzewa zaczepionego w x lub, odpowiednio, w y. Istotnie, wartość pierwszego elementu z pary (x,ε) podczas takiego przetwarzania nie może wyjść poza poddrzewo zaczepione w x. Analogicznie, wartość drugiego elementu pary (ε,y) nie może wyjść poza poddrzewo zaczepione w y.

Złożoność algorytmu wyznaczania wierzchołków ważnych z wierzchołków bardzo ważnych wynosi zatem $O\left(\sum_{q\in T} S_q\right)$.

Znalezienie kodów synchronizujących

Znaleźliśmy już wierzchołki ważne. Słowo w jest synchronizujące, jeśli dla każdego wierzchołka ważnego q zachodzi $\delta(q,w)=\varepsilon$. Najprostszy algorytm znajdowania słów synchronizujących wygląda następująco.

```
1: W := zbiór wierzchołków ważnych;
2: S := ∅;
3: forall w in K do
```

```
4:    b := true;
5:    forall q in W do
6:       if δ(q,w) ≠ ε then
7:          b := false;
8:          break;
9:    if b = true then
10:      S := S ∪ {w};
11: return S;
```

Ten algorytm ma złożoność $O(N \cdot L)$. Metoda jego przyspieszenia jest podobna jak poprzednio. W głównej pętli przeglądamy wszystkie wierzchołki ważne. Dla każdego wierzchołka analizujemy od razu wszystkie słowa kodowe, wykorzystując synchroniczne przechodzenie drzewa, i patrzymy, które z nich *nie* synchronizują danego wierzchołka. Te słowa kodowe nie mogą być synchronizujące.

Podobnie jak w algorytmie znajdowania wierzchołków ważnych, w pierwszym drzewie (zmienna x) startujemy z wierzchołka q, który przetwarzamy. Drugie drzewo (zmienna y) opisuje wszystkie słowa kodowe.

Taki poprawiony algorytm będzie miał złożoność $O(N^2)$ (dla każdego wierzchołka ważnego wykonane będzie co najwyżej N operacji). Dzięki wykorzystaniu optymalizacji obcinającej pary (ε, y), które uprzednio były przetwarzane, złożoność algorytmu można obniżyć do $O\left(\sum_{q \in T} S_q\right)$. Szczegółowa analiza złożoności tego algorytmu jest analogiczna do analizy złożoności algorytmu wyznaczania wierzchołków ważnych, więc ją pomijamy.

```
1:  procedure PRZESZUKIWANIESYNCHRONICZNE3(x, y)
2:  begin
3:     if y jest liściem then
4:        if x nie jest liściem then
5:           y.synchronizing := false;
6:     else
7:        if x jest liściem then
8:           if not y.visited then
9:              y.visited := true;
10:             PRZESZUKIWANIESYNCHRONICZNE3(ε, y);
11:       else
12:          PRZESZUKIWANIESYNCHRONICZNE3(x.left, y.left);
13:          PRZESZUKIWANIESYNCHRONICZNE3(x.right, y.right);
14: end
15: { Wyznaczanie kodów synchronizujących: }
16: W := zbiór wierzchołków ważnych;
17: forall x in liście T do
18:    x.synchronizing := true;
19: forall q in wierzchołki wewnętrzne T do
20:    q.visited := false;
21: forall q in W do
22:    PRZESZUKIWANIESYNCHRONICZNE3(q, ε);
```

Analiza złożoności

We wszystkich trzech krokach algorytmu złożoność wynosiła $O\left(\sum_{q\in T} S_q\right)$. Sugerowaliśmy, że jest ona równa $O(|w_1|+|w_2|+\ldots+|w_N|)$.

Udowodnimy tę własność, pokazując, że

$$\sum_{q\in T} S_q = 1+2\sum_{w\in K}|w|.$$

Dowód przeprowadzimy przez indukcję względem rozmiaru drzewa. Dla drzewa składającego się z samego korzenia równość jest prawdziwa: $1 = 1+2\cdot 0$. Załóżmy, że równość jest prawdziwa dla drzew T_1 i T_2 odpowiadających kodom odpowiednio K_1 oraz K_2. Pokażemy, że jest ona prawdziwa również dla drzewa T odpowiadającemu kodowi K, składającego się z drzew T_1 i T_2 połączonych wspólnym korzeniem.

$$\begin{aligned}
\sum_{q\in T} S_q &= S_\varepsilon + \sum_{q\in T_1} S_q + \sum_{q\in T_2} S_q = && (1)\\
&= (2\cdot|K|-1) + \left(1+2\sum_{w\in K_1}|w|\right) + \left(1+2\sum_{w\in K_2}|w|\right) = && (2)\\
&= 1+2\cdot|K_1|+2\cdot|K_2|+2\sum_{w\in K_1}|w|+2\sum_{w\in K_2}|w| = && (3)\\
&= 1+2\left(|K_1|+\sum_{w\in K_1}|w|\right)+2\left(|K_2|+\sum_{w\in K_2}|w|\right) = && (4)\\
&= 1+2\sum_{w\in K_1}(|w|+1)+2\sum_{w\in K_2}(|w|+1) = && (5)\\
&= 1+2\sum_{w\in K}|w|. && (6)
\end{aligned}$$

Zaznaczmy, że w przejściu (1)–(2) korzystamy z Obserwacji 1.

