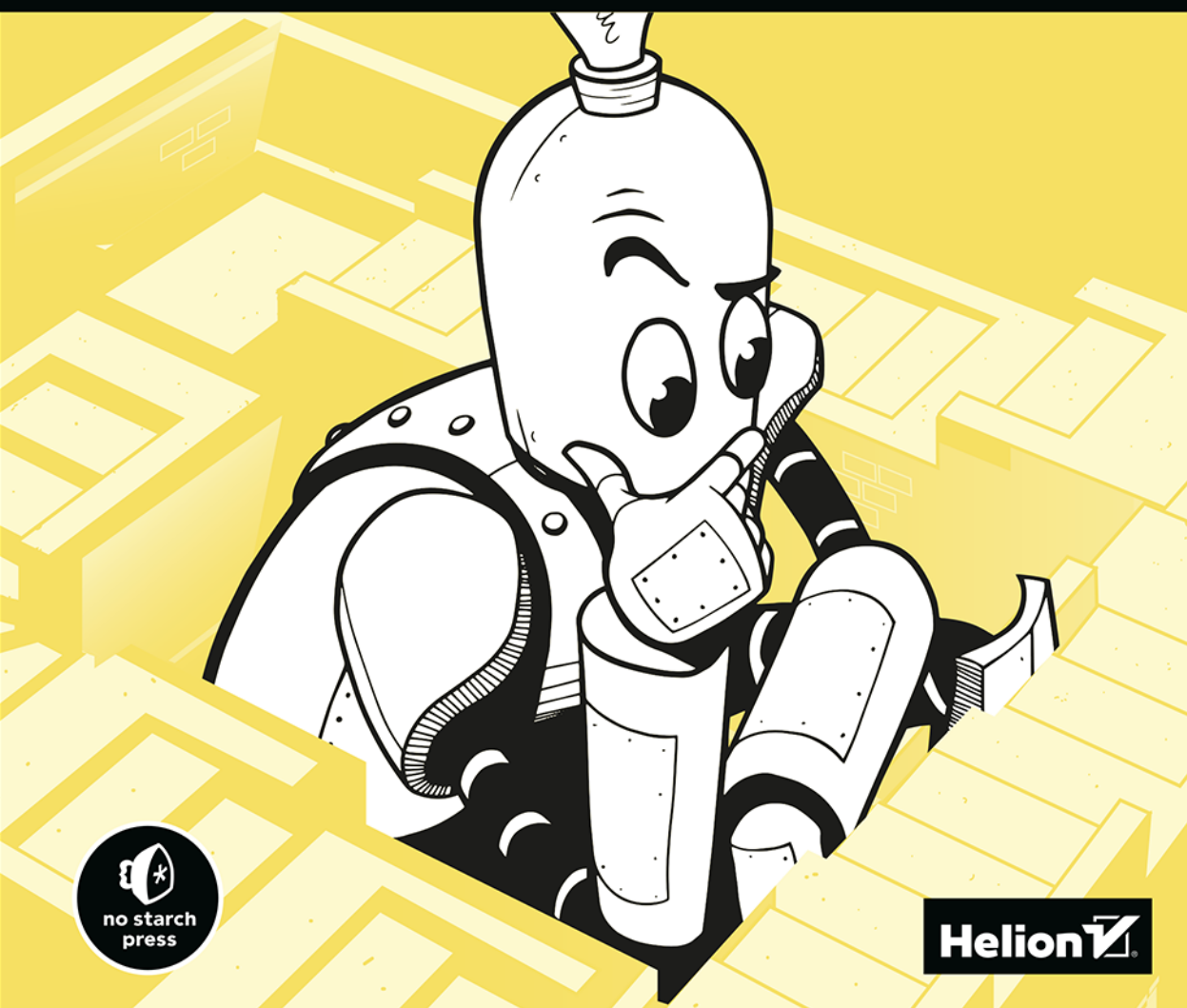
MYŚLENIE ALGORYTMICZNE

JAK ROZWIĄZYWAĆ PROBLEMY ZA POMOCĄ ALGORYTMÓW

DANIEL ZINGARO



no starch press

Helion

Tytuł oryginału: Algorithmic Thinking: A Problem-Based Introduction

Tłumaczenie: Piotr Rajca

ISBN: 978-83-283-8335-7

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/algwro
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/algwro.zip

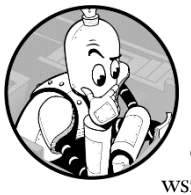
Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

1

Tablice mieszające



W TYM ROZDZIALE ZAJMIEMY SIĘ DWOMA PROBLEMAMI, KTÓRYCH ROZWIĄZANIE BAZUJE NA MOŻLIWOŚCI WYKONYWANIA WYDAJNYCH OPERACJI WYSZUKIWANIA. PIERWSZY POLEGA NA SPRAWDZENIU, CZY wszystkie płatki śniegu w kolekcji są identyczne, drugi na określeniu, które ze słów są słowami złożonymi. Chcemy rozwiązać te problemy prawidłowo, ale okaże się, że niektóre z tych prawidłowych rozwiązań działają po prostu zbyt wolno. Przekonamy się jednak, że dzięki zastosowaniu struktury danych nazywanej tablicą mieszającą będziemy w stanie zapewnić ogromną poprawę wydajności działania; dlatego też przyjrzymy się tej strukturze bardzo dokładnie.

Rozdział zakończymy przedstawieniem trzeciego problemu: określeniem, na ile sposobów można usunąć literę z jednego słowa, by została dodana do kolejnego. Na tym przykładzie poznamy ryzyko wiążące się z bezkrytycznym stosowaniem naszej nowej struktury danych — kiedy uczymy się czegoś nowego, kusi nas, by wszędzie stosować to rozwiązanie.

Problem 1. Płatki śniegu

Problem „Płatki śniegu” jest dostępny na witrynie DMOJ i ma symbol `cco07p2`.

Problem

Mamy do dyspozycji kolekcję płatków śniegu i musimy określić, czy którekolwiek z nich są identyczne.

Płatek śniegu jest reprezentowany przy użyciu sześciu liczb całkowitych, z których każda określa długość jednego z ramion płatka. Poniższy wiersz zawiera przykładowy opis płatka śniegu:

```
3, 9, 15, 2, 1, 10
```

Liczby występujące w opisach płatków śniegu mogą się także powtarzać, jak w poniższym przykładzie:

```
8, 4, 8, 9, 2, 8
```

Co oznacza, że dwa płatki śniegu są identyczne? Ustalimy tę definicję identyczności, analizując kilka kolejnych przykładów.

W pierwszej kolejności przyjrzyjmy się dwóm płatkom śniegu:

```
1, 2, 3, 4, 5, 6
```

i

```
1, 2, 3, 4, 5, 6
```

Bez wątpienia są identyczne, gdyż liczby z opisu pierwszego z nich odpowiadają liczbom podanym na tych samych pozycjach w opisie drugiego.

A oto drugi przykład:

```
1, 2, 3, 4, 5, 6
```

i

```
4, 5, 6, 1, 2, 3
```

Także te dwa płatki są identyczne. By się o tym przekonać, zaczynamy od liczby `1` w opisie drugiego płatka i odczytujemy kolejne liczby położone na prawo od `1`. Czytamy je zatem w takiej kolejności: `1`, `2`, `3`, a następnie, kiedy przejdziemy na początek ciągu, kontynuujemy odczytywanie: `4`, `5`, `6`. Jeśli połączymy te dwa ciągi liczb, uzyskamy opis identyczny z opisem pierwszego płatka śniegu.

Każdy płatek śniegu możemy sobie wyobrazić jako okrąg. Te dwa płatki są takie same, gdyż możemy wybrać takie miejsce w opisie drugiego z nich, że jeśli zaczniemy tam odczytywać kolejne liczby opisu, uzyskamy taki sam ciąg liczb jak w opisie pierwszego płatka.

Kolejny przykład jest nieco bardziej skomplikowany:

```
1, 2, 3, 4, 5, 6
```

i

```
3, 2, 1, 6, 5, 4
```

Na podstawie wcześniejszych przykładów moglibyśmy stwierdzić, że te dwa płatki śniegu nie są identyczne. Jeśli zaczniemy od 1 w opisie drugiego płatka i będziemy odczytywać kolejne liczby, przesuwając się w prawo (i przechodząc następnie na sam początek opisu z lewej strony), uzyskamy taką sekwencję liczb: 1, 6, 5, 4, 3, 2. W żadnym razie nie możemy uznać, że ten opis w jakikolwiek sposób przypomina opis pierwszego płatka, czyli: 1, 2, 3, 4, 5, 6.

Niemniej, jeśli zaczniemy odczytywać kolejne liczby opisu drugiego płatka od 1 i przesuniemy się w lewo, a nie w prawo, to uzyskamy opis identyczny z opisem pierwszego płatka, czyli właśnie 1, 2, 3, 4, 5, 6! Począwszy od 1, przesuwając się w lewo, odczytujemy kolejno liczby: 1, 2 i 3, a następnie, kiedy przejdziemy na prawy koniec ciągu i dalej będziemy się posuwać w lewo, odczytamy dalszą część opisu: 4, 5 i 6.

I to jest trzeci sposób określania identyczności płatków śniegu: płatki także są identyczne, jeśli ich opisy będą takie same w wypadku odczytywania od prawej do lewej.

Podsumowując te wszystkie przykłady, możemy dojść do wniosku, że dwa płatki śniegu są identyczne, jeśli ich opisy są takie same, jeśli możemy sprawić, że będą takie same, poprzez odczytanie opisu jednego z nich od lewej do prawej albo też jeśli możemy sprawić, że będą takie same, poprzez odczytanie opisu jednego z nich od prawej do lewej.

Dane wejściowe

Pierwszy wiersz danych wejściowych zawiera liczbę całkowitą n określającą liczbę płatków śniegu, jakie należy przetworzyć. Wartość n należy do zakresu od 1 do 100 000. Każdy z kolejnych n wierszy reprezentuje jeden płatek śniegu i zawiera sześć liczb całkowitych, których wartości są nie mniejsze od 0 i nie większe od 10 000 000.

Wyniki

Wynikiem wykonania programu będzie pojedynczy wiersz tekstu o takiej treści[1]:

- Jeśli nie będzie identycznych płatków śniegu, należy wyświetlić tekst `No two snowflakes are identical.`[2]
- Jeśli uda się znaleźć przynajmniej dwa identyczne płatki śniegu, to należy wyświetlić tekst `Twin snowflakes found.`[3]

Limit czasu na rozwiązanie wszystkich przypadków testowych wynosi 2 sekundy.

Uproszczenie problemu

Jedna z ogólnych strategii używanych podczas rozwiązywania zadań na konkursach programistycznych polega na rozpoczynaniu od próby rozwiązania nieco uproszczonego problemu. A zatem, w ramach rozgrzewki, spróbujmy usunąć trochę złożoności z naszego problemu związanego z porównywaniem płatków śniegu.

Załóżmy, że zamiast na płatkach śniegu opisanych przy użyciu wielu liczb całkowitych będziemy operować na opisach składających się z jednej liczby. A zatem dysponujemy kolekcją liczb całkowitych i chcemy określić, czy występują wśród nich identyczne. W języku C identyczność liczb całkowitych możemy sprawdzać przy użyciu operatora `==`. Możemy więc sprawdzać kolejne pary liczb i jeśli uda się nam znaleźć choćby jedną parę tych samych liczb, możemy przerwać porównywanie i wyświetlić wynik:

```
Twin snowflakes found.
```

Jeśli nie uda się znaleźć dwóch identycznych liczb, to wyświetlimy wynik:

```
No two snowflakes are alike.
```

Napiszemy zatem funkcję `identify_identical`, która porównuje pary liczb całkowitych, używając do tego celu dwóch zagnieżdżonych pętli. Kod tej funkcji przedstawiam na listingu 1.1.

[1] Problemy prezentowane w książce pochodzą z witryn prowadzących konkursy programistyczne. Rozwiązania tych problemów cały czas można przesyłać i są one sprawdzane przez witryny. Z tego względu generowane wyniki pozostawiamy w języku angielskim. Generowanie spolonizowanych tekstów oznaczałoby, że czytelnik, który pokusi się o przygotowanie własnego rozwiązania, zawsze będzie uzyskiwał błędne wyniki. — *przyp. red.*

[2] Nie ma dwóch identycznych płatków śniegu — *przyp. tłum.*

[3] Znaleziono identyczne płatki śniegu — *przyp. tłum.*

Listing 1.1. Funkcja znajdująca pary identycznych liczb całkowitych

```
void identify_identical(int values[], int n) {
  int i, j;
  for (i = 0; i < n; i++) {
    for (j = i+1; j < n; j++) {    ❶
      if (values[i] == values[j]) {
        printf("Twin integers found.\n");
        return;
      }
    }
  }
  printf("No two integers are alike.\n");
}
```

Liczby całkowite przekazujemy do funkcji przy użyciu tablicy `values`. Przekazujemy do niej także zmienną `n` określającą liczbę wartości zapisanych w tablicy.

