Liczby Pierwsze

Należy znaleźć n kolejnych liczb pierwszych.

Liczba naturalna p jest liczbą pierwszą posiadającą dokładnie dwa różne podzielniki: 1 i siebie samą.

W informatyce liczby pierwsze posiadają olbrzymie zastosowanie – np. w kryptografii, czyli przy szyfrowaniu i rozszyfrowywaniu informacji. Jak jest to ważne dla handlu i bankowości w sieci, chyba nie trzeba nikogo przekonywać. Dlatego znajomość sposobów generacji liczb pierwszych jest obowiązkowa dla każdego informatyka.

Pierwsze, narzucające się podejście do problemu generacji liczb pierwszych jest bardzo prymitywne. Po prostu bierzemy kolejne liczby naturalne poczynając od 2 (1 nie jest pierwsze ponieważ dzieli się tylko przez 1 i brakuje nam drugiego podzielnika). Wybraną liczbę naturalną p próbujemy dzielić przez liczby od 2 do p - 1. Jeśli żadna z tych liczb nie jest podzielnikiem p, to liczba p jest pierwsza. Wyprowadzamy ją i w specjalnym liczniku odnotowujemy ten fakt. Gdy licznik osiągnie stan n, kończymy algorytm.

ALGORYTM WYZNACZANIA LICZB PIERWSZYCH PRZEZ SPRAWDZANIE PODZIELNOŚCI

Wejście:

n - liczba określająca ile liczb pierwszych należy wygenerować, $n \in N$

Wyjście:

n - kolejnych liczb pierwszych.

Zmienne pomocnicze:

lp - zlicza kolejno wygenerowane liczby pierwsze. $lp \in N$.
p - kolejno testowane liczby naturalne. $p \in N$.
d - kolejne dzielniki. $d \in N$.

Lista kroków:

K1:	lp ← 0	zerujemy licznik liczb pierwszych
K2:	p ← 2	generację rozpoczynamy od 2
K3:	**Dopóki** lp < n, **wykonuj** kroki K4...K8	pętla generacji liczb pierwszych
K4:	**Dla** d = 2, 3, ..., p -1, **wykonuj** krok K5	pętla sprawdzania podzielności p przez d
K5:	**Jeśli** p **mod** d = 0, **idź** do kroku K8	jeśli p dzieli się przez d, to nie jest pierwsze
K6:	Pisz p	p jest pierwsze
K7:	lp ← lp + 1	zwiększamy licznik wygenerowanych liczb pierwszych
K8:	p ← p + 1	przechodzimy do kolejnej liczby, kandydata
K9:	Zakończ	

IMPLEMENTACJA W C++:

```
#include <iostream>

using namespace std;

int main( )
{
  unsigned int n, lp, p, d;
  bool t;
  cin >> n;
  lp = 0;
  p  = 2;
  while( lp < n )
  {
    t = true;
    for( d = 2; d < p; d++ )
      if( p % d == 0 )
      {
        t = false;
        break;
      }
    if( t )
    {
      cout << p << " ";
      lp++;
    }
    p++;
  }
  cout << endl;
  return 0;
}
```

IMPLEMENTACJA W PYTHON:

```
n=int(input("Ile liczb pierwszych mam wypisać: "))
lp = 0
p = 2
while lp < n:
    t = True
    for d in range(2, p):
        if p%d == 0:
            t = False
            break
    if t:
        print(p)
        lp+=1
    p+=1
```

Powyższy algorytm generowania liczb pierwszych jest bardzo nieefektywny dla dużych n. Początkowo działa szybko, później wyraźnie zwalnia, aby na końcu osiągnąć wręcz żółwie tempo pracy. Jest to typowa cecha algorytmów o klasie złożoności obliczeniowej O (n 2) – aby przekonać się, iż liczba p jest liczbą pierwszą, algorytm musi wykonać p - 2 testy. Zastanówmy się nad tym, czy algorytm faktycznie powinien sprawdzać podzielność liczby p w całym przedziale [2, p-1].

Jeśli liczba p jest złożona, to rozkłada się na iloczyn czynników pierwszych:

$p = d_1 \times d_2 \times ... \times d_k$

Czynników tych musi być przynajmniej 2 i muszą one być różne od 1 i p (dlaczego?). Prosta analiza pokazuje nam, iż w przedziale od pierwiastka z p do p - 1 może leżeć co najwyżej jeden czynnik. Gdyby było ich więcej, to ich iloczyn przekroczyłby wartość liczby p. Skoro drugi czynnik nie może być większy od pierwiastka z p, to musi być od niego mniejszy lub równy. Z kolei przy teście na pierwszość liczby p wystarczy nam, iż znajdziemy pierwszą podzielność, aby wyeliminować p. Wynika z tego prosty wniosek – podzielność liczby p wystarczy sprawdzić dla dzielników z przedziału od 2 do pierwiastka całkowitego z p. Jeśli żaden podzielnik z tego przedziału nie dzieli p, to p jest pierwsze, ponieważ w pozostałej części przedziału <2, p-1>nie mogą już wystąpić czynniki p.

Drugie ulepszenie będzie polegało na eliminacji niektórych wartości p. Na przykład jedyną liczbą pierwszą parzystą jest 2. Wszystkie inne są już nieparzyste. Możemy pójść dalej tym tropem i dokonać dalszych eliminacji. Dwoma początkowymi liczbami pierwszymi są liczby 2 i 3. Zatem z ciągu dalszych kandydatów na liczby pierwsze możemy wyeliminować wszystkie wielokrotności liczb 2 i 3, takie jak 6, 8, 9, 10, 12, 14, 15... Liczby te mają postać 6k, 6k±2 i 6k±3, dla k = 1, 2, ... Pozostają jedynie do sprawdzenia liczby postaci 6k±1, a tych jest już zdecydowanie mniej.

Trzecie ulepszenie polega na tym, iż sprawdzamy podzielność nie przez kolejne liczby z przedziału od 2 do pierwiastka z p lecz przez liczby pierwsze wpadające w ten przedział. Po prostu algorytm w miarę znajdowania kolejnych liczb pierwszych umieszcza je w osobnej tablicy i wykorzystuje do testowania podzielności nowych kandydatów na liczbę pierwszą. Wymaga to co prawda tablicy n elementowej, ale opłaci się szybkością eliminacji liczb złożonych. Zwykle tak jest, iż szybkość pracy algorytmu zwiększa się kosztem większego zapotrzebowania na pamięć.

