2020

Test pierwszości wielkich liczb

METODA NAIWNA I TEST MILLERA-RABINA

MACIEJ BRONIKOWSKI 248838
DANIEL GRZEŚKÓW 218192

1 WSTĘP

Tematem pracy była realizacja testu pierwszeństwa dla dużych liczb. Temat został zrealizowany za pomocą algorytmu naiwnego, oraz testu Millera-Rabina w systemie i386 (32 bitowym), za pomocą asemblera w składni AT&T. Program testowany był na dystrybucji systemu operacyjnego GNU/Linux Ubuntu.

Naiwna implementacja opiera się o bardzo prosty schemat. Należy sprawdzić, czy testowana liczba jest podzielna przez którąkolwiek z liczb z przedziału [2;sqrt(n)]. Należy zaimplementować jedynie algorytm obliczający pierwiastek, oraz resztę z dzielenia dużej liczby.

Test Millera-Rabina w porównaniu do pierwszego algorytmu jest o wiele bardziej skomplikowany, jednak jednocześnie o wiele bardziej efektywny. Algorytm ten jest testem probabilistycznym, którego wynik jest dokładny z prawdopodobieństwem $(1/4)^n$, gdzie n jest ilością przeprowadzonych testów. Opiera się on na twierdzeniu, że jeżeli nieparzysta liczba pierwsza p zostanie zapisana jako $p = 1 + 2^s * d$, gdzie d jest nieparzyste, to dla dowolnej liczby $a \in [2; p-1]$ w ciąg Millera-Rabina $a^d, a^{2d}, a^{4d}, .., a^{2^z d} (mod\ p)$ $a^d \equiv 1 (mod\ n)$ i $a^{2^z d} \equiv -1 (mod\ n)$. Algorytm wymaga implementacji losowania dużych liczb z przedziału $[2; p-1]$, oraz obliczania wartości $a^e mod\ n$ i $(a * a) mod\ n$. Z racji na szybkość działania, test Millera-Rabina jest często wykorzystywany w kryptografii do wyznaczania dużych liczb pierwszych, które są potrzebne podczas generowania kluczy szyfrujących/deszyfrujących.

2 DZIAŁANIE PROGRAMU

Po uruchomieniu programu należy podać liczbę, której pierwszeństwo zostanie sprawdzone, oraz ilość testów Millera-Rabina, która zostanie przeprowadzona. Obie wartości muszą zostać podane w systemie hexadecymalnym, za pomocą znaków {0,1,2,..,9,A,B,C,D,E,D,E,F}:

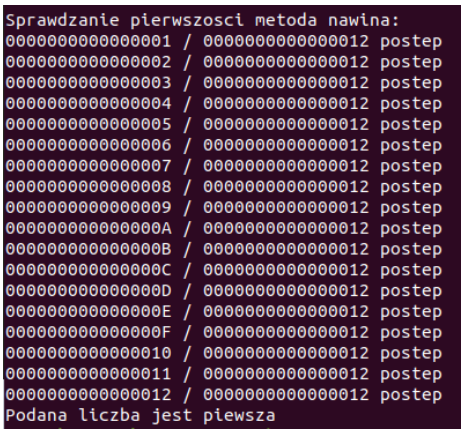
```
jellek@jellek-Virtual-Machine:~/Pulpit/programowanie/AK2-project-gas$ ./prime
Podaj liczbe szesnastkowo (0-9, A-F), ktorej pierwszenstwo chcesz sprawdzic: E8D4A52659
Podaj liczbe szesnastkowo (0-9, A-F), ilości testów <0;FFFFFFFF> (reszta znaków zostanie obcięta): 10
```

Wynikiem jest zapisane hexadecymalnie prawdopodobieństwo poprawności testu Millera-Rabina, oraz komunikat, który informuje użytkownika, czy liczba jest pierwsza, czy też nie, dla poszczególnych algorytmów:

```
Test Millera-Rabina:
Prawdopodobienstwo bledu: 1/4^00000010
Podana liczba jest piewsza

Sprawdzanie pierwszosci metoda nawina:
Podana liczba jest piewsza
```