Zwróć uwagę, że działanie wewnętrznej pętli rozpoczyna się od wartości `i+1`, a nie `0` ❶. Gdybyśmy zaczęli od wartości `0`, to wcześniej czy później zmienne `i` i `j` przyjęłyby tę samą wartość, co oznaczałoby, że porównywalibyśmy ten sam element tablicy. A to z kolei zwróciłoby błędny pozytywny wynik porównywania płatków śniegu.

Spróbujmy teraz przetestować działanie funkcji `identify_identical` przy użyciu prostej funkcji `main`:

```
int main(void) {
  int a[5] = {1, 2, 3, 1, 5};
  identify_identical(a, 5);
  return 0;
}
```

Kiedy wykonasz ten przykład na podstawie wyświetlonych wyników, przekonasz się, że funkcja prawidłowo rozpoznała parę identycznych wartości `1`. Ogólnie rzecz biorąc, nie będę przedstawiał w książce wielu podobnych programów testowych, choć liczę na to — i traktuję to jako niezwykle ważne — że będziesz podczas samodzielnej pracy eksperymentować i testować pisany kod.

Rozwiązywanie podstawowego problemu

Spróbujmy teraz zmodyfikować funkcję `identify_identical` tak, by pozwalała rozwiązywać postawiony problem identyczności płatków śniegu. W tym celu istniejący już kod trzeba rozszerzyć na dwa sposoby:

1. Musimy operować jednocześnie na sześciu liczbach całkowitych, a nie na jednej. Do tego celu z powodzeniem możemy użyć dwuwymiarowej tablicy liczb: każdy jej wiersz będzie reprezentować płatek śniegu i składać się z sześciu kolumn (po jednej kolumnie na element).

2. Jak już wiemy, dwa płatki śniegu mogą być identyczne na trzy sposoby. Niestety, oznacza to, że nie możemy już określać identyczności płatków śniegu, używając prostego operatora ==. Musimy uwzględnić kryterium „odczytu kolejnych liczb od lewej do prawej" i „odczytu liczb od prawej do lewej". (Pomijam już zupełnie to, że stosowany w języku C operator == nie pozwala na porównywanie tablic!) To odpowiednie zmodyfikowanie sposobu porównywania płatków śniegu będzie kluczową aktualizacją, jaką będziemy musieli wprowadzić w algorytmie.

Zaczniemy od napisania pary funkcji pomocniczych, odpowiedzialnych odpowiednio za porównywanie płatków śniegu „od lewej do prawej" i „od prawej do lewej". Każda z nich będzie mieć trzy parametry: pierwszy z porównywanych płatków śniegu, drugi płatek i punkt początkowy, od którego rozpocznie się porównywanie drugiego płatka.

Sprawdzanie od lewej do prawej

Poniżej przedstawiam nagłówek funkcji `identical_right`:

```
int identical_right(int snow1[], int snow2[], int start) {
```

Aby określić, czy dwa płatki śniegu są identyczne, na podstawie porównywania ich opisów od lewej do prawej, będziemy przeglądać tablicę `snow1` począwszy od elementu o indeksie `0` i tablicę `snow2` począwszy od elementu o indeksie `start`. Jeśli się okaże, że odpowiadające sobie elementy nie są identyczne, funkcja zwróci 0, co będzie oznaczało, że nie udało się znaleźć identycznych płatków śniegu. Jeśli wszystkie porównywane elementy będą takie same, funkcja zwróci 1. Wyobraź sobie, że wartość 0 oznacza logiczny fałsz, a 1 — logiczną prawdę.

Listing 1.2 przedstawia pierwszą próbę napisania kodu tej funkcji.

Listing 1.2. Sprawdzanie identyczności płatków śniegu poprzez odczyt opisu od lewej do prawej (z błędami!)

```
int identical_right(int snow1[], int snow2[], int start) { //z błędami!
  int offset;
  for (offset =0; offset < 6; offset++) {
    if (snow1[offset] != snow2[start + offset])  ❶
      return 0;
  }
  return 1;
}
```

Jak zapewne już wiesz, ten kod nie będzie działał zgodnie z naszymi oczekiwaniami. Problemem jest wyrażenie `start + offset` ❶. Jeśli zmienna `start` będzie mieć wartość `4`, a `offset` — wartość 3, wyrażenie to przyjmie wartość `7`.

A odwołanie o postaci `snow2[7]` oznacza problemy, gdyż ostatnim elementem tablicy `snow2`, do którego możemy się odwołać, jest `snow2[5]`.

Ten kod nie uwzględnia tego, że po dotarciu do końca opisu (jego prawy kraniec) musimy przejść na jego początek (z lewej strony). Jeśli nasz kod będzie chciał użyć błędnego indeksu o wartości `6` lub większej, będziemy musieli zmodyfikować ten indeks poprzez odjęcie od niego wartości `6`. W ten sposób będziemy mogli kontynuować porównywanie opisów płatków, używając indeksu `0` zamiast `6`, `1` zamiast `7` itd. Spróbujmy zatem użyć nowego kodu, który przedstawiam na listingu 1.3.

Listing 1.3. Sprawdzanie identyczności płatków śniegu poprzez odczyt opisu od lewej do prawej

```
int identical_right_2(int snow1[], int snow2[], int start) {
  int offset, snow2_index;
  for (offset =0; offset < 6; offset++) {
    snow2_index = start + offset;
    if (snow2_index >= 6)
      snow2_index = snow2_index - 6;
    if (snow1[offset] != snow2[snow2_index])
      return 0;
  }
  return 1;
}
```

Ta funkcja działa prawidłowo, lecz wciąż możemy ją dodatkowo usprawnić. Jedną ze zmian, na które zdecydowałoby się zapewne wielu programistów, byłoby zastosowanie operatora `%` — dzielenia modulo. Operator `%` zwraca resztę z dzielenia, a zatem wyrażenie `x % y` zwraca resztę z dzielenia całkowitego liczby `x` przez liczbę `y`. Na przykład wyrażenie `6 % 3` zwraca `0`, gdyż po podzieleniu liczby `6` przez `3` nie uzyskujemy żadnej reszty. Z kolei wyrażenie `6 % 4` zwraca `2`, gdyż podzielenie `6` przez `4` daje resztę `2`.

Możemy skorzystać z operatora `%` do obsługi przejścia z końca tablicy na jej początek. Zwróć uwagę, że wyrażenie `0 % 6` zwraca `0`, wyrażenie `1 % 6` daje `1` itd., a `5 % 6` daje `5`. Każda z tych liczb jest mniejsza od `6`, więc zostanie zwrócona jako wynik dzielenia przy użyciu operatora `%`. Liczby od `0` do `5` reprezentują prawidłowe indeksy tablicy `snow2`, dobrze zatem, że operator `%` ich nie zmienia. W wypadku pierwszego ze sprawiających problem indeksów, `6`, wyrażenie `6 % 6` zwraca `0` — podzielenie `6` przez `6` nie daje bowiem żadnej reszty, a użycie tego wyrażenia jako indeksu będzie odpowiadać przejściu na początek tablicy. A to jest dokładnie to, o co nam chodzi.

Zmodyfikujmy zatem funkcję `identical_right`, używając w niej operatora `%`; nową postać kodu przedstawiam na listingu 1.4.

Listing 1.4. Sprawdzanie identyczności płatków śniegu z użyciem operatora %

```
int identical_right(int snow1[], int snow2[], int start) {
  int offset;
  for (offset =0; offset < 6; offset++) {
    if (snow1[offset] != snow2[(start + offset) % 6])
      return 0;
  }
  return 1;
}
```

To, czy zastosujesz sztuczkę z operatorem `%`, czy nie, zależy już tylko od Ciebie. Pozwala ona zaoszczędzić wiersz kodu i jest często stosowanym wzorcem, który wielu programistów będzie w stanie zidentyfikować. Niemniej zastosowanie tego rozwiązania nie zawsze jest równie łatwe, i to nawet w problemach, w których występuje podobne przejście z końca na początek. Tak właśnie będzie w wypadku drugiej z naszych funkcji pomocniczych, `identical_left`, którą zajmiemy się w następnym podpunkcie.

Sprawdzanie od prawej do lewej

Funkcja `identical_left` jest bardzo podobna do `identical_right`, z tą różnicą, że opisy płatków śniegu analizujemy w niej od prawej do lewej, a następnie przechodzimy na prawo, czyli na koniec opisu. W poprzedniej funkcji, analizującej opisy od lewej do prawej, musieliśmy sprawdzać, czy indeks nie przyjmie błędnej wartości większej od `6` lub równej `6`; w tej funkcji natomiast musimy zapewnić, że nie przyjmie on wartości `-1` lub mniejszej.

Niestety, w tym wypadku nie możemy skorzystać z operatora `%`. W języku C wyrażenie `-1 / 6` zwraca `0` i resztę z dzielenia `-1`, a zatem `-1 % 6` daje `-1`. A my potrzebowalibyśmy, by wyrażenie `-1 % 6` przyjmowało wartość `5`.

Kod funkcji `identical_left` przedstawiam na listingu 1.5.

Listing 1.5. Sprawdzanie identyczności płatków śniegu poprzez odczyt opisu od prawej do lewej

```
int identical_left(int snow1[], int snow2[], int start) {
  int offset, snow2_index;
  for (offset = 0; offset < 6; offset++) {
    snow2_index = start - offset;
    if (snow2_index < 0)
      snow2_index = snow2_index + 6;
    if (snow1[offset] != snow2[snow2_index])
      return 0;
  }
  return 1;
}
```

Zwróć uwagę na podobieństwo tej funkcji z funkcją z listingu 1.3. Jedyne różnice pomiędzy nimi to odejmowanie zmiennej `offset` zamiast jej dodawania, a także zmiana indeksu po uzyskaniu wartości `-1`, a nie `6`.

Połączenie fragmentów w całość

Skoro dysponujemy już tymi dwiema funkcjami pomocniczymi, `identical_right` i `identical_left`, możemy napisać funkcję, która będzie sprawdzać, czy dwa płatki śniegu są identyczne. Funkcja ta będzie nosić nazwę `are_identical`, jej kod przedstawiam na listingu 1.6. Działanie tej funkcji sprowadza się do porównania płatków śniegu poprzez analizę ich opisów od lewej do prawej i od prawej do lewej począwszy od każdego z miejsc opisu drugiego płatka (`snow2`).

Listing 1.6. Sprawdzanie identyczności płatków śniegu

```
int are_identical(int snow1[], int snow2[]) {
  int start;
  for (start = 0; start < 6; start++) {
    if (identical_right(snow1, snow2, start))     ❶
      return 1;
    if (identical_left(snow1, snow2, start))      ❷
      return 1;
  }
  return 0;
}
```

Sprawdzamy, czy płatki `snow1` i `snow2` są identyczne, analizując opis `snow2` od lewej do prawej ❶. Jeśli się okaże, że według tych kryteriów płatki są identyczne, zwracamy wartość `1` (prawdę). W przeciwnym razie wykonujemy podobne sprawdzenie, analizując opis drugiego płatka od prawej do lewej ❷.

W tym miejscu warto się zatrzymać i przetestować działanie funkcji `are identical` na kilku przykładowych parach płatków. Zrób to, zanim przejdziesz do dalszej lektury!

Rozwiązanie 1. Porównywanie parami

Za każdym razem, kiedy musimy porównać dwa płatki, zamiast używać operatora `==`, wywołujemy funkcję `are_identical`. Dzięki niej porównywanie płatków jest równie łatwe jak porównywanie liczb całkowitych.

Spróbujmy zmodyfikować przedstawioną wcześniej funkcję `identify_identical` (patrz listing 1.1) tak, by porównywała płatki śniegu, używając do tego celu funkcji `are_identical` (patrz listing 1.6). Będziemy porównywać pary płatków śniegu i dla każdej z nich wyświetlać odpowiedni komunikat zależnie od tego, czy płatki będą identyczne, czy nie. Kod nowej wersji funkcji `identify_identical` przedstawiam na listingu 1.7.

Listing 1.7. Znajdowanie identycznych płatków śniegu

```
void identify_identical(int snowflakes[][6], int n) {
  int i, j;
  for (i = 0; i < n; i++) {
    for (j = i+1; j < n; j++) {
      if (are_identical(snowflakes[i], snowflakes[j])) {
        printf("Twin snowflakes found.\n");
        return;
      }
    }
  }
  printf("No two snowflakes are alike.\n");
}
```

Ta funkcja jest niemal taka sama, z dokładnością do kilku symboli, jak jej pierwsza wersja przedstawiona na listingu 1.1. Jedyną zmianą jest zastąpienie operatora `==` wywołaniem funkcji `are_identical` podczas porównywania płatków śniegu.

Odczyt danych wejściowych

Nasz program nie jest jeszcze w pełni gotowy, by go przesłać na witrynę oceniającą. Nie napisaliśmy wciąż kodu odpowiedzialnego za odczytywanie płatków śniegu ze standardowego strumienia wejściowego. Zacznijmy od ponownego przeczytania opisu problemu zamieszczonego na początku rozdziału. Musimy odczytać wiersz zawierający liczbę całkowitą *n*, która określi liczbę płatków, a następnie wczytać kolejne *n* wierszy z opisami poszczególnych płatków.

Na listingu 1.8 przedstawiam kod funkcji `main`, która przetwarza dane wejściowe i wywołuje funkcję `identify_identical` z listingu 1.7.