ALGORYTM WYZNACZANIA LICZB PIERWSZYCH PRZEZ SPRAWDZANIE PODZIELNOŚCI – WERSJA ULEPSZONA

Wejście:

n – liczba określająca ile liczb pierwszych należy wygenerować, $n \in N$.
Wyjście:

n kolejnych liczb pierwszych. Wygenerowane liczby pierwsze znajdują się również w kolejnych n elementach tablicy tlp. Indeksy rozpoczynają się od 0.

Zmienne pomocnicze:

lp	–	zlicza kolejno wygenerowane liczby pierwsze. $lp \in N$.
p	–	kolejno testowane liczby naturalne. $p \in N$.
g	–	zawiera pierwiastek całkowity z p. $g \in N$.
k	–	używane do generacji liczb p, $k \in N$.
d	–	wraz z k używane do generacji liczb p, $d \in Z$.
tlp	–	tablica liczb pierwszych. $tlp[i] \in N$, dla $i = 0, 1, ..., n - 1$.
i	–	wykorzystywane przy numeracji podzielników z tlp. $i \in N$.

LISTA KROKÓW

K1:	lp ← 0	ustawiamy licznik liczb pierwszych
K2:	k ← 1; d ← 1; p ← 2	oraz zmienne do generacji p = 6k±1
K3:	**Dopóki** lp < n, **wykonuj** kroki K4...K16	w pętli znajdujemy kolejne liczby pierwsze
K4:	**Jeśli** lp < 3, **to** p ← p + lp i **idź do** kroku K14	początkowe trzy liczby pierwsze są zadane z góry
K5:	p ← 6 × k + d	pozostałe musimy szukać wśród liczb 6k±1
K6:	**Jeśli** d = 1, **to idź do** kroku K8	modyfikujemy d i k dla następnej liczby p
K7:	d ← 1 i **idź do** kroku K9	
K8:	d ← -1; k ← k + 1	
K9:	g ← [**sqr (** p **)**]	granica sprawdzania podzielności p
K10:	i ← 2	
K11:	**Dopóki** tlp [i] ≤ g, **wykonuj** kroki K12...K13	
K12:	**Jeśli** p **mod** tlp [i] = 0, to następny obieg pętli K3	sprawdzamy podzielność p przez podzielniki z tlp
K13:	i ← i + 1	indeks następnego podzielnika
K14	tlp [lp] ← p	liczbę pierwszą zapamiętujemy w tlp
K15:	Pisz p	
K16:	lp ← lp + 1	
K17:	Zakończ	

IMPLEMENTACJA W C++

```
#include <iostream>
#include <cmath>

using namespace std;

int main( )
{
  unsigned int i, k, g, n, lp, p, * tlp;
  int d;
  bool t;

  cin >> n;
  tlp = new unsigned int [ n ];
  lp = 0; k  = 1; d  = 1; p = 2;
  while( lp < n )
  {
    t = true;
    if( lp < 3 ) p += lp;
    else
    {
```

```
      p = 6 * k + d;
      if( d == 1 )
      {
        d = -1; k++;
      }
      else d = 1;
      g = ( unsigned int )sqrt ( p );
      for( i = 2; tlp [ i ] <= g; i++ )
        if( ! ( p % tlp [ i ] ) )
        {
          t = false;
          break;
        }
    }
    if( t )
    {
      tlp [ lp++ ] = p;
      cout << p << " ";
    }
  }
  cout << endl;
  delete [ ] tlp;
  return 0;
}
```

IMPLEMENTACJA W PYTHON

```
from math import sqrt
n=int(input("Ile liczb pierwszych mam wypisać: "))
tlp = [2,3]
lp, k, d, p = 0, 1, 1, 2
while lp < n:
    t = True
    if lp < 3:
        p += lp
    else:
        p = 6*k + d
        if d == 1:
            d = -1
            k += 1
        else:
            d = 1
        g = int(sqrt(p))
        i=2
        while tlp[i] <= g:
            if not(p%tlp[i]):
                t = False
                break
            i+=1
    if t:
        lp+=1
        tlp.append(p)
print(tlp)
del tlp
```

W przedziale <2, n> należy wyszukać wszystkie liczby pierwsze

Sito Eratostenesa

Liczby pierwsze można wyszukiwać poprzez eliminację ze zbioru liczbowego wszystkich wielokrotności wcześniejszych liczb.

Przykład:

Mamy następujący zbiór liczb naturalnych:
{2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50}

Ze zbioru wyrzucamy wszystkie wielokrotności pierwszej liczby 2. Otrzymujemy następujący zbiór:
{2 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49 }

W zbiorze pozostały liczby nieparzyste – z wyjątkiem pierwszej liczby 2. Liczby podzielne przez dwa zostały wyeliminowane. Teraz eliminujemy wielokrotności kolejnej liczby 3. Otrzymamy następujący zbiór:
{2 3 5 7 11 13 17 19 23 25 29 31 35 37 41 43 47 49 }

Teraz w zbiorze pozostały liczby niepodzielne przez 2 i 3 – z wyjątkiem pierwszych 2 i 3. Zwróć uwagę, iż kolejna liczba 4 została już ze zbioru wyeliminowana. Skoro tak, to ze zbioru zniknęły również wszystkie wielokrotności liczby 4, ponieważ są one również wielokrotnościami liczby 2, a te wyeliminowaliśmy przecież na samym początku. Przechodzimy zatem do liczby 5 i eliminujemy jej wielokrotności otrzymując zbiór wynikowy:
{2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 49 }

Oprócz 2, 3 i 5 pozostałe w zbiorze liczby nie dzielą się już przez 2, 3 i 5. Liczba 6 jest wyeliminowana (wielokrotność 2), zatem przechodzimy do 7. Po wyeliminowaniu wielokrotności liczby 7 zbiór przyjmuje postać:
{2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 }

W zbiorze pozostały same liczby pierwsze.
{2 3 5 7 11 13 17 19 23 29 31 37 41 43 47}

Przy eliminacji wystarczy usunąć ze zbioru wielokrotności liczb leżących w przedziale od 2 do pierwiastka z n. Wyjaśnienie tego faktu jest identyczne jak w algorytmie szukania liczb pierwszych przez testowanie podzielności. Jeśli liczba p jest złożona, to musi posiadać czynniki pierwsze w przedziale od 2 do pierwiastka z p.

Powyższe operacje wyrzucania wielokrotności prowadzą do przesiania zbioru wejściowego. W zbiorze pozostają tylko liczby pierwsze, liczby będące wielokrotnościami poprzedzających je liczb zostają ze zbioru odsiane. Algorytm dokonujący tej eliminacji nosi nazwę sita Eratostenesa i jest jednym z najszybszych sposobów generowania liczb pierwszych.

