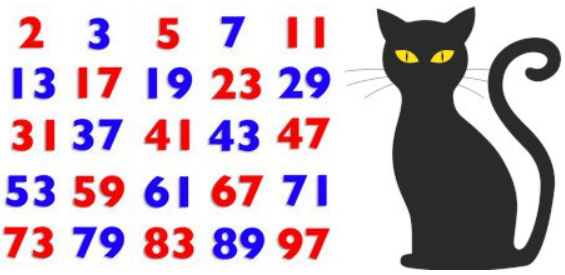
Liczby pierwsze



#liczby_pierwsze
#plik_dyskowy
#open #close
#break #sizeof

Wszyscy wiemy, co to są liczby pierwsze, tak? To takie liczby naturalne, które nie mają innych podzielników poza jednością i samą sobą. Przykładem liczb pierwszych są liczby 7 i 11, natomiast liczby takie jak 0 czy 15 są liczbami złożonymi, bo mają więcej niż dwa podzielniki. Liczba 1 nie jest ani pierwsza, ani złożona.

W jaki sposób sprawdzić, czy dana liczba naturalna n jest pierwsza (ang. **prime**, wymawiaj: *prajm*)?* W najprostszym podejściu można sprawdzić, czy liczba ma jakieś zakazane podzielniki. Wystarczy jeden taki trefny dzielnik i już nici z pierwszości. Żeby jednak potwierdzić tę pierwszość, trzeba sprawdzić wszystkie możliwości (prawa de Morgana, pamiętacie?). Podzielniki 1 oraz n są dozwolone, wobec tego funkcja sprawdzająca może mieć taką postać:

```
bool is_prime(int n)
{
  if(n < 2) return false;
  for(int p{ 2 }; p < n; p++)
    if(n % p == 0)
      return false;
  return true;
}
```

Łatwo sprawdzić, że tak napisana funkcja działa, ale dla większych liczb strasznie się ślimaczy, musimy więc ją przyspieszyć. Po pierwsze zauważmy, że nie musimy sprawdzać parzystych podzielników, wystarczy jeśli sprawdzimy parzystość liczby n. To powinno przyspieszyć funkcję `is_prime()` około dwukrotnie. Zawsze coś! Zatem przed pętlą sprawdzimy, czy $n = 2$ (i wtedy zwrócimy wartość prawdy), a w przeciwnym przypadku sprawdzimy, czy n jest parzysta (jeśli tak, to jest złożona). Sprawdzanie ewentualnych podzielników można zacząć od liczby 3 i iść co 2 (po liczbach nieparzystych):

```
bool is_prime(int n)
{
  if(n < 2) return false;
  if(n == 2) return true;
  if(n % 2 == 0) return false;
```

*Boże broń, tylko nie **first**...

```
  for(int p{ 3 }; p < n; p += 2)
    if(n % p == 0)
      return false;
  return true;
}
```

Jest lepiej, ale w dalszym ciągu nie jest zadowalająco. Gdzie szukać zwiększenia efektywności? Otóż możemy znacząco ograniczyć zbiór sprawdzanych podzielników do liczb niewiększych od $\sqrt{n}$. Zauważmy, że jeśli liczba n jest liczbą złożoną, to wtedy musi być postaci $n = a \cdot b$, gdzie liczby a oraz b są naturalne. Jeżeli a będzie liczbą większą od $\sqrt{n}$ (czyli będzie poza naszym zakresem sprawdzania), to liczba b musi być mniejsza od $\sqrt{n}$, a więc ją obejmiemy sprawdzaniem. Pamiętamy tylko, aby sprawdzać do $\sqrt{n}$ włącznie, aby nie przeoczyć liczb złożonych, które są kwadratami liczb pierwszych, na przykład $25 = 5^2$.

Zatem warunek kontynuacji pętli sprawdzającej powinien mieć postać $p \leqslant \sqrt{n}$, a jeszcze lepiej: $p^2 \leqslant n$, bo wtedy nie używamy zmiennoprzecinkowej funkcji `sqrt()`. Oczywiście pamiętamy, że w języku C++ nie mamy operatora kwadratu, więc p^2 zapiszemy jako $p \cdot p$.

Nasza funkcja ma teraz postać:

```
bool is_prime(int n)
{
  if(n < 2) return false;
  if(n == 2) return true;
  if(n % 2 == 0) return false;
  for(int p{ 3 }; p * p <= n; p += 2)
    if(n % p == 0)
      return false;
  return true;
}
```

Czy dużo zyskaliśmy? Poprzednio należało się liczyć z około $\frac{1}{2}n$ sprawdzeń (w najgorszym przypadku), natomiast teraz mamy co najwyżej $\frac{1}{2}\sqrt{n}$ sprawdzeń. Dla dużych n różnica w czasie działania może być znaczna.

Czy da się lepiej? Oj, da się. Przede wszystkim zauważmy, że wystarczy, jeśli będziemy sprawdzać podzielność tylko przez podzielniki pierwsze. Liczby parzyste (wielokrotności liczby 2) już wyeliminowaliśmy, ale jak pominąć pozostałe liczby złożone? Otóż wystarczy mieć zapisaną w programie tablicę liczb pierwszych. Już słyszymy głosy: ale przecież jest ich multum! A właśnie, że wcale nie i do tego napiszemy pomocniczy program, który wpisze tę tablicę do naszego programu.