Listing 1.8. Funkcja main dla rozwiązania problemu 1.

```
#define SIZE 100000

int main(void) {
  static int snowflakes[SIZE][6];    ❶
  int n, i, j;
  scanf("%d", &n);
  for (i = 0; i < n; i++)
    for (j = 0; j < 6; j++)
      scanf("%d", &snowflakes[i][j]);
  identify_identical(snowflakes, n);
  return 0;
}
```

Zwróć uwagę, że tablica `snowflakes` została zdefiniowana jako statyczna ❶. Użyłem takiego rozwiązania, gdyż jest ona ogromna i bez zastosowania tablicy statycznej ilość potrzebnej pamięci zapewne przekroczyłaby obszar stosu dostępny dla funkcji. Zastosowałem zatem słowo kluczowe `static`, by umieścić tablicę we własnym, odrębnym obszarze pamięci, którego wielkość nie będzie problemem. Niemniej słowo kluczowe `static` należy stosować z rozwagą. Normalnie zmienne lokalne są inicjowane za każdym razem, gdy funkcja jest wywoływana, zmienne statyczne natomiast zachowują swoje wartości pomiędzy kolejnymi wywołaniami (patrz punkt „Słowo kluczowe static" we wprowadzeniu do książki).

Zwróć także uwagę, że przydzieliłem pamięć dla tablicy mogącej pomieścić `100000` płatków śniegu ❶. Możesz uznać, że takie rozwiązanie jest marnowaniem pamięci. Co się stanie, jeśli wprowadzimy dane tylko dla kilku płatków? W wypadku problemów rozwiązywanych w ramach konkursów programistycznych zazwyczaj nic nie stoi na przeszkodzie, by wymagania dotyczące pamięci były określane sztywno, z uwzględnieniem najbardziej obciążającego przypadku: przypadki testowe zapewne sprawdzą działanie rozwiązania, przekazując do niego maksymalną przewidzianą ilość danych!

Dalsza część funkcji `main` jest bardzo prosta. Odczytujemy w niej liczbę płatków śniegu za pomocą funkcji `scanf`, po czym używamy tej liczby w pętli, której każda iteracja wczytuje w pętli sześć liczb całkowitych. Następnie wywołujemy funkcję `identify_identical`, by wygenerować odpowiedni wynik.

Po połączeniu tej funkcji `main` z funkcjami przedstawionymi wcześniej uzyskamy kompletny program, który możemy opublikować na witrynie oceniającej. Spróbuj to zrobić, przekonasz się zapewne, że zamiast powodzenia zostanie wyświetlony błąd przekroczenia limitu czasu (*time-limit exceeded*). Wygląda na to, że musimy jeszcze co nieco popracować nad naszym rozwiązaniem!

Diagnozowanie problemu

Nasze pierwsze rozwiązanie okazało się zbyt wolne i powoduje wystąpienie błędu przekroczenia limitu czasu. Przyczyną problemów są dwie zagnieżdżone pętle `for`, które porównują każdy płatek śniegu ze wszystkimi pozostałymi, co powoduje, że dla dużych wartości n liczba wykonywanych operacji porównania jest ogromna.

Spróbujmy zatem określić liczbę operacji porównania płatków śniegu, które nasz program wykonuje. Ponieważ porównujemy wszystkie pary płatków, pytanie to możemy także sformułować w inny sposób: jaka jest sumaryczna liczba par płatków śniegu? Na przykład, gdybyśmy mieli do dyspozycji cztery płatki, ponumerowane jako 1, 2, 3 i 4, to nasze rozwiązanie wykona sześć operacji porównania płatków; sprawdzone zostaną pary: 1 i 2, 1 i 3, 1 i 4, 2 i 3, 2 i 4, 3 i 4. Każda z nich jest tworzona poprzez wybranie jednego z n płatków jako pierwszego z pary i jednego z n–1 płatków jako drugiego.

Dla każdej z n decyzji dotyczących wyboru pierwszego płatka mamy n–1 decyzji dotyczących wyboru drugiego. Przy czym wartość $n(n–1)$ jest dwa razy większa od faktycznej liczby wykonywanych operacji porównania, gdyż uwzględnia na przykład zarówno parę składającą się z płatków 1 i 2, jak i z płatków 2 i 1. Nasz

program natomiast porównuje te dwa płatki tylko raz, dlatego też powyższe wyrażenie możemy podzielić przez 2. W efekcie uzyskujemy to, że na n płatków śniegu nasze rozwiązanie wykonuje $n(n–1)/2$ operacji porównania.

Może się wydawać, że nie jest to zbyt wiele, spróbujmy jednak podstawić do tego wyrażenia kilka przykładowych wartości. I tak dla $n = 10$ uzyskamy: $10(9)/2 = 45$. Wykonanie 45 porównań nie przysporzy najmniejszych problemów żadnemu komputerowi. A jak to będzie wyglądać dla $n = 10$? W tym wypadku wyrażenie przyjmie wartość 4950; co także nie będzie stanowić żadnego problemu. Wygląda na to, że dla niewielkich wartości n nie będziemy mieć kłopotów. Jednak w opisie problemu podano, że maksymalna dopuszczalna liczba porównywanych płatków śniegu wynosi 100 000. A zatem podstawmy tę wartość do wzoru $n(n–1)/2$; okazuje się, że w tym wypadku zostanie wykonanych 4 999 950 000 operacji porównania. Gdybyśmy wykonali przypadek testowy obejmujący taką liczbę płatków śniegu, to sprawdzenie go na przeciętnym laptopie zajęłoby około 4 minut. To o wiele za dużo — nasze rozwiązanie musi podać wynik w ciągu 2 sekund, a nie kilku minut! Obecnie bezpiecznie możesz założyć, że komputer jest w stanie wykonać około 30 000 000 operacji na sekundę. Jeśli oprzeć się na takim założeniu, to widać, że próba wykonania 4 000 000 000 operacji porównywania płatków śniegu w ciągu 2 sekund jest nierealna.

Przekształciwszy wyrażenie $n(n–2)/2$, uzyskujemy $n^2/2–n/2$. Największym wykładnikiem występującym w tym wyrażeniu jest 2. Twórcy algorytmów określają rozwiązania tego typu jako $O(n^2)$ albo nazywają *algorytmami o złożoności kwadratowej*. Zapis $O(n^2)$ wymawia się jako „rzędu O n kwadrat" i można go sobie wyobrazić jako informację o tempie wzrostu ilości pracy w zależności od wielkości problemu. Krótkie wprowadzenie do zagadnień złożoności algorytmów można znaleźć w dodatku A.

Wykonywanie tak dużej liczby porównań jest konieczne, gdyż identyczne płatki śniegu mogą się znajdować w dowolnych miejscach tablicy. Gdyby istniał jakiś sposób pozwalający na umieszczanie takich płatków bliżej siebie w tablicy, to moglibyśmy znacznie szybciej określić, czy konkretny płatek śniegu jest elementem identycznej pary. Aby umieścić identyczne płatki bliżej siebie w tablicy, możemy spróbować ją posortować.

Sortowanie płatków śniegu

Język C udostępnia funkcję biblioteczną o nazwie `qsort`, która znacząco ułatwia sortowanie tablic. Kluczowym warunkiem jej użycia jest przygotowanie odpowiedniej funkcji porównującej: ma ona pobierać wskaźniki do dwóch sortowanych elementów i zwracać liczbę całkowitą mniejszą od 0, jeśli pierwszy element jest mniejszy od drugiego, wartość `0`, jeśli elementy są równe, liczbę całkowitą większą od 0, jeśli pierwszy element jest większy od drugiego. Dysponujemy już funkcją `are_identical` służącą do sprawdzania, czy dwa płatki śniegu są identyczne; funkcja ta dla takich samych płatków śniegu zwraca wartość `0`.

Co jednak oznacza stwierdzenie, że jeden płatek śniegu jest mniejszy lub większy od drugiego? Można ulec pokusie, by podać tu jakąś arbitralną regułę.

Na przykład moglibyśmy stwierdzić, że „mniejszy" jest płatek mający mniejszą wartość w pierwszej parze odpowiadających sobie elementów opisu, których wartości są różne. Właśnie w taki sposób działa funkcja porównująca przedstawiona na listingu 1.9.

Listing 1.9. Funkcja porównująca na potrzeby sortowania płatków

```
int compare(const void *first, const void *second) {
  int i;
  const int *snowflake1 = first;
  const int *snowflake2 = second;
  if (are_identical(snowflake1, snowflake2))
    return 0;
  for (i = 0; i < 6; i++)
    if (snowflake1[i] < snowflake2[i])
      return -1;
  return 1;
}
```

Niestety, takie posortowanie płatków śniegu nie pomoże nam w rozwiązaniu problemu. Poniżej przedstawiam przykład przypadku testowego, który określa cztery płatki śniegu i którego próba wykonania na Twoim laptopie zapewne zakończy się niepowodzeniem:

```
4
3 4 5 6 1 2
2 3 4 5 6 7
4 5 6 7 8 9
1 2 3 4 5 6
```

Identyczne są płatki pierwszy i czwarty, jednak wykonanie tego przypadku testowego zwróci komunikat informujący, że nie ma identycznych płatków śniegu. Dlaczego tak się dzieje?

Oto dwa fakty, o których podczas wykonywania dowie się funkcja `qsort`:

1. Czwarty płatek jest mniejszy od drugiego.
2. Drugi płatek jest mniejszy od pierwszego.

Na tej podstawie funkcja dojedzie do wniosku, że czwarty płatek jest mniejszy od pierwszego. Co więcej, dojdzie do tego wniosku nawet bez porównywania tych płatków śniegu! Jej działanie bowiem bazuje na tym, że relacja mniejszości jest przechodnia — jeśli *a* jest mniejsze od *b*, a *b* jest mniejsze od *c*, to *a* powinno być mniejsze od *c*. Wychodzi na to, że nasza definicja „większości" i „mniejszości" jednak ma znaczenie.

Niestety, nie jest oczywiste, w jaki sposób należy zdefiniować relacje mniejszości i większości płatków śniegu, by były one przechodnie. Jeśli odczuwasz zawód

z tego powodu, to być może pocieszy Cię wiadomość, że będziemy w stanie rozwiązać ten problem bez uciekania się do sortowania tablicy płatków śniegu.

Ogólnie rzecz biorąc, gromadzenie podobnych wartości przy wykorzystaniu sortowania może być bardzo użyteczną techniką przetwarzania danych. Dodatkową zaletę stanowi to, że dobre algorytmy sortowania są bardzo szybkie — na pewno działają szybciej niż ze złożonością $O(n^2)$; w tym problemie jednak nie będziemy mogli skorzystać z sortowania.

Rozwiązanie 2. Zmniejszenie liczby wykonywanych operacji

Okazało się, że porównywanie wszystkich par płatków śniegu i sortowanie płatków wymaga zbyt wiele pracy. Aby posunąć się w kierunku satysfakcjonującego i, jak się okaże, ostatecznego rozwiązania problemu, spróbujemy skorzystać z pomysłu unikania porównywania tych płatków śniegu, które w oczywisty sposób są od siebie różne. Na przykład, jeśli będziemy dysponować parą płatków:

```
1, 2, 3, 4, 5, 6
```

i

```
82, 100, 3, 1, 2, 999
```

to w żaden sposób nie będą one mogły być identyczne. Nawet nie warto marnować czasu na ich porównywanie.

Liczby w opisie drugiego płatka zbyt znacząco różnią się od liczb w opisie pierwszego. Aby określić sposób wykrywania odmienności dwóch płatków bez konieczności ich bezpośredniego porównywania, możemy zacząć od porównania pierwszych liczb w opisach, gdyż, jak widać w przedstawionym przykładzie, 1 bardzo różni się od 82. A teraz rozważmy takie płatki:

```
3, 1, 2, 999, 82, 100
```

i

```
82, 100, 3, 1, 2, 999
```

Te dwa płatki śniegu *są* identyczne, choć liczby 3 i 82 bardzo się od siebie różnią.

Prostym testem, pomocnym w określeniu, czy dwa płatki śniegu mogą być identyczne, może być *sumowanie* ich elementów. Kiedy zsumujemy wartości naszych dwóch przykładowych płatków śniegu, uzyskamy wartość 21 dla pierwszego

z nich — 1, 2, 3, 4, 5, 6 — i wartość 1187 dla drugiego — 82, 100, 3, 1, 2, 999. Mówimy, że *kod* pierwszego płatka wynosi 21, a drugiego 1187.

Nasze nowe podejście opiera się na założeniu, że będziemy mogli wrzucić płatki „z kodem 21" do jednego kubełka, a płatki „z kodem 1187" do drugiego i nigdy nie będziemy musieli ich ze sobą porównywać. Takie przydzielenie do kubełka będzie można wykonać dla każdego płatka śniegu: będziemy mogli zsumować elementy opisu danego płatka, uzyskać jego kod x, a następnie zapisać ten płatek wraz ze wszystkimi innymi mającymi ten sam kod.

Oczywiście znalezienie dwóch płatków o tym samym kodzie 21 nie gwarantuje wcale, że będą one identyczne. Na przykład dwa płatki, 1, 2, 3, 4, 5, 6 i 16, 1, 1, 1, 1, 1, mają kod 21, ale bez wątpienia nie są identyczne.