Dodatkowo podczas testu pierwszeństwa dla naiwnego podejścia, jeżeli liczba jest wystarczająco długa, wyświetlany jest postęp testu (jeżeli pierwiastek z podanej liczby mieści się na więcej niż jednym słowie 32 bitowym). Wynika to z tego, że program może wykonywać test naiwny naprawdę długo, przy tak dużych liczbach. Wyświetlane są wtedy kolejne zmiany licząc od drugiego najmniej znaczącego słowa liczby, przez którą liczba testowana będzie dzielona:



3 IMPLEMENTACJA

3.1 DYNAMICZNA ALOKACJA PAMIĘCI

Z racji, że długość liczby badanej nie jest znana, najlepiej jest przyznać pamięć na poszczególne dane dopiero wtedy, gdy liczba zostanie wprowadzona. Do dynamicznej alokacji pamięci służy funkcja allcate, w pliku allocate.s, której argumentem jest ilość bajtów, które mają zostać zaalokowane. Funkcja zwraca adres pamięci, od którego zaczyna się zaalokowana pamięć. Do dynamicznej alokacji zostało wykorzystane wywołanie systemowe sys_brk, (kod wywołania 45), który przesuwa koniec segmentu danych programu w pamięci komputera. Pierwszym etapem jest pobranie pierwszego adresu za używanym przez program segmentem danych. Jest to możliwe dzięki wczytaniu do rejestru %ebx wartości 0, przed wywołaniem systemowym. Następnie, aby przesunąć wskaźnik końca segmentu danych, do obecnego wskaźnika należy dodać wartość w bajtach, o którą ma on być przesunięty, wczytać tą wartość do rejestru %ebx i ponownie wywołać sys_brk. Funkcja ta wykorzystywana jest również do usunięcia dynamicznie zaalokowanej pamięci. Wystarczy jako ilość bajtów do zaalokowania przekazać wartość ujemną, co spowoduje zmniejszenie ilości pamięci przeznaczonej na dane programu. Należy jednak pamiętać, że działa to na podobnej zasadzie, jak odkładanie elementów na stos. Ostatnio przydzielona pamięć pierwsza zostanie usunięta. Nie można usuwać pamięci ze środka.

3.2 WCZYTYWANIE DANYCH

Do wczytywania danych użyta została funkcja termLoad w pliku termLoad.s. Kod funkcji rozpoczyna się od wypisania informacji o tym, że wprowadzana wartość będzie liczbą, której pierwszeństwo zostanie sprawdzone, oraz, że znaki, którymi można się posługiwać, to (0-9,A-F). Kolejnym etapem jest pobieranie kolejnych znaków, aż do uzyskania znaku końca linii (kod ASCII 10). Kolejne wartość są wstawiane na kolejne pozycje zmiennej lokalnej segment(%ebp) aż do jej zapełnienia, a następnie wartość zmiennej jest odkładana na stos. Kolejne odłożenia są zliczane, aby poznać długość zmiennej, z racji, że nigdzie nie jest podawana jej długość. Po zakończeniu wczytywania długość danych trafia do zmiennej globalnej dataLength, oraz alokowana jest pamięć dla danych i wskaźnik zapisywany jest w zmiennej globalnej dataStartPtr. Dane, które były odłożone na stos, są kolejno zdejmowane i wpisywane na kolejne pozycje wektora od dataStartPtr. Dane należy również przesunąć o odpowiednią ilość w prawo, ponieważ wpisywane były od lewej. Informacja o tym przechowywana jest w zmiennej lokalnej shift(%ebp).

Kolejnym etapem jest wczytanie ilości testów Millera-Rabina, które będą wykonane. Wartość ta mieści się już w pojedynczym słowie 32 bitowym, jednak dane są wczytywane w podobny sposób, jak wartość liczby testowanej. Różnica jest jedynie taka, że wczytany zostanie tylko jeden segment danych. Dane zapisywane są do zmiennej globalnej millerTestCount. Wczytywanie poprzedzone jest komunikatem o tym, że należy wprowadzić ilość testów Millera-Rabina i że wartość musi mieścić się w zakresie <0;FFFFFFFF>.

