Ciąg Cantora

#zbiory_przeliczalne
#liczby_naturalne #liczby_całkowite
#liczby_wymierne #liczby rzeczywiste
#złożoność_obliczeniowa

W tym podrozdziale zajmiemy się pewnym problemem matematycznym: *zbiorami przeliczalnymi* (ang. **countable sets**, wymawiaj: *kałntebl sets*, z akcentem na *a*). Mimo że zagadnienie nie jest stricte związane z algorytmiką, to na jego przykładzie prześledzimy, jak odpowiedni dobór metody wpływa na złożoność obliczeniową algorytmu, a więc i czas działania programu. Teorią zbiorów zajmował się (między innymi) wielki matematyk niemiecki Georg Cantor.*

Generalnie rzecz biorąc, zbiór przeliczalny to taki zbiór, którego elementy można ustawić w ciąg, czyli – w pewnym sensie – ponumerować. Jeśli liczba elementów jest skończona, to nie ma problemu: zawsze da się to zrobić. Jeśli zbiór ma nieskończenie wiele elementów, wtedy może być różnie. Zacznijmy od czegoś prostego – zbioru liczb naturalnych $\mathbb{N}$. W tym przypadku nie będzie żadnego problemu, liczby takie układają się w ciąg *siłami natury*:

$$0, 1, 2, 3, 4, \ldots$$

A jak będzie ze zbiorem liczb całkowitych $\mathbb{Z}$?† Napiszmy, jak to z grubsza wygląda:

$$\ldots, -4, -3, -2, -1, 0, 1, 2, 3, 4, \ldots$$

Taki zapis niestety jest do luftu, bo każdy element powinien mieć jednoznacznie ustalony „numer". Mówiąc kolokwialnie, tylko w jednym miejscu powinien występować wielokropek (...). Damy sobie z tym radę, przeplatając elementy dodatnie z ujemnymi:

$$0, 1, -1, 2, -2, 3, -3, 4, -4, \ldots$$

Znaczy się, udało się, czyli zbiór liczb całkowitych też jest przeliczalny. Więcej, można powiedzieć, że ma dokładnie tyle samo elementów, co zbiór liczb naturalnych (dlaczego?).‡

Następny stopień wtajemniczenia to zbiór liczb wymiernych $\mathbb{Q}$ (ang. **rational numbers**, wymawiaj: *raszynal nambers*, z akcentem na *a* w pierwszym słowie). Są to liczby, które można przedstawić w postaci p/q, gdzie $p, q \in \mathbb{Z}$ oraz oczywiście $q \neq 0$. Tu sprawa może być nieco bardziej skomplikowana, ale pomoże nam w tym konstrukcja ciągu pochodząca właśnie od matematyka wymienionego w tytule tego podrozdziału.

*Georg Cantor (1845-1918) jest uważany za jednego z twórców podwalin matematyki współczesnej, a zwłaszcza teorii mnogości. (Jego imię wymawia się z niemiecka: *georg*.)

†Boże broń, nie oznaczajcie go przez $\mathbb{C}$ – taki symbol zarezerwowany jest w cywilizowanym świecie dla zbioru liczb zespolonych!

‡Ściślej mówiąc, te dwa zbiory są *równoliczne*. Moc takiego zbioru – czyli w pewnym sensie liczbę jego elementów – oznacza się symbolem $\aleph_0$ (*alef zero*).

Po pierwsze, z rozważań na temat zbioru liczb całkowitych wynika, że możemy spokojnie ograniczyć się do dodatnich liczb wymiernych. Jeśli z tym nam pójdzie OK, to z całym zbiorem też damy radę.

Tworzymy tabelkę, której wiersze będzie numerować p, zaś kolumny będą numerowane przez q. W każdej komórce tabelki zapiszemy ułamek p/q:

	$q \rightarrow$				
$\downarrow p$	**1**	**2**	**3**	**4**	$\cdots$
1	1/1	1/2	1/3	1/4	$\cdots$
2	2/1	2/2	2/3	2/4	$\cdots$
3	3/1	3/2	3/3	3/4	$\cdots$
4	4/1	4/2	4/3	4/4	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Łatwo zauważyć, że w powyższej tabelce występuje każda (dodatnia) liczba wymierna i to do tego nieskończoną ilość razy, choćby liczba 1 jest reprezentowana przez 1/1, 2/2, 3/3, 4/4 i tak dalej. Trzeba tylko wymyślić, jak przespacerować się po tabelce, aby uzyskać żądany ciąg.[§] Oto proponowany sposób spaceru:

	$q \rightarrow$				
$\downarrow p$	**1**	**2**	**3**	**4**	$\cdots$
1	↙	↙	↙	↙	$\cdots$
2	↙	↙	↙	↙	$\cdots$
3	↙	↙	↙	↙	$\cdots$
4	↙	↙	↙	↙	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Jak to należy rozumieć? Zaczynamy od lewego górnego rogu. Będziemy iść według biegu strzałek, a gdy dojdziemy do lewej krawędzi tabelki, przeniesiemy się na początek następnego ukośnego ciągu strzałek, do pierwszego wiersza tabeli. W powyższym przykładzie będzie wyglądać to tak:

1/1, 1/2, 2/1, 1/3, 2/2, 3/1, 1/4, 2/3, 3/2, 4/1, ...

Jeśli będziemy kontynuować taką przechadzkę w nieskończoność, wtedy wszystkie ułamki z tabelki znajdą się w naszym ciągu. Wniosek jest prosty: zbiór liczb wymiernych $\mathbb{Q}$ jest przeliczalny.[¶]

Czy możemy posunąć się dalej i zrobić to samo ze zbiorem liczb rzeczywistych $\mathbb{R}$? Nie, co za dużo, to niezdrowo. Po drodze do zbioru liczb rzeczywistych mamy jeszcze zbiór liczb niewymiernych, który musimy dołożyć do zbioru $\mathbb{Q}$. W efekcie otrzymujemy zbiór zupełnie innego sortu, którego elementów nie da się ustawić w ciąg.[‖]

[§] Kolejność elementów jest dowolna, ale powinna być jednoznaczna.

[¶] Czyli jest równoliczny ze zbiorem liczb naturalnych $\mathbb{N}$. Jest to o tyle ciekawe, że $\mathbb{Q}$ jest zbiorem gęstym (w przeciwieństwie do zbioru $\mathbb{N}$), a więc w szczególności dowolnie mały przedział otwarty na osi liczbowej zawiera nieskończenie wiele liczb wymiernych.

[‖] Mówi o tym twierdzenie Cantora (a jakże). Zbiory takie, jak $\mathbb{R}$ czy zbiór liczb niewymiernych posiadają moc określaną mianem *continuum*.

Obliczanie wyrazów ciągu Cantora

Napiszemy teraz program, który czyta dodatnią liczbę naturalną n i wypisuje n-ty wyraz ciągu Cantora obliczony według przedstawionej uprzednio metody z tabelką i kolorowymi strzałkami.

Zauważmy, że odpowiedni wyraz (p/q) można obliczyć według prostego algorytmu:

1. Zaczynamy od wczytania n oraz podstawienia $p = q = 1$.
2. Jeśli $n = 1$, wtedy wypisujemy ułamek p/q i kończymy program.
3. W przeciwnym razie zwiększamy p o 1, a q oraz n zmniejszamy o 1.
4. Jeśli $q = 0$, wtedy q otrzymuje wartość p, zaś p otrzymuje wartość 1.
5. Przechodzimy do kroku 2.

Napisanie programu działającego według powyższego algorytmu nie sprawi nam chyba problemu:

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

int main()
{
  int n, p = 1, q = 1;
  cin >> n;
  while(n > 1)
  {
    p++;
    q--;
    n--;
    if(q == 0)
    {
      q = p;
      p = 1;
    }
  }
  cout << p << "/" << q << endl;
}
```

Oczywiście ułamek wypisujemy jako licznik i mianownik oddzielnie.

Na przykład dla $n = 8$ otrzymamy wynik:

```
2/3
```

Natomiast dla $n = 8000$ otrzymamy wynik:

```
125/2
```

W tej wersji algorytmu przechodzimy przez kolejne wyrazy ciągu Cantora, aż dojdziemy do n-tego wyrazu, zatem musimy wykonać z grubsza n operacji, czyli złożoność tego algorytmu jest *liniowa*. Dla n rzędu 10^9 raczej nie mamy co liczyć na rozwiązanie w sensownym czasie. Spróbujemy zrobić to lepiej, to znaczy szybciej.

Cantor – podejście drugie

W poprzednim podejściu posuwaliśmy się wzdłuż ciagu Cantora krokami co 1 element. Zauważmy jednak, że wyrazy ciągu grupują się w coraz dłuższe podciągi: fioletowy ma długość 1, czerwony – 2, zielony – 3, niebieski – 4, i tak dalej. Zatem możemy wykonywać skoki znacznie dłuższe, za każdym razem zwiększając długość skoku o 1, co oznacza dużo lepszą złożoność algorytmu.

Długość kolejnego podciągu oznaczymy przez k – zaczniemy oczywiście od długości 1. Będziemy wykonywać skoki o k: skok polega na zmniejszeniu n o wartość k, a następnie zwiększeniu k o 1. Skoki wykonujemy, póki spełniony jest warunek $n > k$:

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

int main()
{
  int n, k = 1;
  cin >> n;
  while(n > k)
  {
    n = n - k;
    k++;
  }
```

Jeśli powyższa pętla zakończy się, będzie to oznaczało, że zostało nam odszukanie odpowiedniego (n-tego, po wszystkim zmniejszeniach wartości n) elementu w bieżącym podciągu o długości k. Elementy tego podciagu układają się według wzoru:

$$1/k, \quad 2/(k-1), \quad 3/(k-2), \quad \ldots, \quad k/1.$$

Element n-ty ma postać $n/(k-n+1)$ i taki właśnie ułamek należy wypisać:

```
  cout << n << "/" << k - n + 1 << endl;
}
```

Oczywiście program ten daje takie takie same wyniki, jak jego poprzednia wersja.

Pozostaje nam tylko oszacować złożoność tego algorytmu. Oprzemy się tutaj na znanym wzorze na sumę kolejnych liczb naturalnych:

$$1 + 2 + 3 + \cdots + (k-1) + k = \frac{k(k+1)}{2}.$$

Wyrażenie po prawej stronie będzie rzędu n, zatem wykonamy ilość skoków rzędu $\sqrt{n}$ (ewentualny czynnik $\sqrt{2}$ pomija się przy ocenie złożoności algorytmów). Tak więc ta metoda ma złożoność *pierwiastkową*, co oznacza istotną poprawę w porównaniu z poprzednim programem.

Cantor – podejście trzecie i ostatnie

Istnieje jednak jeszcze lepsza metoda, która pozwala na wyliczenie n-tego wyrazu ciągu Cantora w czasie stałym. Znów posłużymy się tutaj wzorem na sumę kolejnych liczb naturalnych, zauważając wszelako, że jeśli n-ty wyraz wypada w podciągu o długości k, to znaczy, iż spełnione są dwie nierówności:

$$1 + 2 + 3 + \cdots + (k-1) < n,$$
$$1 + 2 + 3 + \cdots + (k-1) + k \geqslant n.$$

Wartość n musi być większa od łącznej długości wszystkich wcześniejszych podciągów, natomiast n nie może przewyższać numeru pozycji ostatniego elementu w podciągu o długości k. Korzystając ze wzoru przytoczonego w poprzedniej sekcji możemy zapisać te warunki tak:

$$\frac{(k-1)k}{2} < n,$$
$$\frac{k(k+1)}{2} \geqslant n.$$

Z tego układu nierówności należy wyliczyć k (n jest dane). Może nie widać tego na pierwszy rzut oka, ale istnieje dokładnie jedno rozwiązanie, mimo iż nierówności są kwadratowe. Po prostych przekształceniach otrzymujemy:

$$k^2 - k - 2n < 0,$$
$$k^2 + k - 2n \geqslant 0.$$

Najpierw pierwsza nierówność: wyróżnik trójmianu kwadratowego wynosi:

$$\Delta = 1 + 8n,$$

zatem należy spodziewać się dwóch pierwiastków:

$$k_1 = \frac{1 - \sqrt{1+8n}}{2}, \quad k_2 = \frac{1 + \sqrt{1+8n}}{2}.$$

Rozwiązaniem tej nierówności jest przedział:

$$k_1 < k < k_2.$$

Widać jednak, że pierwiastek k_1 jest ujemny, zaś wartość k musi być dodatnia i całkowita. Tak więc możemy ograniczyć się do warunku:

$$1 \leqslant k < \frac{1 + \sqrt{1+8n}}{2}.$$

Teraz druga nierówność z układu:

$$k^2 + k - 2n \geqslant 0.$$

Wyróżnik trójmianu jest taki sam:

$$\Delta = 1 + 8n,$$

zaś jego pierwiastki wyrażają się wzorami:

$$k_3 = \frac{-1 - \sqrt{1+8n}}{2}, \quad k_4 = \frac{-1 + \sqrt{1+8n}}{2}.$$

Mamy zatem warunek:

$$(k \leqslant k_3) \vee (k_4 \leqslant k).$$

Znów musimy uwzględnić dodatniość k, co ogranicza nas do warunku:

$$\frac{-1+\sqrt{1+8n}}{2} \leqslant k.$$

Łączymy rozwiązania obydwu nierówności z układu i otrzymujemy przedział:

$$\frac{-1+\sqrt{1+8n}}{2} \leqslant k < \frac{1+\sqrt{1+8n}}{2}.$$

Zauważmy, że długość tego przedziału wynosi dokładnie 1 i mieści się w nim tylko jedna liczba całkowita – jest to nasze rozwiązanie. Przedział jest lewostronie domknięty i prawostronnie otwarty, więc mamy dwie możliwości:

1. Lewy kraniec przedziału $\frac{-1+\sqrt{1+8n}}{2}$ jest liczbą całkowitą i wtedy jest on rozwiązaniem.
2. W przeciwnym razie rozwiązaniem jest podłoga z prawego krańca, czyli $\left\lfloor \frac{1+\sqrt{1+8n}}{2} \right\rfloor$.

Musimy więc sprawdzić, czy wyrażenie $1+8n$ jest kwadratem liczby naturalnej. Jeśli zachodzi ten przypadek, wtedy lewy kraniec przedziału musi być liczbą całkowitą, ponieważ liczba pod pierwiastkiem jest nieparzysta i taki też musi być pierwiastek z niej (jeśli jest całkowity). Z kolei jeśli do liczby nieparzystej dodamy -1, wówczas otrzymamy liczbę parzystą, czyli cały ułamek będzie liczbą całkowitą. I o to chodzi.

Jak sprawdzić, czy liczba jest pełnym kwadratem? Jest kilka sposobów – spróbujemy zastosować biblioteczne funkcje `sqrt()` (pierwiastek kwadratowy) oraz `round()` (zaokrąglenie – nie obcięcie). Ponieważ obydwie te funkcje operują na typie zmiennoprzecinkowym `double`, wymagana jest pewna ostrożność. Obliczymy najpierw pierwiastek kwadratowy z $1+8n$ (jaki by on nie był), i zaokrąglimy go do liczby całkowitej:

```
double root = sqrt(1.0 + 8 * n);
int int_root = (int) round(root);
```

Argumentem funkcji `sqrt()` powinna być liczba zmiennoprzecinkowa, stąd użyliśmy zapisu 1.0, aby wymusić zamianę (konwersję) typu danych na `double`.

Wynikiem działania funkcji `round()` jest zaokrąglenie wyniku pierwiastkowania do liczby całkowitej, która jednak w dalszym ciągu jest typu `double`. Dopiero użycie konstrukcji `(int)` powoduje zmianę jej typu (dokładniej: zrzutowanie) na typ danych `int`.

Jeśli teraz będzie zachodzić poniższa równość, to wyrażenie pod pierwiastkiem jest pełnym kwadratem:

```
int_root * int_root == 1 + 8 * n
```

(Porównujemy tutaj liczby typu `int`, co jest bezpieczne.)

Sposób obliczenia wyrazu ciągu Cantora jest taki sam, jak w poprzedniej sekcji – z tym, że musimy pomniejszyć n o wyrażenie $\frac{(k-1)k}{2}$ (dlaczego?).

A oto i cały program:

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

int main()
{
  int n, k;
  cin >> n;
  double root = sqrt(1.0 + 8 * n);
  int int_root = (int) round(root);
  if(int_root * int_root == 1 + 8 * n)
    k = (-1 + int_root) / 2;
  else
    k = (int) ((1 + root) / 2);
  n = n - ((k - 1) * k) / 2;
  cout << n << "/" << k - n + 1 << endl;
}
```

Zauważmy, że konstrukcja `(int)` przed wyrażeniem typu `double` oznacza *de facto* zastosowanie funkcji podłoga (dlaczego?).

Dodaliśmy jeszcze bibliotekę `cmath` odpowiedzialną za wszelkie matematyczne ekscesy.

Jeszcze jedna uwaga na koniec: jak wielkie może być n? Ano, $8n$ nie może przekroczyć zakresu typu `int`...