W niczym nam to jednak nie przeszkadza, gdyż nasza „reguła sumowania" ma na celu jedynie odrzucenie płatków, które na pewno nie są identyczne. Pozwoli nam ona uniknąć porównywania wszystkich par, a to właśnie było powodem nieefektywności naszego pierwszego rozwiązania, i sprawi, że będziemy musieli porównywać tylko te pary, które nie zostały odrzucone jako w oczywisty sposób różne.

W pierwszym rozwiązaniu przechowywaliśmy wszystkie płatki śniegu jeden za drugim, w zwyczajnej tablicy: pierwszy płatek miał indeks 0, drugi 1 itd. W nowym rozwiązaniu zastosujemy inną strategię przechowywania danych: położenie poszczególnych płatków w tablicy będzie zależeć od wartości ich kodu. A zatem dla każdego płatka obliczymy jego kod i użyjemy go jako indeksu określającego położenie płatka w tablicy.

Aby jednak zastosować takie rozwiązanie, musimy rozwiązać dwa problemy:

1. W jaki sposób obliczyć kod płatka śniegu?
2. Co zrobić, jeśli dwa płatki śniegu będą mieć ten sam kod?

Zacznijmy od rozwiązania problemu z obliczaniem kodu płatków.

Obliczanie kodu płatków śniegu

Na pierwszy rzut oka problem obliczenia kodu płatków może się wydawać prosty. Wystarczy zsumować poszczególne wartości z opisu płatka, jak w poniższym przykładzie:

```
int code(int snowflake[]) {
  return (snowflake[0] + snowflake[1] + snowflake[2]
          + snowflake[3] + snowflake[4] + snowflake[5]);
}
```

Takie rozwiązanie będzie się świetnie sprawdzać w wypadku takich płatków śniegu jak 1, 2, 3, 4, 5, 6 i 82, 100, 3, 1, 2, 999; rozważmy jednak płatek, w którego opisie zostały użyte liczby o bardzo dużych wartościach, taki jak:

```
1000000, 2000000, 3000000, 4000000, 5000000, 6000000
```

Kod takiego płatka ma wartość 21000000. My chcemy użyć tego kodu jako *indeksu* tablicy przechowującej płatki śniegu, ale by to zrobić, musielibyśmy zadeklarować tablicę o 21 000 000 elementów. W sytuacji gdy porównywanych płatków śniegu może być co najwyżej 100 000, deklarowanie tak wielkiej tablicy byłoby koszmarnym marnowaniem pamięci.

W naszym rozwiązaniu będziemy dążyć do zastosowania tablicy o 100 000 elementów. Będziemy obliczać wartość kodu płatka jak wcześniej, a następnie, w jakiś sposób, przekształcimy go do postaci liczby z zakresu od 0 do 99999 (czyli mieszczącej się w zakresie indeksów tablicy). Jednym ze sposobów, by to zrobić, jest skorzystanie z operatora %. Wyznaczenie reszty z dzielenia nieujemnej liczby całkowitej przez liczbę *x* powoduje zwrócenie wartości z zakresu od 0 do *x*–1. Niezależnie od tego, jaka będzie wartość kodu płatka, jeśli podzielimy ją modulo 100 000, to uzyskamy wartość, która będzie stanowić prawidłowy indeks naszej tablicy.

Takie rozwiązanie ma pewną wadę: sprawi ono, że *więcej* płatków śniegu będzie mieć ten sam kod. Na przykład dwa płatki śniegu, 1, 1, 1, 1, 1, 1 i 100001, 1, 1, 1, 1, 1, są różne — ich kody to odpowiednio 6 i 100006 — kiedy jednak podzielimy te kody modulo 100 000, to w obu wypadkach uzyskamy wartość 6. Jest to ryzyko, na które możemy się zgodzić: będziemy jedynie mieć nadzieję, że takich sytuacji nie będzie zbyt dużo, bo gdyby tak się stało, to ponownie wykonywalibyśmy porównywanie wszystkich możliwych par.

A zatem będziemy sumować liczby z opisu płatka śniegu, a następnie dzielić uzyskaną sumę modulo 100 000, jak pokazuję na listingu 1.10.

Listing 1.10. Obliczanie kodu płatka śniegu

```
#define SIZE 100000

int code(int snowflake[]) {
  return (snowflake[0] + snowflake[1] + snowflake[2]
          + snowflake[3] + snowflake[4] + snowflake[5]) % SIZE;
}
```

Kolizje płatków

W naszym pierwszym rozwiązaniu do zapisywania płatka śniegu w tablicy snowflakes w elemencie o indeksie i używaliśmy takiego fragmentu kodu:

```
for (j = 0; j < 6; j++)
  scanf("%d", &snowflakes[i][j]);
```

Rozwiązanie to działało prawidłowo, gdyż w każdym wierszu dwuwymiarowej tablicy był zapisywany dokładnie jeden płatek.

Teraz jednak musimy uwzględniać kolizje pomiędzy płatkami takimi jak 1, 1, 1, 1, 1, 1 i 100001, 1, 1, 1, 1, 1, których kod, ze względu na użycie operatora %,

będzie taki sam, a że jest on używany jako indeks tablicy, oba płatki będziemy musieli zapisać w tym samym elemencie. Oznacza to, że elementy tablicy nie będą już przechowywały pojedynczych płatków śniegu, lecz ich kolekcję, która może zawierać dowolną liczbę płatków, w tym żadnego.

Jednym ze sposobów przechowywania wielu elementów w tym samym miejscu jest zastosowanie *listy połączonej* (ang. *linked list*) — struktury danych, która łączy każdy element z następnym. W naszym wypadku każdy element tablicy płatków śniegu będzie wskazywał na pierwszy płatek listy; pozostałe płatki będzie można odczytać przy użyciu wskaźników `next`.

Zastosujemy typową implementację listy połączonej. Każdy element `snow➥flake_node` będzie zawierać zarówno opis płatka śniegu, jak i wskaźnik do następnego elementu listy. Do zgrupowania tych dwóch komponentów użyjemy struktury. Zastosujemy także słowo kluczowe `typedef`, które pozwoli nam później używać nazwy typu, `snowflake_node`, zamiast pełnej nazwy `struct snowflake_node`:

```
typedef struct snowflake_node {
  int snowflake[6];
  struct snowflake_node *next;
} snowflake_node;
```

Ta zmiana zmusza nas do wprowadzenia kolejnych modyfikacji w dwóch innych funkcjach, `main` i `identify_identical`, które wcześniej operowały na tablicach.

Nowa wersja funkcji main

Zmodyfikowaną wersję funkcji `main` przedstawiam na listingu 1.11.

Listing 1.11. Funkcja main drugiego rozwiązania

```
int main(void) {
  static snowflake_node *snowflakes[SIZE] = {NULL};  ❶
  snowflake_node *snow;  ❷
  int n, i, j, snowflake_code;
  scanf("%d", &n);
  for (i = 0; i < n; i++) {
    snow = malloc(sizeof(snowflake_node));  ❸
    if (snow == NULL) {
      fprintf(stderr, "malloc - błąd przydzielania pamięci\n");
      exit(1);
    }
    for (j = 0; j < 6; j++)
      scanf("%d", &snow->snowflake[j]);  ❹
    snowflake_code = code(snow->snowflake);  ❺
    snow->next = snowflakes[snowflake_code];  ❻
    snowflakes[snowflake_code] = snow;  ❼
  }
  identify_identical(snowflakes);
```

```
    // Jeśli chcesz zapewnić prawidłowość programu, to tu możesz zwolnić pamięć
    // przydzieloną wcześniej przy użyciu funkcji malloc
    return 0;
}
```

Przeanalizujmy dokładnie kod tej funkcji. W pierwszej kolejności zwróć uwagę, że zmieniliśmy typ tablicy z dwuwymiarowej tablicy liczb na jednowymiarową tablicę wskaźników na elementy typu `snowflake_node` ❶. Zadeklarowaliśmy także zmienną `snow` ❷, która będzie wskazywać na węzeł aktualnie pobieranego i przetwarzanego płatka śniegu.

Pamięć dla każdego z węzłów `snowflake_node` przydzielamy przy użyciu funkcji `malloc` ❸. Po wczytaniu i zapisaniu sześciu liczb opisu płatka ❹ obliczamy kod płatka — w tym celu wywołujemy funkcję przedstawioną na listingu 1.10 — i zapisujemy go w zmiennej `snowflake_code` ❺.

Ostatnią operacją jest dodanie płatka śniegu do tablicy `snowflakes`; sprowadza się ona do dodania kolejnego węzła do listy połączonej. W tym celu zapisujemy węzeł płatka jako pierwszy element listy. W pierwszej kolejności w polu `next` dodawanego węzła zapisujemy wskaźnik pierwszego elementu listy ❻, a następnie zapisujemy wskaźnik dodawanego węzła jako początek listy ❼. W tym wypadku kolejność wykonywanych operacji na znaczenie: gdybyśmy odwrócili kolejność, w jakiej są wykonywane te dwie ostatnie operacje, to utracilibyśmy dostęp do bieżących elementów listy!

Zwróć uwagę, że z perspektywy poprawności miejsce listy, w którym dodamy nowy węzeł, nie ma żadnego znaczenia. Równie dobrze możemy go dodać na początku, końcu czy też gdziekolwiek w środku — wybór należy do nas. My jednak wybierzemy najszybsze rozwiązanie, a najszybsze jest dodawanie nowego węzła na początku listy, gdyż w ten sposób nie musimy jej przeglądać. Gdybyśmy chcieli dodawać nowy węzeł na końcu listy, musielibyśmy przejrzeć jej całą zawartość. Gdyby taka lista miała milion elementów, musielibyśmy odczytywać wskaźnik następnego węzła milion razy, zanim udałoby się nam dotrzeć do ostatniego — a to byłoby bardzo wolne!

Przeanalizujmy krótki przykład działania tej nowej wersji funkcji `main`. Poniżej przedstawiam przypadek testowy:

```
4
1 2 3 4 5 6
8 3 9 10 15 4
16 1 1 1 1 1
100016 1 1 1 1 1
```

Każdy element tablicy `snowflakes` ma początkowo wartość `NULL`, czyli jest pustą listą połączoną. Kiedy do tej tablicy zaczniemy dodawać płatki śniegu, jej elementy zaczną wskazywać na struktury `snowflake_node`. Suma liczb z opisu pierwszego płatka śniegu wynosi 21, dlatego też węzeł reprezentujący ten płatek zapiszemy w elemencie tablicy `snowflakes` o indeksie 21. Drugi płatek trafi do elementu

o indeksie 49. Trzeci płatek śniegu także trafi do elementu o indeksie 21. Oznacza to, że po jego dodaniu w elemencie o indeksie 21 będzie zapisana lista połączona zawierająca *dwa* płatki: 16, 1, 1, 1, 1, 1 i 1, 2, 3, 4, 5, 6.

A co z czwartym płatkiem? Także on trafia do elementu tablicy o indeksie 21, co oznacza, że teraz będzie w nim zapisana lista zawierająca trzy płatki śniegu. Czy są to jednak identyczne płatki? Nie! Ten przykład pokazuje, że zapisanie płatków śniegu na jednej liście połączonej nie jest wystarczającym powodem, by uznać, że są one identyczne. Aby dojść do prawidłowego wyniku, musimy porównać wszystkie możliwe pary tych płatków. I to porównanie płatków zapisanych na listach jest ostatecznym elementem rozwiązania naszego problemu.

Nowa wersja funkcji identify_identical

Nasza nowa funkcja `identify_identical` musi porównywać wszystkie pary płatków śniegu na każdej z list połączonych. Jej kod przedstawiam na listingu 1.12.

Listing 1.12. Wykrywanie identycznych płatków śniegu na listach połączonych

```
void identify_identical(snowflake_node *snowflakes[]) {
  snowflake_node *node1, *node2;
  int i;
  for (i = 0; i < SIZE; i++) {
    node1 = snowflakes[i];  ❶
    while (node1 != NULL) {
      node2 = node1->next;  ❷
      while (node2 != NULL) {
        if (are_identical(node1->snowflake, node2->snowflake)) {
          printf("Twin snowflakes found.\n");
          return;
        }
        node2 = node2->next;
      }
      node1 = node1->next;  ❸
    }
  }
  printf("No two snowflakes are alike.\n");
}
```

Zaczynamy od ustawienia zmiennej `node1` na pierwszy element listy ❶. Następnie używamy zmiennej `node2`, by przejść przez wszystkie węzły położone na prawo od `node1` ❷ aż do końca listy. W ten sposób porównujemy pierwszy płatek śniegu zapisany na liście ze wszystkimi pozostałymi. Następnie przesuwamy zmienną `node1` na drugi węzeł listy ❸ i porównujemy drugi płatek śniegu ze wszystkimi umieszczonymi na prawo od niego. Te operacje powtarzamy aż do momentu, gdy zmienna `node1` dotrze do końca listy.

Ta nowa wersja kodu jest niebezpiecznie podobna do funkcji `identify_identical` z pierwszego rozwiązania (patrz listing 1.7), która porównywała wszystkie możliwe pary płatków śniegu. Tym razem jednak sprawdzamy wyłącznie wszystkie

pary w obrębie jednej listy. A co się stanie, jeśli ktoś przygotuje taki przypadek testowy, w którym wszystkie płatki śniegu trafią na jedną listę? Czy wówczas wydajność działania nowego rozwiązania nie byłaby równie zła jak poprzedniego?