Sito Eratostenesa jest algorytmem dwuprzebiegowym. Najpierw dokonuje on eliminacji ze zbioru liczb złożonych oznaczając je w określony sposób, a w drugim obiegu przegląda zbiór ponownie, wyprowadzając na wyjście liczby, które nie zostały oznaczone. Podstawowe znaczenie w tym algorytmie ma wybór odpowiedniej struktury danych do reprezentacji zbioru liczb. Na tym polu można dokonywać różnych optymalizacji. W pierwszym podejściu zastosujemy tablicę wartości logicznych S. Element S [i] będzie odpowiadał liczbie o wartości i. Zawartość S [i] będzie z kolei informowała o tym, czy liczba i pozostała w zbiorze (S [i] = true) lub została usunięta (S [i] = false).

Podejście I

Najpierw przygotowujemy tablicę reprezentującą zbiór liczbowy wypełniając ją wartościami logicznymi true. Odpowiada to umieszczeniu w zbiorze wszystkich liczb wchodzących w zakres od 2 do n. Następnie z tablicy będziemy usuwali kolejne wielokrotności początkowych liczb od 2 do pierwiastka całkowitego z n w pisując do odpowiednich elementów wartość logiczną false. Na koniec przeglądniemy zbiór i wypiszemy indeksy elementów zawierających wartość logiczną true – odpowiadają one liczbom, które w zbiorze pozostały.

Za pierwszą wielokrotność do wyrzucenia ze zbioru przyjmiemy kwadrat liczby i. Przyjrzyj się naszemu przykładowi. Gdy wyrzucamy wielokrotności liczby 2, to pierwszą z nich jest 4 = 2 2. Następnie dla wielokrotności liczby 3 pierwszą do wyrzucenia jest 9 = 3 2, gdyż 6 zostało wyrzucone wcześniej jako wielokrotność 2. Dla 5 będzie to 25 = 5 2, gdyż 10 i 20 to wielokrotności 2, a 15 jest wielokrotnością 3, itd. Pozwoli to wyeliminować zbędne obiegi pętli usuwającej wielokrotności.

ALGORYTM SITA ERATOSTENESA

Wejście:

n – liczba określająca górny kraniec przedziału poszukiwania liczb pierwszych, $n \in N$, $n > 1$.

Wyjście:

Kolejne liczby pierwsze w przedziale od 2 do n.

Zmienne pomocnicze:

S – tablica wartości logicznych. S [i] obrazek {false, true}, dla i = 2, 3, ..., n.
g – zawiera granicę wyznaczania wielokrotności. $g \in N$.
i – przebiega przez kolejne indeksy elementów S [i]. $i \in N$.
w – wielokrotności wyrzucane ze zbioru S, $w \in N$.

LISTA KROKÓW

K1:	**Dla** i = 2, 3, ..., n, **wykonuj** S [i] ← true	zbiór początkowo zawiera wszystkie liczby
K2:	g ← [**sqr** (n)]	obliczamy granicę eliminowania wielokrotności
K3:	**Dla** i = 2, 3, ..., g, **wykonuj** kroki K4...K8	w pętli wyrzucamy ze zbioru wielokrotności i
K4:	**Jeśli** S [i] = false, to następny obieg pętli K3	sprawdzamy, czy liczba i jest w zbiorze
K5:	w ← i^2	jeśli tak, wyrzucamy jej wielokrotności
K6:	**Dopóki** w ≤ n, **wykonuj** kroki K7...K8	ze zbioru
K7:	S [w] ← false	
K8:	w ← w + i	następna wielokrotność
K9:	**Dla** i = 2, 3, ..., n, **wykonuj** krok K10	przeglądamy zbiór wynikowy
K10:	**Jeśli** S [i] = true, to pisz i	wyprowadzając pozostałe w nim liczby
K11:	Zakończ	

IMPLEMENTACJA W C++

```
#include <iostream>
#include <cmath>

using namespace std;

int main( )
{
  unsigned int g, i, n, w;
  bool * S;

  cin >> n;
  S = new bool [ n + 1 ];
  for( i = 2; i <= n; i++ ) S [ i ] = true;
  g = ( unsigned int )sqrt ( n );
  for( i = 2; i <= g; i++ )
    if( S [ i ] )
    {
     w = i * i;
      while( w <= n )
      {
        S [ w ] = false; w += i;
      }
    }
  for( i = 2; i <= n; i++ ) if( S [ i ] ) cout << i << " ";
  cout << endl;
  delete [ ] S;
  return 0;
}
```

IMPLEMENTACJA W PYTHON

```
from math import sqrt
n=int(input("Podaj liczbę n, gdzie n jest końcem przedziału (2, n): "))
S = [True, True]
for i in range(2, n+1):
    S.append(True)
g = int(sqrt(n))
for i in range(2, g+1):
    if S[i]:
        w=i*i
        while w<=n:
            S[w]=False
            w+=i
for i in range(2, n+1):
    if S[i]:
        print(i)
del S
```

Podejście 2

Jedyną parzystą liczbą pierwszą jest 2. Zatem wszystkie pozostałe liczby parzyste (4, 6, 8, ...) już nie mogą być pierwsze. Utwórzmy tablicę S, której elementy będą reprezentowały kolejne liczby nieparzyste, począwszy od 3. Poniżej przedstawiamy początkowe przypisania indeksów liczbom nieparzystym.

indeks	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
liczba	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31	33	35	37	39	41	43	45	47	49	51	53	55	57	59	61	63	65	67	69	71	73	75	77	79	81

Komórka o indeksie i oznacza liczbę o wartości 2i + 1. Np. komórka 7 reprezentuje liczbę 2 × 7 + 1 = 15.
Liczba o wartości k jest reprezentowana przez komórkę (k - 1) / 2. Np. liczbę 45 reprezentuje komórka (45 - 1) / 2 = 22.
Z poprzedniego algorytmu pamiętamy, że pierwszą wyrzucaną wielokrotnością jest zawsze kwadrat liczby podstawowej. Nasza tablica rozpoczyna się od liczby 3, zatem pierwszą wielokrotnością, którą usuniemy, będzie liczba 9 na pozycji 4:

indeks	1	2	3	**4**	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
liczba	3	5	7	**9**	11	13	15	17	19	21	23	25	27	29	31	33	35	37	39	41	43	45	47	49	51	53	55	57	59	61	63	65	67	69	71	73	75	77	79	81