Funkcja podczas wczytywania znaków sprawdza, czy zostały wprowadzone jakieś dane i czy wprowadzane znaki są poprawne. Jeżeli tak nie jest, odpowiedni komunikat błędu jest wyświetlany, a program kończy swoje działanie.

```
jellek@jellek-Virtual-Machine:~/Pulpit/programowanie/AK2-project-gas$ ./prime
Podaj liczbe szesnastkowo (0-9, A-F), ktorej pierwszenstwo chcesz sprawdzic:
Nie podano zadnej wartosci. Konczenie programu
jellek@jellek-Virtual-Machine:~/Pulpit/programowanie/AK2-project-gas$ ./prime
Podaj liczbe szesnastkowo (0-9, A-F), ktorej pierwszenstwo chcesz sprawdzic: XVBSGBAF
Podano niepoprawny znak. Dozwolone znaki (0-9, A-F). Konczenie programu
jellek@jellek-Virtual-Machine:~/Pulpit/programowanie/AK2-project-gas$
```

3.3 TEST NAIWNY

Naiwny test pierwszeństwa opiera się o prosty algorytm:

```
P – liczba, której pierwszeństwo zostanie sprawdzone
G – pierwiastek z liczby testowanej

G := sqrt(P)
for i := 0, 1 .. G do
  if P mod i = 0 then
    write NO
    halt
  end
end
write YES
```

Jednak z uwagi, że liczba testowana jest dowolnej długości, należy zaimplementować algorytm pierwiastka i dzielenia dużych liczb. Dodatkowo z uwagi, że test może trwać naprawdę długo, zaimplementowany został algorytm postępu testu. Algorytm jest wykonywany przez funkcję naiveAproach w pliku naiveAproach.s.

3.3.1 Obliczanie pierwiastka dużej liczby

Obliczanie pierwiastka zaimplementowane w funkcji root (plik root.s) można przedstawić algorytmem:

```
P – liczba, z której pierwiastek zostanie obliczony
R – wynik
M – maska bitowa

R := 0
M := 0x4000000
while M>P do
  M >>= 2                  --przesunięcie maski w prawo, aż maska będzie mniejsza od P
end
while bit!=0 do
   if P >= R + M than     --jeżeli maska+wynik jest mniejsza lub równa liczbie pierwiastkowanej
      P -= R+M
      R := (R>>1) + M     --przesunięcie wyniku o 1 pozycję w prawo i dodanie maski
   else
      R >>=1              --przesunięcie wyniku o 1 pozycję w prawo
   end
   M >>= 2                --przesunięcie maski bitowej o 2 pozycje w prawo
end
```

Implementacja jednak komplikuje się, ponieważ liczba, której pierwiastek zostanie wyliczony, może być dowolnej długości. Pierwszym elementem jest alokacja pamięci dla wyniku, na którego wskaźnikiem będzie rootStartPtr. Należy go również wyzerować. Następnie wyliczana jest najwyższa pozycja maski bitowej, której wartość będzie mniejsza, niż wartość liczby pierwiastkowanej. Wartość ta przechowywana jest w zmiennej lokalnej bit(%ebp). Z powodu działania algorytmu wczytywania, wiadomo, że wartość maski będzie na pewno ustawiona na najbardziej znaczącym słowie, dlatego test jest wykonywany tylko dla niej. W zmiennej lokalnej bitIndex(%ebp) przechowywany jest indeks słowa 32 bitowego, na którym maska jest ustawiona. Kolejne elementy są już bardzo zbliżone do tych w algorytmie, jednak dodawanie, test większości i przesunięcia bitowe musza być wykonywane dla kolejnych pozycji liczby. (Przesunięcie bitowe w lewo oraz w prawo zostały specjalnie zrealizowane w osobnych funkcjach, ponieważ są często wykorzystywane: shiftOne – przesunięcie o jedną pozycję w prawo i shiftOneLeft – przesunięcie o jedną pozycję w prawo). Wynik zostaje zapisany do zmiennej globalnej rootStartPtr.

