Palindrom reloaded

#prawo_de_Morgana
#kod_ASCII
#UTF-8

Wrócimy znów do sprawy palindromów (zobacz podrozdział *Od przodu i od tyłu*), bogatsi o pewne umiejętności i wiedzę. Napiszemy program, który sprawdza, czy wprowadzony ciąg znaków (wyraz) jest palindromem, ale posłużymy się tym razem typem logicznym i własnoręcznie napisaną funkcją. Przy okazji poznamy (przypomnimy sobie?) jedno z najważniejszych praw logicznych używanych w informatyce: prawo de Morgana.

Poprzednio przy sprawdzaniu, czy dany wyraz jest palindromem, tworzyliśmy odwrócony ciąg znaków i porównywaliśmy go z oryginalnym ciągiem. Można to jednak zrobić bardziej elegancko i efektywnie – poprzez porównywanie odpowiednich par znaków.

Pamiętamy, że ciąg znaków to tabelka, której rubryki (komórki) to kolejne znaki ciągu. Najpierw porównujemy ze sobą pierwszy i ostatni znak. Jeśli są równe, wtedy porównujemy drugi znak z przedostatnim. Jeśli i te są równe, porównujemy trzeci znak z trzecim od końca i tak dalej, aż dojdziemy do środka ciągu (tę ostatnią sytuację sprecyzujemy za chwilę). Jeśli po drodze napotkaliśmy choć jedną niezgodność, wtedy od razu możemy zawyrokować, że dany ciąg znaków nie jest palindromem i możemy zaniechać dalszego sprawdzania. Jeśli natomiast sprawdzenie wszystkich par dojdzie do końca (czyli do środka ciągu) i nie napotkamy po drodze żadnej niezgodnej pary, wówczas możemy stwierdzić, że dany wyraz jest palindromem.

Reasumując, aby odpowiedzieć NIE, wystarczy znaleźć jedną niezgodność, natomiast by odpowiedzieć TAK, należy stwierdzić zgodność wszystkich par. To nic innego, jak pewna forma fundamentalnego prawa logicznego, sformułowanego w XIX wieku przez angielskiego matematyka i logika, Augustusa de Morgana.*

Znaki tworzące ciąg znaków t mają numery $0, 1, 2, \ldots, d-1$, gdzie d oznacza długość ciągu t, czyli rezultat funkcji `t.length()`. Sprawdzanie par najłatwiej przeprowadzić posługując się dwoma zmiennymi (indeksami) oznaczającymi numery znaków: na przykład i oraz j, gdzie pierwsza z nich oznacza znak z początku ciągu, a druga – znak z końca ciągu.

Tak więc powyżej opisane zmienne powinny zostać zadeklarowane i zainicjalizowane w następujący sposób:

```
int i{ }, j{ t.length()-1 };
```

Na początku i otrzymuje wartość 0 (numer początkowego znaku), zaś j powinno uzyskać wartość `t.length() - 1`, czyli numer ostatniego znaku. Następnie w pętli sprawdzamy, czy $t[i]$ jest

*Można spotkać różne sformułowania tego prawa/twierdzenia: zaprzeczenie koniunkcji/alternatywy lub zaprzeczenie kwantyfikatora.

równe $t[j]$. Jeżeli tak, to idziemy dalej, czyli zwiększamy i oraz zmniejszamy j. Jeśli jednak coś *nie teges*, to *bye bye Jimi*, nie ma szans na palindrom.

Jak długo powinna kręcić się taka pętla? Nasuwa się pomysł, że sygnałem do zakończenia powinno być zrównanie się zmiennych i oraz j. Ma to sens? Tylko wtedy, gdy długość ciągu t jest nieparzysta, wtedy w pewnym momencie wartości obydwu zmiennych zrównają się. Dla ciągu o parzystej ilości znaków zmienne te nie zrównają się, a miną się w okolicy środka ciągu (dlaczego?). Zatem najlepszy warunek kontynuacji tej pętli to „`i < j`". Wtedy co prawda nie „dotkniemy" środkowego znaku wyrazu, ale *who cares?*, ten znak nie ma żadnego wpływu na wynik sprawdzania.

Tak powinien wyglądać kompletny program, który czyta wyraz (zapisany małymi literami alfabetu łacińskiego) i wypisuje TAK/NIE w zależności od tego, czy jest to palindrom (sprawdzenie dokonuje się w funkcji `pal()`):

```
#include <bits/stdc++.h>
using namespace std;

bool pal(string t)
{
  int i{ }, j{ t.length()-1 };
  while(i < j)
    if(t[i] != t[j])
      return false;
    else
    { i++; j--; }
  return true;
}

int main()
{
  string s;
  cin >> s;
  if(pal(s))
    cout << "TAK\n";
  else
    cout << "NIE\n";
  return 0;
}
```

Zauważmy, że instrukcja `return false;` jest wykonywana wtedy, gdy choć jedna para $t[i]$ i $t[j]$ nie jest równa i sprawdzanie jest przerywane (`return` powoduje wyjście z pętli i z funkcji). Instrukcja `return true;` znajduje się już poza pętlą i jest wykonywana, gdy sprawdzanie przebiegło bez zakłóceń do końca. Profesorze de Morgan, szacunek![†]

Można by się zapytać, po co tak dobitnie podkreślamy, że badany wyraz zapisany jest małymi literami alfabetu łacińskiego. Czy to jest takie istotne? A i owszem, jest. Po pierwsze, wielkie i małe litery są uznawane za różne znaki – mają wszakże różne *kody ASCII*[‡] (wymawiaj:

[†]W szkolnej edukacji często pomija się imiona uczonych, zatem na pytanie *Jak miał na imię Morgan?* uczniowie odpowiadają bez wahania: *De!*

[‡]American Standard Code for Information Interchange.

aski). Dla przykładu wielka litera **A** ma kod ASCII 65, natomiast mała litera **a** legitymuje się kodem 97. (Tych liczb oczywiście nie musimy się uczyć na pamięć, do czego jeszcze wrócimy w tym kursie.) Zatem wyraz **Anna** nie będzie traktowany przez funkcję `pal()` jako palindrom.

Po drugie, ograniczenie do liter alfabetu łacińskiego pozwala uniknąć zawiłości związanych z kodowaniem znaków narodowych (diakrytycznych), takich jak **ą**, **ę** itd. W użyciu są różne systemy kodowania, ale nawet gdybyśmy ograniczyli się do najpopularniejszego w internecie systemu UTF-8[§], to okazałoby się, że na przykład mała litera **ą** jest reprezentowana przez *dwa znaki*, którym odpowiadają ujemne (!) wartości całkowite -60 oraz -120. Polskie litery zostawimy więc twórcom stron WWW, a my zajmiemy się sprawami bardziej podstawowymi.

[§] 8-bit Unicode Transformation Format.