Zauważ, że kolejne wielokrotności liczby 3, które są reprezentowane w tablicy, znajdują się w niej w odstępach co 3:

indeks	1	2	3	**4**	5	6	**7**	8	9	**10**	11	12	**13**	14	15	**16**	17	18	**19**	20	21	**22**	23	24	**25**	26	27	**28**	29	30	**31**	32	33	**34**	35	36	**37**	38	39	**40**
liczba	3	5	7	**9**	11	13	**15**	17	19	**21**	23	25	**27**	29	31	**33**	35	37	**39**	41	43	**45**	47	49	**51**	53	55	**57**	59	61	**63**	65	67	**69**	71	73	**75**	77	79	**81**

Oznaczmy przez p odstępy pomiędzy wielokrotnościami, a przez q pozycję kwadratu liczby, której wielokrotności usuwamy z tablicy. Na początku mamy:

p 0 = 3
q 0 = 4

Po usunięciu tych wielokrotności tablica S nie będzie zawierała dalszych liczb podzielnych przez 3. Przejdźmy do kolejnej liczby, czyli do 5. Kwadrat 5 znajduje się na pozycji 12:

indeks	1	2	3	4	5	6	7	8	9	10	11	**12**	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
liczba	3	5	7	9	11	13	15	17	19	21	23	**25**	27	29	31	33	35	37	39	41	43	45	47	49	51	53	55	57	59	61	63	65	67	69	71	73	75	77	79	81

Kolejne wielokrotności 5 znajdują się w naszej tablicy w odstępach co 5 (niektóre z nich są usunięte, ponieważ są również wielokrotnościami liczby 3) :

indeks	1	2	3	4	5	6	7	8	9	10	11	**12**	13	14	15	16	**17**	18	19	20	21	**22**	23	24	25	26	**27**	28	29	30	31	**32**	33	34	35	36	**37**	38	39	40
liczba	3	5	7	9	11	13	15	17	19	21	23	**25**	27	29	31	33	**35**	37	39	41	43	**45**	47	49	51	53	**55**	57	59	61	63	**65**	67	69	71	73	**75**	77	79	81

Zapiszmy:

p 1 = p 0 + 2 = 3 + 2 = 5
q 1 = q 0 + 8 = 4 + 8 = 12

Następna liczba to 7 i jej pierwsza wielokrotność 49 na pozycji 24. Kolejne wielokrotności są w odstępach co 7:

indeks	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	**24**	25	26	27	28	29	30	**31**	32	33	34	35	36	37	**38**	39	40
liczba	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31	33	35	37	39	41	43	45	47	**49**	51	53	55	57	59	61	**63**	65	67	69	71	73	75	**77**	79	81

Znów zapiszmy:

p 2 = p 1 + 2 = 5 + 2 = 7
q 2 = q 1 + 12 = 12 + 12 = 24

Następna liczba to 9 i jej pierwsza wielokrotność 81 na pozycji 40. Kolejne wielokrotności są w odstępach co 9:

indeks	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
liczba	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31	33	35	37	39	41	43	45	47	49	51	53	55	57	59	61	63	65	67	69	71	73	75	77	79	81

Znów zapiszmy:

p3 = p2 + 2 = 7 + 2 = 9
q3 = q2 + 16 = 24 + 16 = 40

Wyrzucanie wielokrotności liczby 9 możemy pominąć, ponieważ sama liczba 9 jest już wyrzucona z tablicy przez 3. Ostatnią rozważaną tu liczbą jest 11 o kwadracie równym 121. Liczba 121 znajduje się na pozycji (121-1)/2 = 60:

p4 = p3 + 2 = 9 + 2 = 11
q4 = q3 + 20 = 40 + 20 = 60

W tablicy ta pozycja już nie występuje (indeksy kończą się na wartości 40), zatem kończymy. W S pozostały jedynie liczby pierwsze. Na koniec wyprowadźmy wzory rekurencyjne dla kolejnych wartości p oraz q. W tym celu wystarczy zauważyć prostą zależność:

Kolejne p powstaje z poprzedniego przez dodanie 2. Kolejne q powstaje z poprzedniego przez dodanie 2 (p-1):

i	p_i	$2(p_i-1)$	q_i
0	3		4
1	3+2 = 5	2x (5-1) = 8	4+8 = 12
2	5+2 = 7	2x (7-1) = 12	12+12 = 24
3	7+2 = 9	2x (9-1) = 16	24+16 = 40
4	9+2 = 11	2x (11-1) = 20	40+20 = 60

Zatem możemy zapisać wzór rekurencyjny:

$p_0 = 3$, $q_0 = 4$ – wartości startowe

Dla $i > 0$:

$p_i = p_{i-1} + 2$
$q_i = q_{i-1} + 2 (p_i - 1)$

Po tych rozważaniach przystępujemy do zapisu algorytmu.

ALGORYTM ULEPSZONEGO SITA ERATOSTENESA

Wejście:

n – liczba określająca górny kraniec przedziału poszukiwania liczb pierwszych, $n \in N$, $n > 1$.

Wyjście:

Kolejne liczby pierwsze w przedziale od 2 do n.

Zmienne pomocnicze:

S – tablica wartości logicznych. S [i] ∈ {false, true}, dla i = 1, 2, ..., n/2 - 1.
i, k – przebiega przez indeksy S [i]. i ∈ N.
p, q – odstęp wielokrotności oraz pierwsza wielokrotność, p, q ∈ N.
m – połowa z n, m ∈ N.

LISTA KROKÓW:

K1:	**Jeśli** n jest nieparzyste, **to** n ← n + 1	n sprowadzamy do parzystego
K2:	m ← n **shr** 1	m to połowa z n
K3:	**Dla** i = 1, 2, ..., m - 1 **wykonuj** S [i] ← true	w S są początkowo wszystkie liczby nieparzyste
K4:	i ← 1	indeks kolejnych liczb liczb pierwszych w S
K5:	p ← 3; q ← 4	odstęp 3, pierwsza wielokrotność 4 (9)
K6:	**Jeśli** S [i] = false, **to idź do** kroku K11	przeskakujemy liczby wyrzucone z S
K7:	k ← q	w k indeks pierwszej wielokrotności liczby 2i + 1
K8:	**Dopóki** k < m, **wykonuj** kroki K9...K10	wyrzucamy z S wielokrotności
K9:	S [k] ← false	
K10:	k ← k + p	wyznaczamy pozycję kolejnej wielokrotności, przesuniętą o odstęp p
K11:	i ← i + 1	indeks następnej liczby w S
K12:	p ← p + 2	zwiększamy odstęp wielokrotności
K13:	q ← q + (p **shl** 1) - 2	wyznaczamy pierwszą wielokrotność
K14:	**Jeśli** q < m, **to idź do** kroku K06	
K15:	Pisz 2	pierwsza liczba pierwsza wyprowadzana bezwarunkowo
K16:	**Dla** i = 1, 2, ..., m-1, **wykonuj** krok K17	przeglądamy S wypisując pozostałe w niej liczby pierwsze
K17:	**Jeśli** S [i] = true, **to pisz** 2i + 1	
K18:	Zakończ	