Poświęć minutkę na przesłanie tego drugiego rozwiązania na witrynę oceniającą i po prostu się przekonaj. Okaże się, że jest ono znacznie bardziej efektywne! Nowe rozwiązanie bazuje na zastosowaniu struktury danych określanej jako tablica mieszająca (ang. *hash table*). Przyjrzymy się jej dokładniej w następnym podrozdziale.

Tablice mieszające

Tablica mieszająca składa się z dwóch komponentów — są to:

1. Tablica, której poszczególne lokalizacje są nazywane *kubełkami* (ang. *buckets*).
2. *Funkcja mieszająca* (ang. *hash function*), pobierająca obiekt i zwracająca jego kod, który będzie używany jako indeks kubełka w tablicy.

Kod zwracany przez funkcję mieszającą jest nazywany *kodem mieszającym* (ang. *hashcode*); określa on miejsce w tablicy, gdzie zostanie zapisany obiekt, dla którego funkcja mieszająca zwróciła dany kod.

Przyjrzyj się dokładniej kodom przedstawionym na listingach 1.10 i 1.11, a przekonasz się, że zaimplementowaliśmy już oba elementy tablicy mieszającej. Funkcja `code` pobierająca płatek śniegu i zwracająca jego kod (liczbę z zakresu od 0 do 99 999) jest naszą funkcją mieszającą, a tablica `snowflakes` — tablicą kubełków, z których każdy stanowi listę połączoną.

Projekt tablicy mieszającej

Projektowanie tablicy mieszającej wymaga podjęcia szeregu decyzji. Przyjrzyjmy się bliżej trzem z nich.

Pierwsza decyzja dotyczy wielkości. W poprzednim rozwiązaniu użyliśmy arbitralnej wartości 100 000 jako maksymalnej liczby płatków śniegu, które mogą być wczytane i przetworzone przez program (zgodnie zresztą ze specyfikacją problemu). Mogliśmy jednak zastosować tablicę większą albo mniejszą. Użycie mniejszej tablicy pozwoli zaoszczędzić pamięć. Na przykład podczas inicjalizacji w tablicy zawierającej 50 000 wartości `NULL` zostanie zapisanych o połowę mniej elementów niż w tablicy o 100 000 elementach. Z drugiej strony zastosowanie mniejszej tablicy sprawi, że więcej elementów będzie trafiać do tego samego kubełka. Mówimy, że kiedy obiekty trafiają do tego samego kubełka, występują między nimi *kolizje*. Problem dużej liczby kolizji polega na tym, że powoduje on powstawanie długich list połączonych. W idealnej sytuacji wszystkie te listy powinny być krótkie, takie by nie trzeba było ich przeglądać ani wykonywać operacji na umieszczonych na nich elementach. Zastosowanie większej tablicy pozwala uniknąć niektórych spośród takich kolizji.

Podsumowując, musimy wypracować pewien kompromis pomiędzy zużyciem pamięci i czasem działania. Jeśli wielkość tablicy będzie zbyt mała, to liczba kolizji raptownie wzrośnie. Jeśli natomiast tablica będzie zbyt duża, to problemem stanie się zużycie pamięci.

Druga decyzja, jaką należy podjąć, jest związana z funkcją mieszającą. W naszym programie funkcja ta sumuje liczby opisujące płatek śniegu i dzieli tę sumę modulo 100 000. Co ważne, gwarantuje, że jeśli dwa płatki śniegu są identyczne, to znajdą się w tym samym kubełku. (Choć oczywiście w tym samym kubełku mogą się znaleźć płatki śniegu, które nie są identyczne). To właśnie dzięki tej właściwości funkcji mieszającej identycznych płatków śniegu możemy poszukiwać na tej samej liście, a nie na różnych listach.

Przy rozwiązywaniu problemów z wykorzystaniem tablic mieszających funkcja mieszająca musi uwzględniać to, co oznacza, że dwa obiekty są identyczne. Jeśli dwa obiekty są identyczne, to funkcja mieszająca musi umieścić je w tym samym kubełku. A gdy dwa obiekty muszą być dokładnie sobie równe, by można je było uznać za „identyczne", możemy znacznie skomplikować zagadnienie i sprawić, że odwzorowanie obiektów na kubełki będzie zdecydowanie bardziej złożone niż w wypadku rozpatrywanych tu płatków śniegu. W ramach przykładu przeanalizujmy funkcję mieszającą o nazwie `oaat`, przedstawioną na listingu 1.13.

Listing 1.13. Złożona funkcja mieszająca

```
#define hashsize(n) ((unsigned long)1 << (n))
#define hashmask(n) (hashsize(n) - 1)
unsigned long oaat(char *key, unsigned long len,
                   unsigned long bits) {
  unsigned long hash, i;
  for (hash = 0, i = 0; i < len; i++) {
    hash += key[i];
    hash += (hash << 10);
    hash ^= (hash >> 6);
  }
  hash += (hash << 3);
  hash ^= (hash >> 11);
  hash += (hash << 15);
  return hash & hashmask(bits);
}

int main(void) { // Proste wywołanie funkcji oaat
  long snowflake[] = {1, 2, 3, 4, 5, 6};
  //2^17 jest najmniejszą potęgą liczby 2 większą lub równą 100 000
  unsigned long code = oaat((char *)snowflake, sizeof(snowflake), 17);
  printf("%u\n", code);
  return 0;
}
```

W wywołaniu funkcji `oaat` musimy przekazać trzy parametry:

`key` Dane, dla których chcemy obliczyć kod mieszający.

`len` Długość tych danych.

`bits` Długość kodu mieszającego wyrażona jako liczba jego bitów.

Maksymalną dopuszczalną wartość kodu mieszającego możemy wyznaczyć, podnosząc 2 do potęgi `bits`. Na przykład, jeśli wybierzemy liczbę 17, to 2^{17} = 131 072, czyli maksymalną dopuszczalną wartością kodu mieszającego będzie 131 072.

A jak działa funkcja `oaat`? Zaczyna się od pętli, wewnątrz której dodaje do zmiennej wartość bieżącego bajtu klucza. Ta część jest podobna do naszej funkcji mieszającej, która sumowała liczby opisujące płatek śniegu (patrz listing 1.10). Wykonywane następnie operacje przesunięć bitowych w lewo i bitowych alternatyw wykluczających mają na celu wymieszanie wartości klucza. Mają one wywołać *efekt lawiny*, co oznacza, że niewielka zmiana w bitach klucza spowoduje ogromną zmianę w wartości uzyskanego kodu mieszającego. Jeśli celowo nie przygotowujesz odpowiednio spreparowanych danych wejściowych dla tej funkcji lub jeśli nie używasz bardzo wielu kluczy, to jest raczej mało prawdopodobne, by użycie tej funkcji doprowadziło do wystąpienia licznych kolizji. To prowadzi do ważnego spostrzeżenia: dla każdej funkcji mieszającej *zawsze* można wskazać taką kolekcję danych, która doprowadzi do wystąpienia bardzo dużej liczby kolizji, a tym samym do bardzo złej wydajności działania. Wyszukana funkcja mieszająca, taka jak `oaat`, nie jest w stanie nas przed tym ochronić. Jeśli jednak nie zwracamy przesadnej uwagi na złośliwie spreparowane dane wejściowe, to bardzo często z powodzeniem będziemy mogli korzystać ze stosunkowo dobrych funkcji mieszających, takich jak `oaat`, i zakładać, że będą one w stanie prawidłowo rozsiewać dane w całej tablicy mieszającej.

Właśnie z tego powodu nasze drugie rozwiązanie problemu płatków śniegu, to korzystające z tablicy mieszającej, zapewniło tak dobre wyniki. Zastosowaliśmy dobrą funkcję mieszającą, która zapisywała bardzo wiele różnych płatków śniegu w różnych kubełkach. Ponieważ nie zabezpieczamy naszego kodu przed żadnym atakiem, nie musimy się przejmować, że jakaś wrogo nastawiona osoba przeanalizuje nasz kod i określi sposób pozwalający na doprowadzenie do milionów kolizji.

I w końcu, w ramach ostatniej decyzji projektowej, musimy się zastanowić, co ma pełnić funkcję kubełków. W przedstawionym rozwiązaniu każdym kubełkiem była lista połączona. Takie rozwiązanie, bazujące na zastosowaniu listy połączonej, jest nazywane *schematem łańcuchowym* (ang. *chaining scheme*).

W innym rozwiązaniu, określanym jako *adresowanie otwarte* (ang. *open-addressing*), każdy kubełek zawiera co najwyżej jeden element i żadne listy powiązane nie są używane. W tym wypadku rozwiązywanie kolizji polega na przeglądaniu kolejnych kubełków aż do momentu znalezienia pustego. Załóżmy, że chcemy zapisać element w kubełku o numerze 50, ale okazuje się, że jest już pełny. W takim razie będziemy sprawdzać kolejno kubełki o numerach: 51, 52, 53 itd., aż do momentu znalezienia pustego. Niestety, takie proste rozwiązanie może

prowadzić do słabej wydajności działania, zwłaszcza jeśli w tablicy mieszającej jest zapisanych wiele elementów; dlatego w praktyce często stosowane są bardziej wyrafinowane metody poszukiwania pustych kubełków.

Schemat łańcuchowy zazwyczaj można zaimplementować znacznie łatwiej niż adresowanie otwarte i to właśnie z tego powodu zdecydowałem się go użyć w przedstawionym rozwiązaniu. Niemniej adresowanie otwarte ma zalety, takie jak oszczędzanie pamięci, której zużywamy mniej dzięki temu, że nie trzeba rezerwować pamięci na węzły list połączonych.

Dlaczego warto używać tablic mieszających?

Zastosowanie tablicy mieszającej w ogromnym stopniu przyspieszyło działanie naszego rozwiązania problemu płatków śniegu. Na typowym laptopie wykonanie przypadku testowego obejmującego 100 000 płatków śniegu zajmie poniżej 2 sekund! Nie trzeba już porównywać wszystkich możliwych par ani sortować płatków — wystarczy jedynie kilka prostych operacji na listach połączonych. Przy braku patologicznych danych możemy oczekiwać, że na każdej liście będzie się znajdować jedynie kilka elementów. W takim wypadku porównanie wszystkich par w ramach jednej listy będzie wymagało wykonania stałej, niewielkiej liczby operacji. Oznacza to, że oczekujemy, że zastosowanie tablicy mieszającej pozwoli nam stworzyć rozwiązanie o *liniowym* czasie działania — czyli wymagające wykonania n kroków (a nie $n(n–1)/2$, jak było w pierwszym z przedstawionych rozwiązań). Pod względem złożoności takie rozwiązania są określane jako rozwiązania $O(n)$.

Jeśli podczas rozwiązywania problemu zauważymy, że często poszukujemy tego samego elementu, warto się zastanowić nad zastosowaniem tablicy mieszającej. Dzięki niej możemy przekształcić powolne przeszukiwanie w błyskawiczne odwołanie. Może się także zdarzyć, że problem będzie można rozwiązać poprzez sortowanie danych; choć w wypadku problemu „Płatki śniegu" nie było to możliwe. A później, by szybko odnajdywać elementy w posortowanej tablicy, można zastosować technikę wyszukiwania binarnego (opisaną w rozdziale 6.). Nawet jednak posortowanie tablicy i skorzystanie z wyszukiwania binarnego nie jest w stanie konkurować z wydajnością, jaką zapewniają tablice mieszające.

Problem 2. Słowa złożone

Przeanalizujmy inny problem, którego naiwne rozwiązanie będzie bazować na zastosowaniu wolnego przeszukiwania. Następnie ponownie skorzystamy w tablicy mieszającej, by drastycznie to rozwiązanie przyspieszyć. Ten problem będziemy analizować nieco szybciej niż poprzedni, gdyż teraz już wiemy, czego należy szukać.

Problem „Słowa złożone" jest dostępny na witrynie UVa i ma symbol 10391.

Problem

Dysponujemy listą słów, z których każde jest łańcuchem znaków zapisanych małymi literami. Na przykład taka lista może zawierać słowa: `krat`, `kratka`, `owal` i `ka`. Zakładamy, że łańcuchy te nie są zbyt długie. Naszym zadaniem jest znalezienie na tej liście łańcuchów, które są *słowami złożonymi*, czyli stanowią połączenie dokładnie dwóch innych łańcuchów występujących na liście. Na przedstawionej wcześniej przykładowej liście takim słowem złożonym jest jedynie `kratka`, gdyż stanowi połączenie słów `krat` i `ka`.

Dane wejściowe

Poszczególne łańcuchy (słowa) są zapisywane po jednym w wierszu i podawane w kolejności alfabetycznej. Może być co najwyżej 120 000 takich łańcuchów.

Wyniki

Problem wymaga, by każde słowo złożone zostało wyświetlone w osobnym wierszu i by podawać je w kolejności alfabetycznej.

Rozwiązanie przypadku testowego musi zostać podane w ciągu 3 sekund.

