Rozkład na sumę liczb

#algorytm_zachłanny
#zakres_wartości
#sprawdzanie_zakresu

Zapoznaliśmy się już z ciągiem Fibonacci'ego i wykorzystamy go do rozkładu liczb naturalnych na ich sumę. Będzie nam chodziło o wykorzystanie różnych dodatnich liczb Fibonacci'ego.*

Zaczniemy od sprawdzenia, jaka największa liczba Fibonacci'ego „zmieści się" w typie danych `int`. Początek ciągu wygląda niewinnie, ale w rzeczywistości rośnie on w sposób wykładniczy, więc stosunkowo szybko wyjdziemy poza zakres typu i dostaniemy bezsensowne wyniki. Należy tu nadmienić, że w języku C/C++ nie ma wbudowanej kontroli poprawności zakresu liczb i programista musi radzić sobie sam.†

Trudno przewidzieć *a priori*, ile elementów ciągu uda się nam bezpiecznie wygenerować, więc zrobimy to metodą prób i błędów. Wykorzystamy program z poprzedniego podrozdziału, nieco zwiększając stałą MAX – do wartości 50:

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

int main()
{
  const int MAX{ 50 };
  int F[MAX + 1];
  F[0] = 0;
  F[1] = 1;
  for(int n{ 2 }; n <= MAX; n++)
    F[n] = F[n - 1] + F[n - 2];
  for(int n{ 0 }; n <= MAX; n++)
    cout << n << ' ' << F[n] << '\n';
  return 0;
}
```

Program uruchamiamy i mamy mniej więcej taki rezultat:

```
0 0
1 1
2 1
3 2
```

*Rozkład taki zawsze jest możliwy – mówi o tym twierdzenie Zeckendorfa.
†Dzięki temu obliczenia wykonywane są szybciej, ale trzeba uważać na arytmetyczne niespodzianki.

```
. . .
43 433494437
44 701408733
45 1134903170
46 1836311903
47 -1323752223
48 512559680
49 -811192543
50 -298632863
```

Jeśli przyjrzymy się uważnie, wtedy zobaczymy, że ostatnią wiarygodną liczbą jest ta o indeksie 46 (równa około 1,8 mld) i na niej właśnie trzeba skończyć generowanie ciągu. Później pojawiają się nawet liczby ujemne, co bez wątpienia świadczy, że typ `int` się „przekręcił”.

Ustalamy zatem $MAX = 46$ i zabieramy się do roboty! Jak dokonać takiego rozkładu, dajmy na to dodatniej liczby całkowitej x? Najlepiej zacząć od jak największej liczby Fibonacci'ego, która jest mniejsza od x. Właśnie: mniejsza, a nie mniejsza bądź równa – dzięki temu będziemy mieć pewność, że rozłożymy wprowadzoną liczbę na sumę co najmniej dwóch składników (dlaczego?).

Zaczniemy od $n = MAX$ i będziemy schodzić coraz niżej, aż trafimy na $F[n] < x$:

```
int x;
cin >> x;
int n{ MAX };
while(F[n] >= x)
  n--;
```

To będzie pierwszy znaleziony składnik rozkładu, zatem wypisujemy go na ekranie i odejmujemy od x, nie zapominając o zmniejszeniu n, aby przejść do następnej liczby Fibonacci'ego:

```
cout << F[n] << '\n';
x = x - F[n];
n--;
```

Teraz będziemy stopniowo wyczerpywać wprowadzoną liczbę x odejmując od niej jak największe mieszczące się w niej liczby Fibonacci'ego. Kiedy liczba x osiągnie wartość zero, będzie to sygnał, że rozkład dobiegł końca:

```
while(x > 0)
{
  if(F[n] <= x)
  {
    cout << F[n] << '\n';
    x = x - F[n];
  }
  n--;
}
```

Zauważmy, że zmienna n musi ulec zmniejszeniu bez względu na działanie instrukcji warunkowej.

Dla porządku przytoczymy całą treść omawianego programu:[‡]

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

int main()
{
  const int MAX{ 46 };
  int F[MAX + 1];
  F[0] = 0;
  F[1] = 1;
  for(int n{ 2 }; n <= MAX; n++)
    F[n] = F[n - 1] + F[n - 2];
  int x;
  cin >> x;
  int n{ MAX };
  while(F[n] >= x)
    n--;
  cout << F[n] << '\n';
  x = x - F[n];
  n--;
  while(x > 0)
  {
    if(F[n] <= x)
    {
      cout << F[n] << '\n';
      x = x - F[n];
    }
    n--;
  }
  return 0;
}
```

Po uruchomieniu programu i podaniu na przykład $x = 1234$ otrzymamy wynik:

```
987
233
13
1
```

Należy pamiętać, że wyświetlony rozkład może być jednym z możliwych, na przykład liczbę 1234 można by rozłożyć w taki sposób:

```
610
377
```

[‡]Usunęliśmy drugą pętlę `for`, która nie jest już potrzebna.

```
144
89
8
5
1
```