IMPLEMENTACJA C++

```
#include <iostream>

using namespace std;

int main( )
{
  unsigned int i, k, p, q, n, m;
  bool * S;

  cin >> n;
  if( n & 1 ) n++;
```

```
  m = n >> 1;
  S = new bool [ m + 1 ];
  for( i = 1; i < m; i++ ) S [ i ] = true;
  i = 1; p = 3; q = 4;
  do
  {
    if( S [ i ] )
    {
      k = q;
      while( k < m )
      {
        S [ k ] = false; k += p;
      }
    }
    i++; p += 2;
    q += ( p << 1 ) - 2;
  } while( q < m );
  cout << 2 << " ";
  for( i = 1; i < m; i++ ) if( S [ i ] ) cout << ( i << 1 )+1 << " ";
  cout << endl;
  delete [ ] S;
  return 0;
}
```

IMPLEMENTACJA W PYTHON

```
n=int(input("Podaj liczbę n, gdzie n jest końcem przedziału (2, n): "))
S  = [True]
if(n&1): #operacje bitowe na liczbach
    n+=1
m=n>>1
for i in range(1, m):
    S.append(True)
print(S)
i, p, q = 1, 3, 4
if S[i]:   # ponieważ w python nie ma pętli do while pierwszy cykl należy wykonać indywidualnie
    k=q
    while(k < m):
        S[k] = False
        k += p
i+=1
p+=2
q+=(p << 1) - 2
while q < m:
    if S[i]:
        k=q
        while(k < m):
            S[k] = False
            k += p
    i+=1
    p+=2
    q+=(p << 1) - 2
print("2")
for i in range(1, m):
    if S[i]:
        print((i << 1)+1)
del S
```

Podejście 3

Autorem prezentowanego algorytmu jest chiński informatyk **Xuedong Luo**. W rozwiązaniu 2 zbiór liczbowy ograniczyliśmy do liczb nieparzystych. Teraz pójdziemy dalej i wrzucimy z tego zbioru dodatkowo wszystkie liczby podzielne przez 3. W efekcie nasz zbiór S przybierze postać:

Wartość indeksu i:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	...	i_n	i_p	...	m
$S[\ i\ ]$ odpowiada liczbie:	5	7	11	13	17	19	23	25	29	31	35	37	41	43	47	49	53	55	59	61	...	$3i_n+2$	$3i_p+1$		n

gdzie:
in – indeks nieparzysty,
ip – indeks parzysty.

Element S [i] odwzorowuje liczbę niepodzielną ani przez 2, ani przez 3. Wartość liczby określamy w zależności od tego, czy indeks i jest parzysty, czy nieparzysty:

S [in] → 3in + 2
S [ip] → 3ip + 1

Dodatkowe założenie obejmuje n, które powinno spełniać równanie:

n = 3m + 2, gdzie m jest liczbą nieparzystą

Algorytm wyrzuca ze zbioru S wszystkie wielokrotności początkowych liczb. Najpierw wyznaczana jest pozycja kwadratu liczby na podstawie poprzedniej pozycji. W dalszej kolejności algorytm wyznacza wszystkie pozycje wielokrotności liczby począwszy od jej kwadratu i pomijając wielokrotności podzielne przez 2 lub przez 3. Elementy zbioru S znajdujące się na tych pozycjach są zaznaczane jako wyrzucone.

Algorytm jest trochę trudny do zrozumienia, lecz działa znakomicie. Zaletą jest zmniejszone 3 krotnie zapotrzebowanie na pamięć w stosunku do podstawowego algorytmu Euklidesa. Przy określaniu n pamiętaj, iż musi spełniać zależność n = 3m + 2, gdzie m jest liczbą nieparzystą. W przeciwnym razie ostatnie liczby pierwsze mogą nie zostać wyliczone.

Algorytm ulepszonego sita Eratostenesa

Wejście:

n – liczba określająca górny kraniec przedziału poszukiwania liczb pierwszych, $n \in N$, $n > 4$, $n = 3m + 2$, gdzie m jest liczbą nieparzystą.

Wyjście:

Kolejne liczby pierwsze w przedziale od 2 do n.

Zmienne pomocnicze:

m – ilość elementów w zbiorze S. $m \in N$, $m = [(n - 2) / 3]$.
S – tablica wartości logicznych. $S [i] \in \{false, true\}$, dla $i = 1, 2, ..., m$.
c – pozycja pierwszej wielokrotności, wskazuje kwadrat liczby, $c \in N$.
q – górna granica pozycji liczb, których wielokrotności są usuwane ze zbioru, $q \in N$.
k, t, ij – zmienne służą do wyznaczania pozycji następnych wielokrotności. $k, t, ij \in N$.
i, j – indeksy elementów S, $i, j \in N$.

LISTA KROKÓW:

K1:	m ← (n - 2) **div** 3	obliczamy liczbę elementów w S
K2:	**Dla** i = 1, 2, ..., m: **wykonuj** S [i] ← true	w zbiorze S są początkowo wszystkie liczby
K3:	c ← 0	c będzie wskazywało pozycję kwadratu kolejnej liczby w zbiorze
K4:	k ← 1; t ← 2	Zmienne pomocnicze:
K5:	q ← [**sqr (** n **)**/3]	granica pozycji liczb, których wielokrotności eliminujemy
K6:	**Dla** i = 1, 2, ..., q: **wykonuj** kroki K7...K15	w pętli wyznaczamy pozycje liczb do usunięcia
K7:	k ← 3 - k	
K8:	c ← c + 4k × i	pozycja kwadratu liczby o indeksie i
K9:	j ← c	do wyrzucania liczb posłuży indeks j
K10:	ij ← 2i × (3-k) + 1	
K11:	t ← t + 4k	
K12:	**Dopóki** j ≤ m, **wykonuj** kroki K13...K15	pętla usuwająca liczby
K13:	S [j] ← false;	usuwamy j-tą liczbę
K14:	j ← j + ij	; obliczamy pozycję następnej do usunięcia liczby
K15:	ij ← t - ij	
K16:	Pisz 2 3	dwie początkowe liczby pierwsze wyprowadzamy bezwarunkowo
K17:	**Dla** i = 1, 2, ..., m: **wykonuj** kroki K18...K19	przeglądamy zbiór S
K18:	**Jeśli** S [i] = false, to następny obieg pętli K17	pomijamy liczby usunięte
K19:	**Jeśli** i nieparzyste, **to pisz** 3i + 2 inaczej pisz 3i + 1	inaczej wyprowadzamy wartości liczb, które w zbiorze pozostały
K20:	Zakończ	

