Ciągi

1

Aby sprawdzić, czy dany ciąg jest arytmetyczny, najłatwiej obliczyć różnicę dwóch pierwszych wyrazów, a następnie sprawdzić, czy każda następna różnica dwóch kolejnych wyrazów jest taka sama:

```
in >> dlugosc;
for(int i = 0; i < dlugosc; i++)
     in >> ciag[i];
int roznica = ciag[1] - ciag[0];
bool arytm = true;
for(int i = 0; i+1 < dlugosc; i++)
     if (ciag[i+1] - ciag[i] != roznica)
     {
          arytm = false;
          break;
     }
```

Zmienną *arytm* ustawiamy na **true** i jeżeli któraś z różnic okaże się inna niż pierwsza obliczona, zmieniamy jej wartość z powrotem na **false** (i możemy już przerwać pętlę instrukcją **break**). Jeśli po zakończeniu pętli zmienna *arytm* wciąż ma wartość **true**, ciąg jest arytmetyczny. W takim wypadku trzeba zwiększyć o 1 odpowiedni licznik (utrzymywany w zmiennej *ilearytm*) oraz sprawdzić, czy nowa różnica nie jest większa od dotychczas znalezionych (największą dotychczas znalezioną przechowujemy w zmiennej *maxroznica*):

```
if (arytm)
{
     ilearytm++;
     if (roznica>maxroznica)
     maxroznica = roznica;
}
```

Oczywiście musimy tak uczynić dla każdego ciągu, w odpowiedniej pętli.

2

Skoncentrujmy się wyłącznie na zagadnieniu: Jak sprawdzić, czy podana liczba jest sześcianem? Przechowywanie największego znalezionego sześcianu będziemy realizować podobnie jak w zadaniu 1. Jest kilka sposobów rozwiązania tego problemu, wzorcowe korzysta z tego, że skoro liczby w zadaniu są dodatnie i nie przekraczają 1 000 000, to mogą być sześcianami tylko liczb między 1 a 100. Sprawdzamy więc kolejno wszystkie takie liczby:

```
bool czy_szescian(int liczba)
{
    for(int x = 1; x <= 100; x++)
        if (x*x*x == liczba)
            return true;
    return false;
}
```

Alternatywnie można obliczyć przybliżony pierwiastek sześcienny z liczby za pomocą funkcji matematycznych (np. funkcja *pow* w C++), po czym zaokrąglić go do liczby całkowitej, podnieść do sześcianu i sprawdzić, czy wynik jest równy liczbie pierwotnej:

```
bool czy_szescian(int liczba)
{
    double pierwiastek = pow(liczba,1.0/3.0);
    int pz = (int)round(pierwiastek);
    if (pz*pz*pz==liczba)
        return true;
    else
        return false;
}
```

3

To zadanie może sprawić kłopoty bardzo szczególnego rodzaju. Można do niego zastosować pomysł trochę podobny jak w zadaniu 1: obliczamy różnice między każdymi dwoma kolejnymi wyrazami, a potem sprawdzamy, czy wszystkie są równe pierwszej. Jeśli któraś jest inna, odpowiedni wyraz jest tym błędnym, który mieliśmy znaleźć:

```
in >> dlugosc;
for(int i = 0; i < dlugosc; i++)
     in >> ciag[i];
for(int i = 0; i+1 < dlugosc; i++)
     roznice[i] = ciag[i+1] - ciag[i];
```

(...)

```
for(int i = 0; i+1 < dlugosc; i++)
if (roznice[i] != roznice[0])
{
     out << ciag[i+1] << endl;
     break;
}
```

Niestety, ten sposób zawodzi w dwóch przypadkach: kiedy błędny jest pierwszy wyraz oraz kiedy błędny jest drugi wyraz ciągu[5]. Aby nasz algorytm był kompletny, musimy te dwa przypadki rozpatrzyć osobno, i to zanim użyjemy podanego wyżej sposobu.

Najpierw spróbujmy wykryć błąd na pierwszym wyrazie ciągu. Jako że wpływa on tylko na pierwszą różnicę, w takim wypadku wszystkie różnice z wyjątkiem pierwszej będą sobie

równe. Nie trzeba sprawdzać wszystkich: jeśli druga różnica będzie równa kolejnej, a pierwsza inna, od razu możemy stwierdzić, że odpowiedzialny za to jest pierwszy wyraz:

```
if (roznice[0]!=roznice[1] && roznice[1]==roznice[2])
{
     out << ciag[0] << endl;
     continue;
}
```

Jeśli błędny jest drugi wyraz, sytuacja jest podobna, choć nieco bardziej złożona: drugi wyraz wpływa na pierwszą i drugą różnicę, a wszystkie dalsze będą takie same. Wystarczy więc wtedy porównać trzecią i czwartą różnicę (muszą być równe) oraz sprawdzić, czy druga i pierwsza są inne:

```
if (roznice[0]!=roznice[2] && roznice[1]!=roznice[2] &&
     roznice[3]==roznice[2])
{
     out << ciag[1] << endl;
     continue;
}
```