3.3.2 Dzielenie wielkich liczb

Są różne algorytmy dzielenia, a standardowym algorytmem stosowanym w ręcznym liczeniu na papierze jest algorytm długiego dzielenia, który zastosowano w projekcie w wersji binarnej dla dużych liczb. Pozwala on na określenie ilorazu oraz reszty z dzielenia. Jego implementacja znajduje się w funkcji bindiv w pliku bindiv.s.

Algorytm wygląda następująco:

```
D – dzielnik
Q – iloraz dzielenia
R – reszta z dzielenia
N – dzielna
n – liczba bitów dzielnej

if D = 0 then error(DzieleniePrzezZero) end
Q := 0              -- Wpisanie do ilorazu wartości zero
R := 0              -- Wpisanie do reszty wartości zero
for i := n − 1 .. 0 do
  R := R << 1       -- Przesunięcie reszty w lewo o 1 bit
  R(0) := N(i)      -- Przepisanie bitu dzielnej z pozycji i do najmniej znaczącego bitu reszty
  if R ≥ D then
    R := R – D
    Q(i) := 1
  end
end
```

Stosując metodę naiwną nie powinno dojść do sytuacji, w której liczba jest dzielona przez 0, więc ten krok można pominąć. W celu określenia, czy liczba ma taki dzielnik, który dzieli bez reszty daną liczbę, wystarczające jest wyliczenie reszty, więc działania dotyczące obliczania ilorazu również można w ostatecznej implementacji pominąć. Do przepisania bitu z odpowiedniej pozycji zastosować można operację logiczną and (koniunkcję) używając odpowiedniej maski.

Do wyliczenia liczby powtórzeń pętli jest użyta długość dzielnej oraz pozycja najbardziej znaczącego bitu w najbardziej znaczącym dwusłowie dzielnej. Wzór jest następujący:
(długość_dzielnej – 1)*32+najbardziej_znaczący_bit.

Pozycja bitu dzielnej do przepisania w danym przejściu pętli jest kontrolowana poprzez przesuwanie maski w prawo. Pozwala to również na określanie, w której części dzielnej bit się znajduje. Jeśli maska po przesunięciu ma wartość równą zero (bit o wartości 1 był na zerowej pozycji i następuje przesunięcie w prawo) to ustawiany jest najbardziej znaczący bit maski na 1, a zmienna lokalna określająca rozpatrywaną część dzielnej jest inkrementowana).

Do przesuwania bitowego dużych liczb (zajmujących więcej 32 bity w pamięci) wykorzystywane są własne funkcje korzystające z wykrywania przeniesienia po wykonaniu systemowej instrukcji przesunięcia.

Funkcja przeprowadzająca dzielenie ma 6 argumentów:
a) adres do obszaru pamięci, w którym jest zapisany dzielnik,
b) długość (liczba użytych podwójnych słów – 32 bitowych komórek) dzielnika,
c) adres reszty,
d) długość reszty
e) adres dzielnej,
f) długość dzielnej.

Sekwencja operacji w funkcji dzielenia:

D – dzielna
N – dzielnik
R – reszta
M – maska
J – licznik wskazujący na rozpatrywaną część dzielnej
MSB – pozycja najbardziej znaczącego bitu
Rtemp – pomocnicza reszta, by móc wykonać operacje nie zmieniając R