Każdy z trzech kroków algorytmu ma złożoność $O(|w_1|+|w_2|+\ldots+|w_N|) = O(L)$, więc cały algorytm ma taką złożoność. Pierwszy krok da się zoptymalizować do złożoności $O(N)$, jednak nie wpływa to na złożoność całego rozwiązania.

Implementacje rozwiązania wzorcowego można znaleźć w plikach `kod.cpp`, `kod1.pas` i `kod2.java`.

Testy

Zadanie było sprawdzane na 10 zestawach danych testowych. Zestawy zawierały od jednego do pięciu testów. Testowe kody zostały przygotowane za pomocą losowego generatora drzew sterowanego kilkoma parametrami. Generator decydował, czy dzieci aktualnego wierzchołka mają być liśćmi, czy wierzchołkami wewnętrznymi. Dla tych ostatnich obliczał nowy zestaw parametrów i działał rekurencyjnie.

Parametry generatora, opisane poniżej, wpływały na kształt drzew, czyli ich wielkość, wysokość oraz stopień zrównoważenia.

- I — liczba wierzchołków wewnętrznych drzewa.

- Z — *zrównoważenie* drzewa — liczba od 0 do 100, przy czym 50 oznacza drzewo zrównoważone, 0 drzewo z dużą liczbą wierzchołków w lewym poddrzewie a małą w prawym, a 100 odwrotnie. Z steruje podziałem liczby I pomiędzy synów danego wierzchołka.

- LZ — losowość zrównoważenia — liczba od 0 (mała losowość) do 100 (duża losowość). LZ opisuje losowość podziału I między synów danego wierzchołka.

- ZZ — zmiana zrównoważenia — prawdopodobieństwo (w procentach), że w danym wierzchołku zrównoważenie zmieni się na $100 - Z_p$, gdzie Z_p to zrównoważenie ojca wierzchołka. Dzięki ZZ drzewa niezrównoważone nie są skrzywione w jedną stronę.

- L — limit — jeśli liczba wierzchołków wewnętrznych spada poniżej tego limitu, to jeden z synów danego wierzchołka musi być liściem. Dzięki parametrowi L można było wygenerować drzewo zrównoważone, w którym długości ścieżek od liści do korzenia (czyli długości kodów) mają znaczną wariancję.

Parametry użyte do wygenerowania testów opisuje poniższa tabelka.

Nazwa	**I**	**Z**	**LZ**	**ZZ**	**L**
kod1.in	50	90	10	20	16
kod2.in	100	50	10	90	60
kod3a.in	200	30	30	80	40
kod3b.in	200	50	0	50	4
kod4a.in	1 000	5	6	20	20
kod4b.in	1 000	99	20	90	30
kod4c.in	2 000	10	6	13	4
kod4d.in	2 000	5	5	54	14
kod4e.in	2 020	3	60	20	13
kod5a.in	20 000	70	20	10	3 000
kod5b.in	30 000	10	20	10	200
kod6.in	100 000	45	30	20	300
kod7.in	500 000	65	30	20	600
kod8.in	500 000	50	20	50	300
kod9a.in	600 000	30	10	30	450
kod9b.in	560 000	50	0	0	10
kod10a.in	20 000	50	20	0	7 000
kod10b.in	600 000	35	30	60	800

Kontekst zadania

Zadanie związane jest z rzeczywistym problemem odporności kodów prefiksowych na błędy transmisji. W zadaniu rozważaliśmy błędy powstałe przez pominięcie prefiksu kodowanego ciągu. Podobnie można rozważać błędy zamiany bitów na przeciwne lub wycięcia środkowego fragmentu wiadomości. Kody prefiksowe, ze względu na właściwości synchronizujące, wykazują wysoką odporność na błędy. Po utracie synchronizacji dekoder spontanicznie wraca do poprawnego odkodowywania. Dzieje się to średnio już po niewielkiej liczbie bitów.

Właściwości synchronizujące kodów poprawia istnienie synchronizujących słów kodowych — tych rozważanych w zadaniu, synchronizujących wierzchołki ważne, a także bardziej ogólnych, synchronizujących wszystkie wierzchołki. Nawet jeśli żadne pojedyncze słowo kodowe nie synchronizuje wszystkich wierzchołków, często można znaleźć ciąg słów kodowych posiadający taką własność. Więcej na temat synchronizacji kodów prefiksowych Czytelnik znajdzie w [34].