Liczbowe palindromy

#odwracanie_liczby

O palindromach już mówiliśmy: są to wyrazy (teksty), które czytane wspak wyglądają tak samo. Palindromami mogą być także liczby naturalne, na przykład takie, jak 45654 lub 808. Oczywiście każda jednocyfrowa liczba jest palindromem.

Co się stanie, jeśli do danej liczby naturalnej n dodamy liczbę n', którą otrzymaliśmy z n przez odwrócenie kolejności jej cyfr. Na przykład dla $n = 123$ mamy $n' = 321$. Otrzymujemy więc $n + n' = 444$.

Otrzymaliśmy liczbę-palindrom, nieprawdaż? Prawdaż. Jednak czy zawsze tak będzie, dla dowolnego n? Nie liczylibyśmy na to, weźmy choćby $n = 607$ i mamy:

$$n + n' = 607 + 706 = 1313.$$

Jako żywo, nie jest to palindrom. Jednak jeśli uznamy 1313 za nowe n, wtedy mamy:

$$n + n' = 1313 + 3131 = 4444.$$

a to już jest palindrom. Istotnie, jeśli wykonamy odpowiednią ilość razy powyższą operację, wtedy *zawsze* otrzymamy palindrom. (Nie będziemy tutaj dowodzić tego stwierdzenia.)

Napiszemy zatem program, który wczyta liczbę naturalną n i będzie tak długo obliczał n' (liczbę o odwrotnym porządku cyfr) i dodawał je do n, aż w końcu uzyskamy palindrom. Na każdym etapie obliczeń wartość n powinna być wypisana na ekranie.

Zacznijmy od napisania funkcji `reverse_number()` (wymawiaj: *riwers namber*, z akcentem w pierwszym słowie na *e*), która odwraca kolejność cyfr liczby naturalnej n. Wykorzystamy tutaj dwa mechanizmy: obliczanie kolejnych cyfr metodą znajdowania reszty z dzielenia przez podstawę układu pozycyjnego (tutaj: 10) oraz schemat Hornera.* Pierwszy z nich dostarczy nam cyfr w porządku od ostatniej do pierwszej – wykorzystamy je w schemacie Hornera, który operuje na cyfrach w kolejności od pierwszej do ostatniej. W rezultacie utworzymy liczbę dziesiętną n' o odwróconej kolejności cyfr.

Pierwszy mechanizm działa z grubsza tak:

```
while(n > 0)
{
  . . . // tu zrobimy coś z (n % 10)
  n = n / 10;
}
```

*Poznaliśmy je w podrozdziałach *Kot dwójkowy* oraz *Kot dziesiętny*.

Teraz dołożymy schemat Hornera (zmienna r oznacza liczbę z odwróconą kolejnością cyfr):

```
int r = 0;
while(n > 0)
{
  r = r * 10 + n % 10;
  n = n / 10;
}
```

Cała funkcja wygląda tak:

```
int reverse_number(int n)
{
  int r = 0;
  while(n > 0)
  {
    r = r * 10 + n % 10;
    n = n / 10;
  }
  return r;
}
```

Osią funkcji `main()` jest pętla `while`, która kręci się, aż uzyskamy palindrom, co poznamy po spełnionym warunku:

```
n == reverse_number(n)
```

Tak powinna wyglądać ta pętla:

```
while(n != reverse_number(n))
{
  n = n + reverse_number(n);
  cout << n << endl;
}
```

Pozostało jeszcze przedstawić kompletny program:

```
#include <iostream>
using namespace std;

int reverse_number(int n)
{
  int r = 0;
  while(n > 0)
  {
    r = r * 10 + n % 10;
    n = n / 10;
  }
  return r;
}
```

```
int main()
{
  int n;
  cin >> n;
  cout << endl;
  while(n != reverse_number(n))
  {
    n = n + reverse_number(n);
    cout << n << endl;
  }
}
```

Po jego uruchomieniu i wpisaniu na przykład:

```
2358
```

otrzymamy wynik:

```
2358
10890
20691
40293
79497
```

Z kolei po wpisaniu liczby:

```
41014
```

zobaczymy tę samą liczbę jako wynik działania programu.