Wskazywanie słów złożonych

W jaki sposób możemy rozpoznawać słowa złożone, kiedy już zakończymy wczytanie danych wejściowych? Przyjrzyjmy się przykładowi słowa `kratka`. Słowo to można złożyć z dwóch innych na pięć sposobów:

1. Pierwszym słowem może być `k`, a drugim `ratka`.
2. Pierwszym słowem może być `kr`, a drugim `atka`.
3. Pierwszym słowem może być `kra`, a drugim `tka`.
4. Pierwszym słowem może być `krat`, a drugim `ka`.
5. Pierwszym słowem może być `kratk`, a drugim `a`.

W ramach pierwszej iteracji powinniśmy przejrzeć listę w poszukiwaniu słów `k` i `ratka`. Jeśli oba poszukiwane słowa zostaną odnalezione, będzie to oznaczać, że udało się nam znaleźć słowo złożone. W ramach drugiej iteracji będziemy poszukiwać na liście słów `kr` i `atka`. Takie poszukiwania będziemy kontynuować, aż sprawdzimy wszystkie pięć możliwości; przy czym dotyczy to tylko jednego słowa — `kratka`. Najprawdopodobniej będziemy mieć do sprawdzenia więcej słów, nawet do 120 000. To naprawdę bardzo dużo operacji przeszukiwania, a przeszukiwanie ogromnych list słów jest niezwykle czasochłonne. Na szczęście uda się nam przyspieszyć działanie dzięki zastosowaniu tablicy mieszającej.

Rozwiązanie

Nasze rozwiązanie będzie ponownie korzystać z tablicy mieszającej, której kubełkami będą listy połączone. Oczywiście, zgodnie z oczekiwaniami, będziemy także potrzebować funkcji mieszającej.

W tym wypadku jednak nie zastosujemy funkcji takiej jak w problemie „Płatki śniegu", gdyż doprowadziłoby to do występowania kolizji pomiędzy słowami będącymi anagramami, takimi jak `akt` i `kat`. W odróżnieniu od poprzedniego problemu w tym słowa powinny być rozróżniane na podstawie nie tylko samych liter, lecz także ich położenia w słowie. Oczywiście występowanie kolizji jest nieuniknione, powinniśmy jednak zrobić co tylko w naszej mocy, by zminimalizować ich liczbę. W tym celu zastosujemy tę dziwaczną funkcję `oaat` przedstawioną wcześniej na listingu 1.13.

W rozwiązaniu tego problemu zastosujemy także cztery funkcje pomocnicze.

Wczytanie wiersza tekstu

Zaczniemy od przedstawienia funkcji pomocniczej służącej do wczytania wiersza tekstu (patrz listing 1.14).

Listing 1.14. Funkcja wczytująca wiersz tekstu

```
/* Na podstawie rozwiązania z https://stackoverflow.com/questions/16870485 */
char *read_line(int size) {
  char *str;
  int ch;
  int len = 0;
  str = malloc(size);
  if (str == NULL) {
    fprintf(stderr, "malloc - błąd przydzielania pamięci\n");
    exit(1);
  }
  while ((ch = getchar()) != EOF && (ch != '\n')) {    ❶
    str[len++] = ch;
    if (len == size) {
      size = size * 2;
      str = realloc(str, size);     ❷
      if (str == NULL) {
        fprintf(stderr, "realloc - błąd przydzielania pamięci\n");
        exit(1);
      }
    }
  }
  str[len] = '\0';    ❸
  return str;
}
```

Niestety, specyfikacja problemu nie określa maksymalnej długości wiersza tekstu.

Nie możemy określić na stałe maksymalnej długości słowa, ustawiając ją na przykład na 16 lub 100 znaków, gdyż nie mamy żadnej kontroli nad danymi wejściowymi. Funkcja `read_line` pobiera zatem długość początkową, która według naszych przypuszczeń powinna wystarczyć do wczytania większości wierszy. W wywołaniu funkcji używamy wartości początkowej 16, gdyż pozwala ona na zapisanie większości słów w języku angielskim, z którymi najprawdopodobniej możemy się zetknąć. Możemy używać funkcji `read_line` do wczytania znaków aż do dotarcia do maksymalnej długości tablicy ❶. Jeśli tablica zostanie wypełniona, a słowo jeszcze się nie zakończy, to możemy użyć funkcji `realloc`, by podwoić jej długość ❷ i utworzyć w ten sposób miejsce na dodatkowe znaki. Pamiętajmy także, by na końcu łańcucha `str` widniał znak o wartości `'\0'` ❸, w przeciwnym razie nie byłby to prawidłowy łańcuch!

Przeszukiwanie tablicy mieszającej

Kolejna funkcja, przedstawiona na listingu 1.15, służy do wyszukiwania w tablicy mieszającej konkretnego słowa.

Listing 1.15. Poszukiwanie słowa

```
#define NUM_BITS 17

typedef struct word_node {
  char **word;
  struct word_node *next;
} word_node;

int in_hash_table(word_node *hash_table[], char *find,
                  unsigned find_len) {
  unsigned word_code;
  word_node *wordptr;
  word_code = oaat(find, find_len, NUM_BITS);  ❶
  wordptr = hash_table[word_code];  ❷
  while (wordptr) {
    if ((strlen(*(wordptr->word)) == find_len) &&          ❸
        (strncmp(*(wordptr->word), find, find_len) == 0))
      return 1;
    wordptr = wordptr->next;
  }
  return 0;
}
```

Funkcja `in_hash_table` wymaga przekazania tablicy mieszającej, a także słowa, które chcemy w niej odszukać. Jeśli uda się znaleźć podane słowo, funkcja zwraca 1, w przeciwnym razie, jeżeli słowo nie zostanie odnalezione, funkcja zwraca 0. Trzecim parametrem wywołania funkcji jest `find_len`, określa on liczbę znaków słowa `find`, które składają się na poszukiwane słowo. Ten trzeci parametr będzie nam

potrzebny, gdyż będziemy chcieli uzyskać możliwość poszukiwania początkowego fragmentu łańcucha; bez tego parametru nie wiedzielibyśmy, ile początkowych znaków słowa należy porównywać.

Działanie tej funkcji opiera się na obliczeniu kodu mieszającego słowa ❶ i użyciu tego kodu do odszukania odpowiedniej listy połączonej, która następnie zostanie przeszukana ❷. Tablica mieszająca zawiera wskaźniki na łańcuchy, a nie same łańcuchy — właśnie z tego powodu wyrażenie `*(wordptr->word)` rozpoczyna się od * ❸. (Jak się przekonamy, analizując kod funkcji `main`, tablica mieszająca zawiera wskaźniki na łańcuch, a nie same łańcuchy, aby umożliwić nam unikanie wielokrotnego przechowywania tych samych łańcuchów).

Rozpoznawanie słów złożonych

Teraz jesteśmy już gotowi, by się zająć sprawdzaniem wszelkich możliwych sposobów podziału słowa w celu określenia, czy jest ono słowem złożonym. To zadanie realizuje funkcja przedstawiona na listingu 1.16.

Listing 1.16. Rozpoznawanie słów złożonych

```
void identify_compound_words(char *words[],
                             word_node *hash_table[],
                             int total_words) {
  int i, j;
  unsigned len;
  for (i = 0; i < total_words; i++) {  ❶
    len = strlen(words[i]);
    for (j = 1; j < len; j++) {   ❷
      if (in_hash_table(hash_table, words[i], j) &&   ❸
          in_hash_table(hash_table, &words[i][j], len - j)) {
        printf("%s\n", words[i]);
        break;  ❹
      }
    }
  }
}
```

Funkcja `identify_compound_words` jest dość podobna do funkcji `identify identical` z poprzedniego problemu, polegającego na rozpoznawaniu identycznych płatków śniegu (patrz listing 1.12). Dla każdego słowa ❶ generuje ona wszystkie możliwe sposoby jego podziału na dwa łańcuchy ❷, a następnie próbuje znaleźć w tablicy mieszającej fragment początkowy (czyli fragment słowa przed miejscem podziału) i końcowy (czyli fragment od miejsca podziału do końca słowa). Miejsce podziału określa zmienna `j` ❸. Pierwsze poszukiwanie dotyczy pierwszych `j` znaków słowa `i`. Z kolei drugie wyszukiwanie dotyczy fragmentu słowa `i` zaczynającego się od indeksu `j` (i mającego długość `len – j`). Jeśli oba fragmenty uda się znaleźć, to analizowane słowo będzie słowem złożonym. Zwróć uwagę na

użycie instrukcji `break` ❹, bez której — gdyby słowo można było prawidłowo podzielić na kilka różnych sposobów — zostałoby ono kilkukrotnie wyświetlone.

Być może zaskoczyło Cię użycie w tej funkcji zarówno tablicy mieszającej (`hash_table`), jak i tablicy słów (`words`). Węzły w tablicy mieszającej będą wskazywać na słowa w tablicy `words`, ale dlaczego używamy tu obu tych struktur danych? Dlaczego nie możemy używać tylko tablicy mieszającej? Otóż wynika to z wymogów rozwiązywanego problemu, które nakazują, by odnalezione słowa złożone były wyświetlane w kolejności alfabetycznej! Tablice mieszające nie zachowują żadnego sortowania — rozmieszczają elementy na całym swoim obszarze, zależnie od kodu. Oczywiście moglibyśmy sortować odnalezione słowa złożone w ramach jakiegoś etapu kończącego przetwarzanie, jednak oznaczałoby to ponowne wykonywanie pracy, która już została za nas zrobiona — w końcu słowa są wczytywane w kolejności alfabetycznej. A zatem, przeglądając słowa zgodnie z kolejnością, w jakiej są zapisane w tablicy `words`, zapewniamy odpowiednie posortowanie wyników bez wykonywania jakichkolwiek działań.

Funkcja main

Funkcję `main` programu stanowiącego rozwiązanie tego problemu przedstawiam na listingu 1.17.

Listing 1.17. Funkcja main programu znajdującego słowa złożone

```
int main(void) {
  static char *words[1 << NUM_BITS] = {NULL};  ❶
  static word_node *hash_table[1 << NUM_BITS] = {NULL};  ❷
  int total = 0;
  char *word;
  word_node *wordptr;
  unsigned length, word_code;
  word = read_line(WORD_LENGTH);
  while (*word) {
    words[total] = word;    ❸
    wordptr = malloc(sizeof(word_node));
    if (wordptr == NULL) {
      fprintf(stderr, "malloc - błąd przydzielania pamięci\n");
      exit(1);
    }
    length = strlen(word);
    word_code = oaat(word, length, NUM_BITS);
    wordptr->word = &words[total];
    wordptr->next = hash_table[word_code];   ❹
    hash_table[word_code] = wordptr;         ❺
    word = read_line(WORD_LENGTH);
    total++;
  }
  identify_compound_words(words, hash_table, total);
  return 0;
}
```

Do określenia wielkości tablicy mieszającej i tablicy `words` użyliśmy dziwnie wyglądającego fragmentu kodu o postaci `1 << NUM_BITS` ❶ i ❷. Jak widać na listingu 1.17, stała `NUM_BITS` ma wartość 17; a zatem zapis `1 << NUM_BITS` jest skrótowym sposobem obliczenia wartości 2^{17}, która wynosi 131 072. Jest to najmniejsza potęga dwójki, której wartość przekracza 120 000 (maksymalną liczbę słów, które mogą zostać wczytane). Funkcja mieszająca `oaat` wymaga, by długość tablicy mieszającej była potęgą 2, dlatego użyliśmy wartości 2^{17} jako wielkości tablicy mieszającej i tablicy `words`.

Po zadeklarowaniu struktur danych możemy już zacząć używać przygotowanych wcześniej funkcji pomocniczych do wypełnienia tych struktur. Każde słowo zapisujemy w tablicy `words` ❸, a wskaźnik na to słowo zapisujemy w tablicy mieszającej ❹ i ❺. Do dodawania wskaźników do tablicy mieszającej zastosowaliśmy taką samą technikę jak w poprzednim problemie, tym z płatkami śniegu: każdy kubełek jest listą połączoną, a każdy nowy wskaźnik jest dodawany na początku listy, przed wskaźnikami już na niej zapisanymi. Po wczytaniu słów wywołujemy funkcję `identify_compound_words`, by wygenerować wyniki.

Podsumowując, nasze rozwiązanie używa tablicy mieszającej i tablicy `words`, by utworzyć błyskawicznie działającą implementację: tablica mieszająca zapewnia szybkie wyszukiwanie, z kolei tablica `words` pozwala przetwarzać słowa w kolejności alfabetycznej. Zastosowanie naiwnego rozwiązania bez tablicy mieszającej byłoby znacznie wolniejsze. Przyjrzyj się jeszcze raz kodowi z listingu 1.16 i załóż, że dysponujemy n słowami. W wypadku zastosowania tablicy mieszającej oczekujemy, że każde wyszukiwanie ❸ będzie wymagać wykonania niewielkiej, stałej liczby kroków. Gdybyśmy jednak nie używali tablicy mieszającej, to każde takie wyszukiwanie wymagałoby przeglądnięcia całej tablicy `words`, czyli wykonania n kroków! Podobnie jak przy problemie „Płatki śniegu", także tutaj zastosowanie tablicy mieszającej pozwoliło poprawić wydajność algorytmu z $O(n^2)$ do $O(n)$.