1. Wyznaczenie MSB w najbardziej znaczącej części dzielnej.
2. Wyzerowanie R.
3. Ustawienie bitu M(MSB) na 1.
4. Wyliczenie liczby powtórzeń pętli.
5. Wpisanie wartości 0 do J.
6. Pętla od obliczonej liczby powtórzeń do 0 włącznie:
a) Przesunięcie bitowe R w lewo o 1.
b) Koniunkcja M z D(J).
c) Przesunięcie M w prawo.
d) Jeśli M równe 0 to wpisanie 0x80000000H do M i inkrementacja J.
e) Wpisanie wartości R do Rtemp.
f) Rtemp = Rtemp – N.
g) Jeśli Rtemp jest większe lub równe 0 to wpisanie wartości Rtemp do R.
7. Sprawdzenie, czy R jest równe 0. Jeśli tak, to ustawienie wyjścia funkcji na 0, a jeśli nie to ustawienie wyjścia na 1.

3.3.3 Wyświetlanie postępu testu

Postęp testu wyświetlany jest w bardzo prosty sposób. Przy każdej zmianie słowa 32-bitowego na drugiej najmniej znaczącej pozycji wywoływana jest funkcja progress z pliku progress.s. Dla każdego słowa 32-bitowego pomijając najmniej znaczące uruchamiana jest funkcja displayNumber z pliku displayNumber.s, której argumentem jest liczba do wyświetlenia. Funkcja displayNumber za pomocą przesunięć w lewo i w prawo, wyodrębnia kolejne słowa 4 bitowe licząc od lewej, następnie zamienia je na wartość kodu ascii (dla wartości 0-9 należy dodać 48, a dla wartości A-F 54), a na końcu wyświetla znak w terminalu. Wyświetlana informacja ma format: $a_n…a_{10}a_9a_8$ / $b_n…b_{10}b_9b_9$ – postęp, gdzie a – kolejne wartości hexadecymalne liczby przez którą testowana wartość jest dzielona, oraz b - kolejne wartości hexadecymalne liczby testowanej.

3.3.4 Przebieg testu

Przebieg testu jest już bardzo zbliżony do algorytmu przedstawionego na samym początku. Jednak dla przyspieszenia działania programu, wykonywany jest najpierw test, czy liczba nie jest podzielna przez 2, a następnie, jeżeli warunek nie jest spełniony, liczba dzielona jest przez liczby 3,5,7,…,sqrt(n). Jeżeli nie jest podzielna przez żadną z wymienionych liczb, wyświetlany jest komunikat o tym, że liczba jest pierwsza, w przeciwnym wypadku komunikat informuje o tym, że liczba nie jest pierwsza.

3.4 TEST MILLERA-RABINA

Test Millera-Rabina został zrealizowany w funkcji millerRabin w pliku millerRabin.s. Algorytm działania funkcji można przedstawić następująco:

```
P = 1+2^S*D
P – nieparzysta liczba, której pierwszeństwo jest testowane
N – ilość testów Millera-Rabina
S – wykładnik potęgi 2 dzielnika P-1
D – mnożnik potęgi 2 dzielnika P-1
A – baza ciągu Millera-Rabina
X – kolejne wyrazy ciągu Millera-Rabina
J – liczba wyliczonych wartości ciągu Millera-Rabina

D := P-1
S := 0
while D != 0 do
   S := S + 1
   D := D / 2      -wyliczanie wartości niewiadomych wyrażenia P = 1 + 2^S * D
end

for i := 1,2,..,n do
   A := Losowa(2,P-2)   -- losowanie liczby z przedziału [2;P-2]
   X := A^D mod P       -- pierwszy wyraz ciąg Millera-Rabina
   if X = 1 or X = P-1 then
      continue          -- pierwszy wyraz ciągu jest ostatnim, należy wybrać inne A
   end
   while J < S i X != P-1 do
      X := (X*X) mod P
      if X = 1 than     -- tylko ostani wyraz ciągu Millera-Rabina jest równy 1
         write NO
         halt
      end
      j := j + 1
   end
   if X != P – 1 than
      write NO          -- przedostatni wyraz ciągu musi być równy P - 1
      halt
   end
end
write YES               -- wszystkie test wykonane pomyślnie
```

Jak widać powyżej, algorytm ten jest o wiele bardziej skomplikowany i wymaga implementacji wielu nowych algorytmów takich jak: generowanie liczny pseudoloswej w podanym przedziale, obliczanie X^e mod n, oraz obliczanie (X*X) mod n.