IMPLEMENTACJA W C++

Program w pierwszym wierszu czyta liczbę n = 3m + 2, gdzie m jest liczbą nieparzystą. Jeśli n nie spełnia warunku, to program odpowiednio dopasuje n i m, co może skutkować powiększeniem przedziału wyszukiwania liczb.

```
#include <iostream>
#include <cmath>
using namespace std;
int main( )
{
  unsigned int c, k, t, q, m, n, i, j, ij;
  bool * S;
  cin >> n;
  m = n / 3;
  if( ! ( m & 1 ) ) m++;
  S = new bool [ m + 1 ];
```

```
  c = 0; k = 1; t = 2;
  q = (  ( unsigned int )sqrt ( n ) ) / 3;
  for( i = 1; i <= m; i++ ) S [ i ] = true;
  for( i = 1; i <= q; i++ )
  {
    k  = 3 - k;
    c  += ( k << 2 ) * i;
    j  = c;
    ij = ( i << 1 ) * ( 3 - k ) + 1;
    t  += k << 2;
    while( j <= m )
    {
      S [ j ] =  false;
      j    += ij;
      ij   =  t - ij;
    }
  }
  cout << 2 << " " << 3 << " ";
  for( i = 1; i <= m; i++ )
    if( S [ i ] )
    {
      if( i & 1 )     cout << 3 * i + 2;
      else            cout << 3 * i + 1;
      cout << " ";
    }
  cout << endl;
  delete [ ] S;
  return 0;
}
```

IMPLEMENTACJA W PYTHON

```
from math import sqrt
n=int(input("Podaj liczbę n, n=3m+2, gdzie m jest liczbą nieparzystą: "))
S  = [True]
m = int(n/3)
if not(m & 1):
    m += 1
c, k, t = 0, 1, 2
q = int(sqrt(n)/3)
for i in range(1, m+1):
    S.append(True)
for i in range(1, q+1):
    k = 3 - k
    c += (k << 2)*i
    j = c
    ij = (i << 1) * (3 - k) + 1
    t += k << 2
    while j <= m:
        S[j] = False
        j += ij
        ij = t - ij
print(2,3)
for i in range(1, m+1):
    if S[i]:
        if i & 1:
            print(3 * i + 2)
        else:
            print(3 * i + 1)
del S
```

Sito Liniowe

W roku 1978 dwaj informatycy, David Gries z Cornell University oraz Jayadev Misra z University of Texas w Austin, opublikowali w ramach ACMs algorytm wyszukiwania liczb pierwszych, który działa w czasie liniowym O (n) i z tego powodu został nazwany sitem liniowym (ang. linear sieve). W porównaniu do algorytmu sita Eratostenesa, sito liniowe wyrzuca ze zbioru liczby złożone tylko jeden raz – sito Eratostenesa robi to wielokrotnie, gdy wyrzucana liczba rozkłada się na różne czynniki pierwsze. Z drugiej strony algorytm sita liniowego wykorzystuje intensywnie mnożenie, co może powodować spadek jego efektywności dla małych n.

Algorytm operuje na zbiorze liczbowym S = { 2, 3, ..., n }, w którym zdefiniowano dwie operacje:

usuń (S, i) – jest zdefiniowana dla liczby i ∈ S: S ← S \ { i }

następne (S, i) – jest funkcją zdefiniowaną dla liczby i ∈ S, której wynikiem jest bezpośrednio następna liczba naturalna w S, większa od i.
W algorytmie wykorzystuje się twierdzenie:
Liczba złożona x może zostać jednoznacznie zapisana jako:

$$x = p^k \cdot q$$

gdzie:: p jest najmniejszą liczbą pierwszą dzielącą x bez reszty,
k ≥ 1,
p = q lub p jest mniejsze od najmniejszej liczby pierwszej, która dzieli q bez reszty.
Ponieważ w powyższy sposób nie można zapisać żadnej liczby pierwszej, zatem w celu usunięcia ze zbioru S liczb złożonych, algorytm musi jedynie wygenerować wszystkie kombinacje trójek (p, q, k) i usunąć odpowiadające im liczby złożone x. Sztuka polega na uzyskaniu każdej kombinacji dokładnie jeden raz oraz w takiej kolejności, aby następna kombinacja mogła być efektywnie wyliczona z kombinacji bieżącej. W tym celu algorytm stosuje mocne uporządkowanie α dla liczb złożonych wyznaczanych przez trójki (p, q, k), indukowane uporządkowaniem leksykograficznym odpowiednich trójek (p, q, k).
Porządek α zdefiniowany jest następująco:

$$x = p^k \cdot q$$
$$\bar{x} = \bar{p}^{\bar{k}} \cdot \bar{q}$$
$$x \; \alpha \; \bar{x} \Leftrightarrow \begin{cases} p < \bar{p} \\ p = \bar{p} \wedge q < \bar{q} \\ p = \bar{p} \wedge q = \bar{q} \wedge k < \bar{k} \end{cases}$$

Poniższa tabelka przedstawia to uporządkowanie, jednocześnie wyjaśniając sposób pracy algorytmu. Wiersze zawierają kolejne wartości par (p, q) wraz z zawartością zbioru S przed usunięciem liczb złożonych zawierających te składniki p i q. Wartości liczb złożonych x = p k × q dla k = 1, 2, 3, 4 zostały zaznaczone na czerwono.

p	q	S
2	2	2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
2	3	2 3 5 6 7 9 10 11 12 13 14 15 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
2	5	2 3 5 7 9 10 11 13 14 15 17 18 19 20 21 22 23 25 26 27 28 29 30 31
2	7	2 3 5 7 9 11 13 14 15 17 18 19 21 22 23 25 26 27 28 29 30 31
2	9	2 3 5 7 9 11 13 15 17 18 19 21 22 23 25 26 27 29 30 31
2	11	2 3 5 7 9 11 13 15 17 19 21 22 23 25 26 27 29 30 31
2	13	2 3 5 7 9 11 13 15 17 19 21 23 25 26 27 29 30 31
2	15	2 3 5 7 9 11 13 15 17 19 21 23 25 27 29 30 31
3	3	2 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31
3	5	2 3 5 7 11 13 15 17 19 21 23 25 29 31
3	7	2 3 5 7 11 13 17 19 21 23 25 29 31
5	5	2 3 5 7 11 13 17 19 23 25 29 31
		2 3 5 7 11 13 17 19 23 29 31

Zasada działania algorytmu sita liniowego jest następująca: Za p i q przyjmujemy pierwszą liczbę zbioru. Za k przyjmujemy 1. Następnie w pętli generujemy liczby x = p k × q dla kolejnych k = 1, 2, ... do momentu, gdy x przekroczy n. Wygenerowane liczby x usuwamy ze zbioru. Wtedy za q przyjmujemy następną liczbę w zbiorze i całość powtarzamy. Jeśli pierwsze x, dla k = 1 wykracza poza n, to zmieniamy p i q na następną liczbę, która ze zbioru S nie została jeszcze wyrzucona. Algorytm kończymy, jeśli iloczyn p × q wykracza poza n. Z przedstawionej tablicy wynika jasno, iż każde x pojawia się tylko jeden raz, nie dochodzi więc do zbędnych przebiegów.