Problem 3. Sprawdzanie pisowni — usuwanie litery

Czasami można odnieść wrażenie, że problemy są rozwiązywanie w określony sposób, gdyż przypominają inne problemy. W tym podrozdziale przedstawiam problem, który sprawia wrażenie, jakby można go było rozwiązać przy użyciu tablicy mieszającej. Po dokładniejszej analizie jednak się okazuje, że tablica mieszająca jedynie utrudnia to, co faktycznie należy zrobić.

Problem „Sprawdzanie pisowni" jest dostępny na witrynie Codeforces i ma symbol `39J`. (Zapewne najłatwiejszym sposobem, by do niego dotrzeć, jest wpisanie w wyszukiwarce Google hasła: *Codeforces 39J*).

Problem

W tym problemie otrzymujemy dwa łańcuchy znaków, przy czym pierwszy z nich jest o jeden znak dłuższy od drugiego. Naszym zadaniem jest określenie, na ile sposobów można usunąć z pierwszego łańcucha jeden znak, by uzyskać drugi łańcuch. Na przykład istnieje tylko jeden sposób, by z łańcucha `barokowy` uzyskać łańcuch `barkowy`: należy usunąć pierwsze wystąpienie litery `o`. Z kolei łańcuch `abcdxxxef` można przekształcić na `abcdxxef` na trzy sposoby: wystarczy usunąć którąkolwiek z liter `x`.

Kontekstem tego problemu jest programowa kontrola pisowni. Pierwszym słowem mogłoby być `obliczze` (słowo zapisane błędnie), a drugim `oblicze` (słowo zapisane prawidłowo). W tym wypadku błąd można poprawić na dwa sposoby: usuwając jedną z dwóch liter `z` z pierwszego słowa. Problem ma jednak bardziej ogólny charakter i nie ma nic wspólnego z faktycznym zapisem słów w danym języku czy też z literówkami trafiającymi się podczas wpisywania tekstu.

Limit czasu na podanie prawidłowej odpowiedzi na przypadek testowy wynosi 2 sekundy.

Dane wejściowe

Dane wejściowe składają się z dwóch wierszy: w pierwszym z nich zostaje podany pierwszy łańcuch, a w drugim — drugi łańcuch. Każdy z łańcuchów może mieć do 1 000 000 znaków.

Wyniki

Jeśli nie ma sposobu, by przekształcić pierwszy łańcuch na drugi poprzez usunięcie z pierwszego łańcucha jednego znaku, program ma wyświetlić `0`. W przeciwnym wypadku wyniki mają się składać z dwóch wierszy:

- W pierwszym wierszu należy podać liczbę określającą, na ile sposobów można usunąć znak z pierwszego łańcucha, by uzyskać drugi.
- W drugim wierszu należy podać listę indeksów znaków pierwszego łańcucha, które można usunąć, by uzyskać drugi łańcuch. Poszczególne indeksy mają być rozdzielone znakami odstępu. Problem wymaga, by indeksy były liczone począwszy od `1`, a nie od `0`.

  Na przykład dla takich danych wejściowych:

```
abcdxxxef
abcdxxef
```

rozwiązanie ma zwrócić wyniki:

```
3
5 6 7
```

Liczby: 5, 6, i 7 są indeksami trzech liter x z pierwszego łańcucha, przy czym są one liczone od 1 (a nie od 0).

Rozważania o zastosowaniu tablic mieszających

Poświęciłem naprawdę zawstydzająco dużo czasu na poszukiwanie problemów, które nadawałyby się do opublikowania w tej książce. Musiały to być problemy, które pozwalałyby mi nauczyć Cię czegoś na temat określonych struktur danych i algorytmów. Chciałem, by ich rozwiązania były złożone pod względem algorytmicznym, ale same problemy musiały być na tyle proste, byśmy można było zrozumieć zarówno to, co należy zrobić, jak i szczegóły rozwiązania. Naprawdę byłem przekonany, że udało mi się znaleźć do tego podrozdziału właśnie taki problem dotyczący tablic mieszających, jakiego potrzebowałem, a potem… przystąpiłem do jego rozwiązywania.

W problemie 2., dotyczącym odnajdywania słów złożonych, danymi wejściowymi była lista słów. To było bardzo wygodne, gdyż wystarczyło wstawiać poszczególne słowa z listy do tablicy mieszającej, a następnie użyć jej do wyszukiwania początkowego i końcowego fragmentu łańcucha każdego z analizowanych słów. W tym problemie jednak nie otrzymujemy na wejściu listy słów. Pomimo to, kiedy po raz pierwszy zabrałem się do jego rozwiązywania, zacząłem od utworzenia tablicy mieszającej, po czym umieszczałem w niej wszystkie początkowe i końcowe fragmenty drugiego (czyli krótszego) łańcucha. Na przykład dla słowa `abc` wstawiłbym łańcuch `a`, `ab` i `abc` (jako fragmenty początkowe), a także `c` i `bc` (jako fragmenty końcowe). Łańcuch `abc` też mógłby być fragmentem końcowym, byłby już jednak wstawiony do tablicy. Skoro dysponowałem tak przygotowaną tablicą, mogłem przystąpić do analizy każdego znaku pierwszego łańcucha. Usunięcie znaku z łańcucha odpowiada podzieleniu go na dwie części: początkową i końcową. W ten sposób wracamy do problemu 2., wyszukiwania słów złożonych: wystarczyłoby sprawdzić, czy zarówno początkowy, jak i końcowy fragment łańcucha jest dostępny w tablicy mieszającej. Gdyby się okazało, że tak, oznaczałoby to, że usunięcie danej litery z pierwszego łańcucha pozwala przekształcić go w drugi.

Zastosowanie takiej techniki jest całkiem kuszące, prawda? Chcesz ją wypróbować? Możesz do tego celu użyć nawet fragmentów rozwiązania problemu polegającego na znajdowaniu słów złożonych!

Kiedy tworzyłem to rozwiązanie, zapomniałem tylko o jednym: każdy z łańcuchów może mieć do 1 000 000 znaków długości. Oczywiste jest, że nie jesteśmy w stanie zapisać w tablicy mieszającej wszystkich możliwych fragmentów początkowych i końcowych — wymagałoby to zbyt dużo pamięci. Próbowałem rozwiązać ten problem, używając w tablicy mieszającej wskaźników do początku i końca początkowego, a także końcowego fragmentu łańcucha. I choć faktycznie umożliwiło to wyeliminowanie problemu zużycia pamięci, to nie wyeliminowało konieczności porównywania tych niesłychanie długich łańcuchów podczas przeszukiwania tablicy mieszającej. W dwóch poprzednich problemach, z płatkami śniegu i wyszukiwaniem słów złożonych, elementy tablicy mieszającej były małe:

w wypadku płatków śniegu było to sześć liczb całkowitych, a przy słowach złożonych — kilkanaście znaków. To tyle co nic. Jednak w tym problemie sytuacja jest zgoła inna: musimy operować na łańcuchach składających się nawet z 1 000 000 znaków! Porównywanie takich łańcuchów jest niezwykle czasochłonne.

Kolejną czasochłonną operacją wykonywaną w tym rozwiązaniu jest obliczanie kodu mieszającego początkowych i końcowych fragmentów łańcuchów. Moglibyśmy wywołać funkcję `oaat` dla łańcucha mającego 900 000 znaków długości, a następnie dla łańcucha składającego się z jednego dodatkowego znaku. To by jednak oznaczało ponowne wykonanie całej pracy z pierwszego wywołania funkcji `oaat`, choć jedynym, na czym by nam zależało, byłoby obliczenie kodu mieszającego łańcucha, do którego dodano tylko jeden znak.

Niemniej uparcie chciałem zastosować to rozwiązanie. Cały czas chodziło mi po głowie, że tablica mieszająca jest właśnie tym, czego należy tu użyć, i zrezygnowałem z analizowania alternatywnych rozwiązań. W tym momencie najprawdopodobniej powinienem podejść do problemu z zupełnie innej, nowej strony. Zamiast tego dowiedziałem się o *inkrementalnych funkcjach mieszających* (ang. *incremental hash functions*), czyli funkcjach szybko generujących kody mieszające elementów bardzo podobnych do innych, dla których kod mieszający już został określony. Na przykład, gdybym już dysponował kodem mieszającym łańcucha `abcde`, to dzięki zastosowaniu inkrementalnej funkcji mieszającej obliczenie kodu mieszającego łańcucha `abcdef` mógłbym zrobić błyskawicznie, gdyż można przy tym skorzystać z pracy wykonanej przy okazji przetwarzania łańcucha `abcde`, a nie zaczynać wszystko od początku.

Kolejnym moim spostrzeżeniem było to, że jeśli porównywanie superdługich łańcuchów jest zbyt kosztowne, to w ogóle nie należy ich porównywać. Moglibyśmy mieć nadzieję, że zastosowana funkcja mieszająca jest wystarczająco dobra i że będziemy mieć dużo szczęścia z przypadkami testowymi, że nie wystąpią żadne kolizje. Gdybyśmy szukali jakiegoś elementu w tablicy mieszającej i go znaleźli, to… cóż, moglibyśmy mieć nadzieję, że to faktycznie jest łańcuch, o który nam chodziło, a nie jedynie błędnie pozytywne dopasowanie. Gdybyśmy byli skłonni pójść na takie ustępstwo, moglibyśmy zastosować strukturę danych znacznie prostszą od tablic mieszających, których używaliśmy w poprzednich prezentowanych rozwiązaniach. W tablicy `prefix1` każdy indeks `i` zawiera kod mieszający dla początkowego fragmentu pierwszego łańcucha o długości `i`. W taki sam sposób moglibyśmy przygotować trzy kolejne tablice, zawierające kody mieszające odpowiednio dla fragmentów końcowych pierwszego łańcucha, fragmentów początkowych drugiego łańcucha i fragmentów końcowych drugiego łańcucha.

Poniższy fragment kodu pokazuje, w jaki sposób można by określać zawartość tablicy `prefix1`:

```
// long long to w języku C99 typ liczb całkowitych o bardzo dużym zakresie
unsigned long long prefix1[1000001];
prefix1[0] = 0;
for (i = 1; i <= strlen(first_string); i++)
  prefix1[i] = prefix1[i-1] * 39 + first_string[i];
```

Pozostałe trzy tablice można by przygotować w podobny sposób.

W tym rozwiązaniu bardzo duże znaczenie ma zastosowanie liczb bez znaku. W języku C działanie przepełnienia jest precyzyjnie zdefiniowane właśnie w sytuacji posługiwania się liczbami bez znaku, a nie liczbami ze znakiem. Jeśli łańcuch będzie dostatecznie długi, to na pewno wystąpi przepełnienie, dlatego nie chcemy dopuścić do wystąpienia niezdefiniowanego zachowania.

Zwróć uwagę, jak łatwo jest obliczyć kod mieszający dla elementu `prefix1[i]` na podstawie kodu mieszającego poprzedniego elementu (`prefix1[i–1]`): cała operacja sprowadza się do pomnożenia wartości i dodania do wyniku nowego znaku. Dlaczego mnożymy poprzedni kod mieszający przez 39 i dodajemy znak? Dlaczego nie zastosowałem jakiejś innej funkcji mieszającej? Szczerze mówiąc, dlatego, że w przypadkach testowych stosowanych na witrynie Codeforces zastosowane rozwiązanie nie powodowało występowania żadnych kolizji. Owszem, zdaję sobie sprawę, że takie wytłumaczenie nie jest satysfakcjonujące.

Ale nie przejmuj się! I tak istnieje lepsze rozwiązanie postawionego problemu. Aby do niego dotrzeć, zamiast od razu silić się na zastosowanie tablicy mieszającej, przyjrzymy się naszemu problemowi nieco dokładniej.

Rozwiązanie doraźne

Przeanalizujmy dokładniej przedstawiony wcześniej przykład:

```
abcdxxxef
abcdxxef
```

Załóżmy, że z pierwszego łańcucha usuniemy literę `f` (o indeksie 9). Czy to sprawi, że pierwszy łańcuch stanie się identyczny z drugim? Nie. Dlatego `9` nie znajdzie się na wynikowej liście indeksów. Oba łańcuchy mają długą wspólną część początkową, konkretnie rzecz biorąc — obejmuje ona sześć początkowych znaków: `abcdxx`. Od tego miejsca oba łańcuchy się różnią: w pierwszym z nich występuje litera `x`, a w drugim litera `e`. Jeśli nie wyeliminujemy tej rozbieżności, nie ma szans, by oba łańcuchy były identyczne. Litera `f` znajduje się zbyt daleko na prawo, by jej usunięcie doprowadziło do równości łańcuchów.

To prowadzi do naszej pierwszej obserwacji: jeśli przyjmiemy, że p jest długością *najdłuższego wspólnego początkowego fragmentu* łańcuchów (w naszym przykładzie wartością tą jest 6, czyli długość łańcucha `abcdxx`), to usuwane mogą być jedynie znaki o indeksach mniejszych od $p+1$ lub równych $p+1$. W naszym przykładzie możemy zatem rozważać usunięcie liter: `a`, `b`, `c`, `d`, a także pierwszej, drugiej i trzeciej litery `x`. Usunięcie jakiegokolwiek znaku o indeksie większym od $p+1$ nie spowoduje poprawienia znaku powodującego rozbieżności, umieszczonego na pozycji o indeksie $p+1$, a co za tym idzie — nie może doprowadzić do równości łańcuchów.