Pierwszy problem: ile jest tych liczb? To zależy od zakresu wartości sprawdzanego n. Jeśli ograniczymy się do przedziału $0 \leqslant n \leqslant 10^9$, to wtedy potrzebujemy liczb pierwszych z zakresu od 2 do $\sqrt{10^9}$. Tak naprawdę weźmiemy ciut większy zakres, żeby mieć w tabeli liczbę pierwszą, której kwadrat na pewno przewyższa największe możliwe n (wtedy łatwiej będzie napisać warunek zakończenia sprawdzania). A sam pierwiastek kwadratowy z miliarda to raptem około 31622 z grosikami, więc nie ma co siać paniki.

Drugim problemem jest zapisanie listy tych liczb pierwszych w kodzie źródłowym naszego właściwego programu. W tym celu najpierw wypiszemy je sobie do pliku tekstowego, którego

zawartość potem wkleimy (przez schowek) do tekstu programu. Musimy jedynie nauczyć się zapisywać tekst do pliku na dysku. Każdy plik dyskowy używany w programie jest reprezentowany przez zmienną odpowiedniego typu (zmienna taka nazywa się z angielska **handler**). Powiedzmy, że nazwiemy ją po prostu f (od angielskiego **file**,[†] wymawiaj: *fajl*).

Ponieważ operacje dyskowe to działania na styku (program) $\leftrightarrow$ (system operacyjny), więc trzeba tu zachować pewną staranność. Plik dyskowy stowarzyszony ze zmienną f należy najpierw otworzyć, podając jego nazwę i określając tryb, w jakim będziemy z niego korzystać. W naszym przypadku plik będzie otwarty do zapisu (może być też otwierany do odczytu, albo do dopisywania na końcu – istnieją też inne tryby, ale te wymienione są najczęstsze). W takim razie handler f powinien być typu `ofstream` (od ang. **output file stream**, wymawiaj: *autput fajl strim*), zaś nasz plik na dysku nazwiemy `d.txt`:

```
ofstream f;
f.open("d.txt");
```

Można też zapisać to krócej:

```
ofstream f("d.txt");
```

Zapisywanie do tego pliku odbywa się tak samo, jak w przypadku strumienia danych `cout`, to jest przy pomocy operatora `<<`. Na zakończenie programu dobrze jest taki plik zamknąć przy użyciu funkcji `close()`.

Oto przykładowy program pomocniczy (treść funkcji `is_prime()` należy wziąć z ostatniego przykładu powyżej):

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

bool is_prime(int n)
{
  . . .
}

int main()
{
  ofstream f("d.txt");
  for(int n{ 2 }; true; n++)
    if(is_prime(n))
    {
      f << n << ',';
      if(n * n > 1000000000)
        break;
    }
  f.close();
  return 0;
}
```

[†]Plik.

Użyliśmy tu instrukcji `break` (wymawiaj: *brejk*) powodującej wyjście z pętli `for` (której warunek kontynuacji został ustawiony sztywno na `true`) – tak dla odmiany.

Jeśli wyświetlimy zawartość pliku `d.txt` okaże się, że zawiera on jedną dłuuugą linijkę tekstu, zawierającą listę liczb pierwszych oddzielonych przecinkami:

```
2,3,5,7, ..., 31601,31607,31627,
```

Jesteśmy już gotowi do napisania ostatecznej wersji funkcji `is_prime()`. Przed definicją funkcji wstawimy tablicę $P[\,]$ liczb pierwszych oraz ich ilość NP:

```
const int P[]{ 2,3,5,7, ..., 31601,31607,31627 };
const int NP{ sizeof(P) / sizeof(int) };
```

Jak widać, można zadeklarować tablicę bez podania jej rozmiaru, jeśli wypiszemy wszystkie jej elementy (w nawiasach klamrowych). Ostatni znak przecinka został usunięty „ręcznie”, aby uzyskać zgodność z zasadami składni języka.

Operator `sizeof` (wymawiaj: *sajzof*) zwraca rozmiar swojego argumentu w bajtach, a może nim być zmienna, struktura danych czy nazwa typu. W ten chytry sposób obliczyliśmy rozmiar tablicy $P[\,]$ jeszcze na etapie kompilacji programu.

Sama funkcja `is_prime()` będzie już bardzo prosta: jak zamierzaliśmy, ograniczymy się tylko do sprawdzania podzielników pierwszych:

```
bool is_prime(int n)
{
  if(n < 2) return false;
  for(int i{ }; P[i] * P[i] <= n; i++)
    if(n % P[i] == 0)
      return false;
  return true;
}
```

Zmienna i oznacza numer komórki w tablicy liczb pierwszych. Przypominamy ponadto, że ta wersja funkcji ma działać dla n niewiększych od miliarda.

W zastosowaniach profesjonalnych stosuje się bardziej zaawansowane metody sprawdzania, czy liczba jest pierwsza (zwłaszcza dla większych zakresów wartości). Warto wspomnieć tutaj o bardzo wydajnym teście Rabina-Millera opartym na zaawansowanej teorii liczb.[‡]

[‡]Kocurra kusi, żeby kiedyś wrócić do tego zagadnienia, ale to kawałek niełatwej matematyki.