Wy trzej, w szeregu zbiórka!

#sortowanie
#swap

W wielu praktycznych problemach pojawia się zagadnienie *sortowania zbioru elementów*, czyli ustawienia ich w określonej kolejności, według wybranego kryterium. Może to być ustawienie uczniów według wzrostu lub numeru buta, albo sporządzenie alfabetycznej listy personaliów klientów sklepu.

Algorytmów sortowania jest bardzo wiele i stosowane są w nich rozmaite rozwiązania. Jeden z największych współczesnych informatyków, Donald E. Knuth,* uważa że można zdobyć znaczącą wiedzę i umiejętności algorytmiczne przez samo studiowanie różnych metod sortowania.

Algorytm sortowania dobierany jest w zależności od problemu, rodzaju danych i ich ilości. W naszym pierwszym przykładzie posortujemy zbiór trzech liczb $\{a, b, c\}$, które użytkownik wpisze z klawiatury. Uporządkujemy je rosnąco, to znaczy, że każda kolejna liczba powinna być większa od poprzedniej – niezależnie od kolejności, w jakiej wpisał je użytkownik.

Przy tak niewielkim zbiorze danych nie ma co bawić się w wielkie algorytmy: wystarczy kilka porównań sąsiednich liczb i ewentualnych ich przestawień, jeśli „stoją źle", czyli większa przed mniejszą. Taka operacja *przestawiania* (zamiany wartości miejscami) nazywa się po angielsku **swap** (wymawiaj: *słap*) i jest dostępna w języku C++ jako gotowa funkcja.

Generalnie chcemy osiągnąć sytuację, gdy $a < b < c$. Jeśli zatem a byłoby większe od b, to trzeba by zamienić je miejscami:

```
if(a > b)
  swap(a, b);
```

Jeżeli dla przykładu $a = 10$, zaś $b = 8$, to po wykonaniu powyższej instrukcji mamy: $a = 8$ oraz $b = 10$.

Teraz pasowałoby sprawdzić b oraz c i w razie potrzeby skorygować ich ustawienie:

```
if(b > c)
  swap(b, c);
```

Czy to już wystarczy, aby uzyskać pożądane ustawienie? Nie bardzo, łatwo wskazać *kontrprzykład*, czyli takie wartości początkowe a, b, c, aby powyższe dwa porównania/przestawienia

*Donald E. Knuth jest autorem monumentalnego dzieła algorytmicznego *The Art of Computer Programming* oraz systemu składu tekstu TEX (wymawiaj: *tech*), przy użyciu którego składany jest niniejszy podręcznik.

nie były wystarczające. Weźmy najbardziej pesymistyczny przypadek, gdy liczby na wejściu ustawione są malejąco, czyli w porządku odwrotnym, niż zamierzony, na przykład:

$$a = 10, \quad b = 8, \quad c = 6.$$

Po pierwszym porównaniu mamy ustawienie:

$$a = 8, \quad b = 10, \quad c = 6.$$

Po drugim porównaniu osiągamy:

$$a = 8, \quad b = 6, \quad c = 10.$$

An ìre mhath an-diugh,[†] jak mawiają szkoccy górale. Widać, że potrzebne jest ponowne porównanie zmiennych a oraz b, czyli powtórzenie pierwszej instrukcji, dopiero wtedy uzyskamy właściwą kolejność:

$$a = 6, \quad b = 8, \quad c = 10.$$

Niby ta sama instrukcja, ale użyta w innym kontekście (momencie) ma nowe znaczenie. Tak bywa w programowaniu...

Oto i cały program, z wprowadzaniem danych i wypisaniem rezultatu sortowania:

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

int main()
{
  int a, b, c;
  cin >> a >> b >> c;
  if(a > b)
    swap(a, b);
  if(b > c)
    swap(b, c);
  if(a > b)
    swap(a, b);
  cout << a << " " << b << " " << c << endl;
}
```

Po uruchomieniu programu i wpisaniu:

```
10 8 6
```

otrzymujemy posortowane liczby:

```
6 8 10
```

[†]Prawie na miejscu.

Doszliśmy zatem do wniosku, że do posortowania trzyelementowego zbioru wystarczą 3 porównania. Więcej porównań nie jest potrzebne (dlaczego?), natomiast ilość dokonanych przestawień zależy od początkowego ustawienia elementów.[‡]

Czy to musiały być liczby całkowite? Nie, dla dowolnego innego typu liczbowego ten algorytm wygladałby tak samo. (Trzeba tylko zmienić deklarację zmiennych a, b, c.) Więcej, sortowane elementy mogą być dowolnego typu, dla którego możliwe jest porównywanie i określenie, czy element a jest mniejszy od elementu b. Mogą to być więc znaki (`char`) lub ciągi znaków (`string`), albo inne, bardziej wyrafinowane typy danych.

A co, jeśli sortowane elementy się powtarzają? Wtedy trudno mówić o porządku ściśle *rosnącym*, czyli o relacji $a < b < c$. Raczej oczekujemy zależności $a \leqslant b \leqslant c$, czyli o porządku *niemalejącym*. Nietrudno zauważyć, że nasz algorytm działa poprawnie także dla powtarzających się wartości. Dzieje się tak dlatego, że staramy się w nim wyłapać sytuacje złego ustawienia elementów, na przykład warunek w instrukcji warunkowej to $a > b$, co jest zaprzeczeniem pożądanego warunku $a \leqslant b$.

[‡]Jeśli liczby na wejściu ustawione są rosnąco, wtedy żadne przestawienie nie jest potrzebne.