ALGORYTM SITA LINIOWEGO

Wejście:

n – liczba określająca górny kraniec przedziału poszukiwania liczb pierwszych, $n \in N$, $n > 1$.
Wyjście:

Kolejne liczby pierwsze w przedziale od 2 do n.

Zmienne pomocnicze:

S – zbiór liczbowy. $S \subset \{2, 3, ..., n\}$.
p – mnożnik – liczba pierwsza, $p \in S$.
q – mnożnik drugi. $q \in S$.
x – liczba złożona, $x \in S$.
i – liczba, $i \in S$.

LISTA KROKÓW:

K1:	***S*** ← {2, 3, ..., *n*}	początkowo zbiór S zawiera wszystkie liczby z przedziału [2, n]
K2:	*p* ← 2	pierwszy mnożnik
K3:	Dopóki *p* × *p* ≤ *n*, **wykonuj** kroki K4...K11	
K4:	*q* ← *p*	drugi mnożnik
K5:	Dopóki *p* × *q* ≤ *n*, **wykonuj** kroki K6...K10	
K6:	*x* ← *p* × *q*	obliczamy liczbę złożoną
K7:	Dopóki *x* ≤ *n*, **wykonuj** kroki K8...K9	w pętli wyrzucamy ze zbioru liczby złożone
K8:	usuń (***S***, *x*)	
K9:	*x* ← *p* × *x*	następna liczba złożona
K10:	*q* ← następne (***S***, *q*)	za q przyjmujemy następną liczbę w zbiorze, które nie została jeszcze wyrzucona
K11:	*p* ← następne (***S***, *p*)	za p przyjmujemy następną liczbę pierwszą ze zbioru
K12:	**Dla** *i* = 2, 3, ..., *n*, **wykonuj** krok K13	przeglądamy zbiór i wypisujemy pozostałe w nim liczby pierwsze
K13:	Jeśli *i* ∈ *S*, **to pisz** *i*	
K14:	Zakończ	

IMPLEMENTACJA W C++

```
#include <iostream>

using namespace std;

int main( )
{
  unsigned int i, n, p, q, x;
  bool * S;

  cin >> n;
  S = new bool [ n + 1 ];
  for( i = 2; i <= n; i++ ) S [ i ] = true;
  p = 2;
  while( p * p <= n )
  {
    q = p;
    while( p * q <= n )
    {
      x = p * q;
      while( x <= n )
      {
        S [ x ] = false;
        x *= p;
      }
      while( !S [ ++q ] );
    }
    while( !S [ ++p ] );
  }
  for( i = 2; i <= n; i++ ) if( S [ i ] ) cout << i << " ";
  cout << endl;
  delete [ ] S;
  return 0;
}
```

IMPLEMENTACJA W PYTHON

```
n=int(input("Podaj liczbę n, gdzie n jest końcem przedziału (2, n): "))
S  = [True,True]
for i in range(2, n+1):
    S.append(True)
p = 2
while p*p <= n:
    q=p
    while p*q <= n:
        x = p*q
        while x <= n:
            S[x] = False
            x *= p
        q+=1
        while not(S[q]):
            q+=1
    p+=1
    while not(S[p]):
        p+=1
for i in range(2, n+1):
    if S[i]:
        print(i)
del S
```

Sito Atkina-Bernsteina

Sito Atkina-Bernsteina (ang. Atkin-Bernstein Sieve) należy do nowoczesnych algorytmów wyszukujących liczby pierwsze w przedziale od 2 do zadanej liczby naturalnej n. Jest to zoptymalizowany algorytm w stosunku do starożytnego sita Eratostenesa, który najpierw wykonuje wstępne przetwarzanie zbioru liczb, a następnie wyrzuca z niego wszystkie wielokrotności kwadratów początkowych liczb pierwszych zamiast samych liczb pierwszych. Twórcami algorytmu są dwaj profesorowie University of Illinois w Chicago: Arthur Oliver Lonsdale Atkin – emerytowany profesor matematyki oraz Daniel Julius Bernstein – profesor matematyki, kryptolog i programista.

Przedstawiony poniżej algorytm i jego realizacja nie są optymalne. Umieściłem je tutaj ze względów dydaktycznych. Prawdziwą implementację sita Atkina można znaleźć na stronie domowej prof. Bernsteina, czyli u źródeł i tam odsyłam wszystkich zainteresowanych tym tematem.

W przeciwieństwie do sita Eratostenesa, sito Atkina-Bernsteina rozpoczyna pracę ze zbiorem S, w którym wszystkie liczby są zaznaczone jako złożone (czyli nie pierwsze). Ma to sens, ponieważ algorytm ignoruje kompletnie liczby podzielne przez 2, 3 lub 5. Zaoszczędzamy czas na wyrzucaniu ich ze zbioru.

Algorytm opiera swoje działanie na następujących faktach matematycznych:

Wszystkie liczby dające resztę z dzielenia przez 60 równą 0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56 lub 58 są podzielne przez 2, zatem nie są pierwsze – algorytm je ignoruje.
Wszystkie liczby dające resztę z dzielenia przez 60 równą 3, 9, 15, 21, 27, 33, 39, 45, 51 lub 57 są z kolei podzielne przez 3 i również nie są pierwsze – algorytm je ignoruje.
Wszystkie liczby dające resztę z dzielenia przez 60 równą 5, 25, 35 lub 55 są podzielne przez 5 i także nie są pierwsze – algorytm je ignoruje.
Wszystkie liczby dające resztę z dzielenia przez 60 równą 1, 13, 17, 29, 37, 41, 49 lub 53 posiadają resztę z dzielenia przez 12 równą 1 lub 5. Liczby te są pierwsze wtedy i tylko wtedy, gdy liczba rozwiązań równania $4x^2 + y^2 = n$ jest nieparzysta dla $x, y \in N$, a liczba n jest niepodzielna przez kwadraty liczb naturalnych.
Wszystkie liczby dające resztę z dzielenia przez 60 równą 7, 19, 31 lub 43 posiadają resztę z dzielenia przez 12 równą 7. Są one liczbami pierwszymi wtedy i tylko wtedy, gdy liczba rozwiązań równania $3x^2 + y^2 = n$ jest nieparzysta dla $x, y \in N$, a liczba n jest niepodzielna przez kwadraty liczb naturalnych.
Wszystkie liczby dające resztę z dzielenia przez 60 równą 11, 23, 47 lub 59 posiadają resztę z dzielenia przez 12 równą 11. Są one liczbami pierwszymi wtedy i tylko wtedy, gdy liczba rozwiązań równania $3x^2 – y^2 = n$ jest nieparzysta dla $x, y \in N$, a liczba n jest niepodzielna przez kwadraty liczb naturalnych.