Zwróć również uwagę, że jedynie niektóre z potencjalnie możliwych operacji usunięcia zapewnią oczekiwany efekt. Na przykład usunięcie z pierwszego łańcucha liter: `a`, `b`, `c` i `d` nie spowoduje, że stanie się on identyczny z drugim. Ten

efekt zapewni nam jedynie usunięcie dowolnej z trzech liter x. A zatem, oprócz górnej granicy możliwych indeksów ($p+1$), istnieje także dolna granica ich zakresu.

Rozważając kwestię tej dolnej granicy indeksów, zastanówmy się, co by spowodowało usunięcie litery a z pierwszego łańcucha. Czy doprowadzi ono do uzyskania dwóch identycznych łańcuchów? Nie. Powód jest podobny do tego opisanego w poprzednim akapicie: w obu łańcuchach na prawo od a znajdują się różne znaki, dlatego usunięcie a nie może sprawić, że oba łańcuchy staną się takie same. Jeśli zatem długość *najdłuższego wspólnego końcowego fragmentu* obu łańcuchów (w naszym wypadku jest to 4 — długość łańcucha xxef) oznaczymy jako s, to będą nas interesowały jedynie indeksy o wartościach większych od $n–s$ lub równych $n–s$, gdzie n jest długością pierwszego łańcucha. W naszym przykładzie oznacza to, że należy uwzględnić wyłącznie indeksy większe od 5 lub równe 5. W poprzednim akapicie ustaliliśmy, że należy uwzględniać wyłącznie indeksy mniejsze od 7 lub równe 7. Po połączeniu ze sobą tych dwóch warunków dochodzimy do wniosku, że indeksami liter, które można usunąć z pierwszego łańcucha, by stał się on identyczny z drugim, są 5, 6 i 7.

Podsumowując, wynikowe indeksy należą do zakresu od $n–s$ do $p+1$. Dla każdego indeksu z tego zakresu wiemy, że do indeksu $p+1$ oba łańcuchy są identyczne. Wiemy także, że oba łańcuchy są identyczne od indeksu $n–s$. Dlatego, jeśli usuniemy dowolny znak o indeksie z tego zakresu, uzyskamy identyczne łańcuchy. Jeżeli się okaże, że ten zakres jest pusty, będzie to oznaczało, że *nie ma* takiego znaku, którego usunięcie z pierwszego łańcucha doprowadzi do identyczności obu łańcuchów. W takim wypadku program powinien wyświetlić 0. W przeciwnym razie, jeśli zakres indeksów nie jest pusty, możemy użyć pętli for i funkcji printf, by wyświetlić te indeksy na liście. A teraz przekonajmy się, jak to wszystko można zaimplementować!

Najdłuższy wspólny fragment początkowy

Na listingu 1.18 przedstawiam funkcję pomocniczą obliczającą długość najdłuższego wspólnego początkowego fragmentu dwóch łańcuchów znaków.

Listing 1.18. Funkcja obliczająca długość najdłuższego wspólnego początkowego fragmentu łańcuchów

```
int prefix_length(char s1[], char s2[]) {
  int i = 1;
  while (s1[i] == s2[i])
    i++;
  return i - 1;
}
```

Parametr s1 reprezentuje pierwszy łańcuch znaków, a s2 drugi. Początkowym indeksem, od którego zaczynamy porównywanie łańcuchów, jest 1. Począwszy od niego porównujemy kolejne znaki łańcuchów w pętli tak długo, jak są one równe. (W wypadku porównywania łańcuchów takich jak abcde i abcd test

prawidłowo wykryje, że litera e i znak null kończący drugi łańcuch są różne, więc na końcu funkcji zmienna i będzie mieć prawidłową wartość 5).

Najdłuższy wspólny fragment końcowy

Do obliczenia długości najdłuższego wspólnego końcowego fragmentu dwóch łańcuchów użyjemy funkcji przedstawionej na listingu 1.19.

Listing 1.19. Funkcja obliczająca długość najdłuższego wspólnego końcowego fragmentu łańcuchów

```
int suffix_length(char s1[], char s2[], int len) {
  int i = len;
  while (i >= 2 && s1[i] == s2[i-1])
    i--;
  return len - i;
}
```

Kod funkcji `suffix_length` jest bardzo podobny do kodu z listingu 1.18. Jednak tym razem porównujemy łańcuchy od prawej do lewej, a nie od lewej do prawej. Z tego powodu do funkcji trzeba przekazać parametr `len`, który określa długość pierwszego łańcucha znaków. Ostatnią wartością indeksu `i`, dla której możemy wykonać porównanie, jest 2. Gdyby zmienna `i` przyjęła wartość 1, to porównanie odwoływałoby się do elementu `s2[0]`, który nie jest prawidłowym elementem łańcucha!

Funkcja main

Funkcję `main` rozwiązania problemu 3. przedstawiam na listingu 1.20.

Listing 1.20. Kod funkcji main

```
#define SIZE 1000000

int main(void) {
  static char s1[SIZE + 2], s2[SIZE + 2];   ❶
  int len, prefix, suffix, total;
  gets(&s1[1]);   ❷
  gets(&s2[1]);   ❸

  len = strlen(&s1[1]);
  prefix = prefix_length(s1, s2);
  suffix = suffix_length(s1, s2, len);
  total = (prefix + 1) - (len - suffix) + 1;   ❹
  if (total < 0)   ❺
    total = 0;     ❻

  printf("%d\n", total);   ❼
  for (int i = 0; i < total; i++) {   ❽
```

```
    printf("%d", i + len - suffix);
    if (i < total - 1)
      printf(" ");
    else
      printf("\n");
  }
  return 0;
}
```

Wielkość dwóch tablic znakowych używanych do przechowywania łańcuchów wejściowych określamy jako `SIZE + 2` ❶. Maksymalną liczbą znaków, jakie musimy odczytać, jest 1 000 000, ale potrzebujemy także dodatkowego miejsca na znak null, którym w języku C należy zakańczać łańcuchy. Jeszcze jeden dodatkowy element jest nam potrzebny dlatego, że łańcuchy indeksujemy od 1, a nie od 0, czyli pierwszy element tablic, o indeksie `0`, jest tracony.

Następnie wczytujemy pierwszy ❷ i drugi ❸ łańcuch znaków. Zwróć uwagę, że do funkcji `gets` przekazujemy wskaźnik na element tablicy o indeksie `1`, zatem funkcja ta zacznie zapisywać znaki w tablicy właśnie od niego, a nie od indeksu `0`. Po wczytaniu danych wejściowych wywołujemy nasze dwie funkcje pomocnicze, po czym obliczamy liczbę indeksów, które można usunąć z łańcucha `s1`, by przekształcić go w łańcuch `s2` ❹. Jeśli ta liczba jest ujemna ❺, to ustawiamy ją na `0` ❻. Dzięki temu wywołanie `printf` ❼ wyświetli prawidłowy wynik. Następnie używamy pętli `for` ❽, by wyświetlić indeksy znaków, które można usunąć. Wyświetlanie zaczynamy od indeksu `len – suffix`, dlatego, wyświetlając kolejne wartości `i`, dodajemy do niej wartość wyrażenia `len – suffix`.

W ten sposób udało się nam opracować rozwiązanie o liniowym czasie wykonania, pozbawione jakiegokolwiek skomplikowanego kodu i tablic mieszających. Jak widać, zanim rozważymy zastosowanie tablicy mieszającej, warto zadać sobie pytanie, czy problem nie sprawia, że jej użycie będzie niewygodne. Czy wyszukiwanie naprawdę jest konieczne albo czy jakiekolwiek cechy problemu sprawiają, że takie wyszukiwanie w ogóle nie będzie potrzebne.

Podsumowanie

Tablica mieszająca jest strukturą danych: sposobem ich organizacji, który sprawia, że niektóre operacje na danych można wykonywać bardzo szybko. Tablice mieszające usprawniają wyszukiwanie konkretnych elementów. Aby przyspieszać inne operacje, będziemy potrzebować innych struktur danych. Na przykład z rozdziału 7. dowiesz się, czym jest *stóg* (ang. *heap*) — struktura danych, której można używać w celu szybkiego określania minimalnego i maksymalnego elementu tablicy.

Struktury danych są ogólnymi sposobami przechowywania i przetwarzania danych. Przykłady przedstawione w tym rozdziale powinny Ci zapewnić dobrą orientację, kiedy warto używać tablic mieszających, gdyż znajdują one zastosowane

w bardzo wielu problemach, daleko wykraczających poza to, co tu pokazałem. Warto rozważać ich zastosowanie w tych problemach, w których rozwiązania efektywne pod wszelkimi innymi względami są spowalniane przez wielokrotnie realizowane, powolne operacje wyszukiwania.

Uwagi

Problem „Płatki śniegu" pojawił się po raz pierwszy w 2007 roku na Canadian Computing Olympiad.

Problem „Słowa złożone" pojawił się po raz pierwszy we wrześniu 1996 roku na konkursie Waterloo Local Contest.

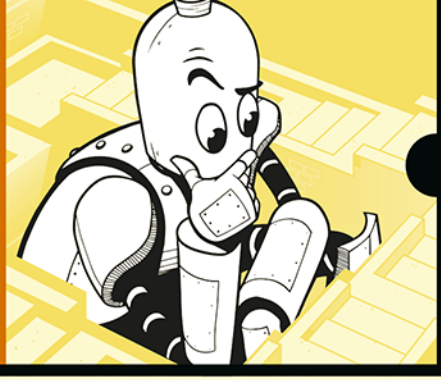
Problem „Sprawdzanie pisowni" przedstawiono po raz pierwszy w 2010 roku na konkursie School Team Contest #1, prowadzonym przez witrynę Codeforces. Rozwiązanie bazujące na wyznaczeniu początkowej i końcowej części wspólnej (co zastosowałem, gdy zrezygnowałem z rozwiązania opartego na tablicach mieszających) pochodzi z notatek opublikowanych na stronie *https://codeforces.com/blog/entry/786*.

Funkcja mieszająca `oaat` (jej nazwa pochodzi od angielskich słów: *one at a time*, co można przetłumaczyć jako: po jednym na raz) została opracowana przez Boba Jenkinsa (patrz *http://burtleburtle.net/bob/hash/doobs.html*).

Dodatkowe informacje na temat zastosowań i implementacji tablic mieszających można znaleźć w książce *Algorithms Illuminated (Part 2): Graph Algorithms and Data Structures* Tima Roughgardena, wydanej w 2018 roku.


PROGRAM
PARTNERSKI
GRUPY HELION
1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ
Zmień swoją stronę WWW w działający bankomat!
Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl
GRUPA
Helion

ALGORYTMY: ZMIERZYSZ SIĘ Z NAPRAWDĘ TRUDNYMI PROBLEMAMI!



Jak już wiesz, struktura danych jest sposobem ich zorganizowania w pamięci komputera, co ma umożliwić szybkie wykonywanie zamierzonych operacji. Pamiętasz też, że algorytm jest sekwencją działań pozwalających na rozwiązanie problemu. Często warunkiem poprawnego działania algorytmu i pomyślnego rozwiązania problemu programistycznego jest trafny wybór struktury danych. To bardzo ważne zagadnienie. Nawet jeśli dobrze znasz wybrany język programowania, to aby pisać dobry kod, musisz nabrać biegłości w posługiwaniu się algorytmami i strukturami danych.

Dzięki tej książce nauczysz się rozwiązywać ambitne problemy algorytmiczne i projektować własne algorytmy. Materiałem do ćwiczeń są tu przykłady zaczerpnięte z konkursów programistycznych o światowej renomie. Dowiesz się, jak klasyfikować problemy, czym się kierować podczas wybierania struktury danych i jak dobierać odpowiednie algorytmy. Sprawdzisz także, w jaki sposób wybór struktury danych może wpłynąć na czas wykonywania algorytmów. Nauczysz się też używać takich metod jak rekurencja, programowanie dynamiczne czy wyszukiwanie binarne. Swoich sił spróbujesz w ramach samodzielnej pracy nad modyfikacją poszczególnych algorytmów. Zamieszczone tu szczegółowe analizy kodu pomogą Ci w zrozumieniu praktycznych aspektów stosowania algorytmów i struktur danych. To znakomity przewodnik na drodze, którą musi pokonać początkujący, aby stać się profesjonalnym programistą Pythona.

W książce między innymi:

- algorytm przeszukiwania wszerz
- algorytm Dijkstry
- struktura zbiorów rozłącznych
- kopce
- tablice mieszające

Dr Daniel Zingaro jest wielokrotnie nagradzanym wykładowcą Uniwersytetu Toronto. Głównym obszarem jego badań naukowych jest metodyka nauczania informatyki i sposób przyswajania tej dziedziny wiedzy. Słynie z niekonwencjonalnego i innowacyjnego podejścia do pracy ze studentami.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-283-8335-7
9 788328 383357
Cena: 89.00 zł
no starch press

INFORMATYKA W NAJLEPSZYM WYDANIU