ALGORYTM SITA ATKINA-BERNSTEINA

Wejście:
n – liczba określająca górny kraniec przedziału poszukiwania liczb pierwszych, $n \in N$, $n > 4$.

Wyjście:

Kolejne liczby pierwsze w przedziale od 2 do n.
Zmienne pomocnicze:
S – zbiór liczbowy. $S \subset \{5, 6, ..., n\}$.
g – granica przetwarzania liczb w S, $g \in N$.
x, y – używane w równaniach kwadratowych. $x, y \in N$.
xx, yy – kwadraty x i y, $xx, yy \in N$.
z – rozwiązanie równania, $z \in N$.
i – indeks, $i \in N$.

LISTA KROKÓW

K1:	**Dla** *i* = 5, 6, ..., *n*: **wykonuj** *S* [*i*] ← false	inicjujemy zbiór liczbowy – wszystkie liczby zaznaczamy jako liczby złożone
K2:	*g* ← [sqr (*n*)]	; granica wyznaczania początkowych liczb pierwszych
K3:	**Dla** każdego (*x, y*) z [1, *g*]: **wykonuj** kroki K04...K12	szukamy rozwiązań równań kwadratowych
K4:	*xx* = *x* × *x*	$xx = x^2$
K5:	*yy* = *y* × *y*	$yy = y^2$
K6:	*z* ← (*xx* **shl** 2) + *yy*	$z = 4x^2 + y^2$
K7:	**Jeśli** (*z* ≤ *n*) ∧ (*z* **mod** 12 = 1 ∨ *z* **mod** 12 = 5), **to** *S* [*z*] ← ¬ *S* [*z*]	jeśli spełnione są warunki, to zmieniamy na przeciwny stan liczby w S
K8:	*z* ← *z* - *xx*	$z = 3x^2 + y^2$
K9:	**Jeśli** (*z* ≤ *n*) ∧ (*z* **mod** 12 = 7), **to** *S* [*z*] ← ¬ *S* [*z*]	
K10:	Jeśli *x* ≤ *y*, to następny obieg pętli K3	ostatnie równanie sprawdzamy tylko dla x > y
K11:	*z* ← *z* - (*yy* **shl** 1)	$z = 3x^2 - y^2$
K12:	**Jeśli** (*z* ≤ *n*) ∧ (*z* **mod** 12 = 11), **to** *S* [*z*] ← ¬ *S* [z]	
K13:	**Dla** *i* = 5, 6, ..., *g*: **wykonuj** kroki K14...K18	eliminujemy liczby złożone sitem
K14:	**Jeśli** *S* [*i*] = false, to następny obieg pętli K13	
K15:	*xx* = *i* × *i*; *z* ← *xx*	z S eliminujemy wielokrotności kwadratów liczb pierwszych
K16:	Dopóki *z* ≤ *n*, **wykonuj** kroki K17...K18	
K17:	*S* [*z*] ← false	
K18:	*z* ← *z* + *xx*	
K19:	Pisz 2 3	dwie pierwsze liczby wyprowadzamy bezwarunkowo
K20:	**Dla** *i* = 5, 6, ..., *n*: **wykonuj** krok K21	przeglądamy zbiór wypisując znalezione liczby pierwsze
K21:	Jeśli *S* [*i*], to pisz *i*	
K22:	Zakończ	

IMPLEMENTACJA W C++

```
#include <iostream>
#include <cmath>

using namespace std;

int main( )
{
```

```
  unsigned int n, g, x, y, xx, yy, z, i;
  bool * S;

  cin >> n;
  S = new bool [ n+1 ];
  for( i = 5; i <= n; i++ ) S [ i ] = false;
  g = ( unsigned int ) ( sqrt ( n ) );
  for( x = 1; x <= g; x++ )
  {
    xx = x * x;
    for( y = 1; y <= g; y++ )
    {
      yy = y * y;
      z = ( xx << 2 ) + yy;
      if(  ( z <= n ) && (  ( z % 12 == 1 ) || ( z % 12 == 5 ) ) ) S [ z ] = !S [ z ];
      z -= xx;
      if(  ( z <= n ) && ( z % 12 == 7 ) ) S [ z ] = !S [ z ];
      if( x > y )
      {
        z -= yy << 1;
        if(  ( z <= n ) && ( z % 12 == 11 ) ) S [ z ] = !S [ z ];
      }
    }
  }
  for( i = 5; i <= g; i++ )
    if( S [ i ] )
    {
      xx = i * i;
      z = xx;
      while( z <= n )
      {
        S [ z ] = false;
        z += xx;
      }
    }
  cout << 2 << " " << 3 << " ";
  for( i = 5; i <= n; i++ )
    if( S [ i ] ) cout << i << " ";
  cout << endl;
  delete [ ] S;
  return 0;
}
```

IMPLEMENTACJA W PYTHON

```
from math import sqrt
n=int(input("Podaj liczbę n, gdzie n jest końcem przedziału (2, n): "))
S  = [False, False, False, False, False]
for i in range(5, n+1):
    S.append(False)
g = int(sqrt(n))
for x in range(1, g+1):
    xx = x * x
    for y in range(1, g+1):
        yy = y * y
        z = (xx << 2) + yy
        if (z <= n) and ((z%12 == 1) or (z%12 == 5)):
            S[z]=not(S[z])
        z -= xx
        if (z <=n) and (z%12 == 7):
```

```
            S[z]=not(S[z])
        if(x > y):
            z -= yy << 1
            if (z <= n) and (z%12 == 11):
                S[z]=not(S[z])
for i in range(5, g+1):
    if S[i]:
        xx = i*i
        z = xx
        while z<=n:
            S[z] = False
            z += xx
print(2)
print(3)
for i in range(5, n+1):
    if S[i]:
        print(i)
del S
```