3.4.1 Generowanie liczby pseudolosowej

W generowaniu liczby pseudolosowej towarzyszą trzy funkcje.

Funkcja random w pliku random.s wylicza liczbę pseudolosową z przedziału [0;FFFFFFFF] dla każdego słowa 32 bitowego o podanej długości. Argumentami są wskaźnik na początek pamięci, w której zostanie wygenerowane liczby, oraz ilość liczb do wygenerowania. Do generowania liczb wykorzystany został algorytm Linear congruential generator, który bazuje na rekurencyjnej funkcji:

$$X_{n+1} = (aX_n + c) \bmod m$$

Jakość wygenerowanych liczb zależy od doboru parametrów a, c i m. Użytymi w projekcie danymi były:

$$a = 1103515245,\ c = 12345,\ m = 2^{32}$$

taki dobór wartości m pozwala na przeprowadzanie operacji modulo niejawnie, po prostu ignorując rejestr zawierający wyższe 32 bity wyniku działania.

Do wygenerowania wartości X_0 służy funkcja srand, zaimplementowana w pliku srand.s. Na podstawie czasu systemowego pobieranego za pomocą wywołania sys_time (kod wywołania 13) ustawiane jest ziarno (zmienna globalna seed), które posłuży jako argument pierwszego wyniku liczby pseudolosowej.

Dla zapewnienia mieszczenia się w zakresie użyć można napisanej przez nas funkcji dzielenia dużych liczb. Osiągnięta losowa liczba to dzielna, a górny zakres to dzielnik – reszta dzielenia na pewno będzie mniejsza niż dzielnik.

Za odpowiednie wywołanie kolejnych funkcji odpowiada funkcja lessrand z pliku lessrand.s, której argumentami jest wskaźnik na wynik losowania, oraz dzielnik, który będzie ograniczał wynik losowania.

3.4.2 Wyznaczanie (a*b) mod n

Funkcja modMul w pliku modMul.s realizuje algorytm wyznaczania reszty z dzielenia liczby a*b. Argumentami są 4 wskaźniki: argument a, argument b, argument n, oraz wynik reszty z dzielenia. Wykorzystany algorytm wygląda następująco:

```
W – wynik mnożenia modulo
M – maska bitowa
A – argument a mnożenia modulo
B – argument b mnożenia modulo
N – argument n mnożenia modulo

W := 0
M := 1
while M != 0 do
  if (B & M) != 0 than      -- Czy dany bit mnożnika ma wartość 1
    W := (W + A) mod N
  end
  A := (A<<1) mod N         -- przesunięcie mnożnej o 1 bit w lewo
  M := M<<1                 -- przesunięcie maski o 1 bit w lewo
end
```

Testowane są kolejne bity mnożnika i jeżeli wartość danego bitu jest równa 1 to o danej wadze mnożnik jest dodawany do wyniku. Implementacja tego algorytmu dla dużych liczb nie jest dużym problemem. Wystarczy zadbać, aby dodawanie, przesunięcia bitowe i test równości były wykonywane dla kolejnych słów 32 bitowych z użyciem przeniesień.

3.4.3 Wyznaczanie a^e mod n

Funkcja modPower w pliku modPowe.s, wylicza resztę z dzielenia potęgi liczby o dowolnym wykładniku. Argumentami funkcji są, a – podstawa, b – wykładnik, n – dzielna do obliczenia reszty, oraz wskaźnik na wynik. Algorytm użyty w implementacji można przedstawić następująco:

```
W – wynik potęgi modulo
M – maska bitowa
A – argument a mnożenia modulo
E – argument e mnożenia modulo
N – argument n mnożenia modulo
P – reszta kolejnych potęg modulo n

P := A                    -- wartość początkowa potęgi a^1
M := 1
W := 1
while M != 0 do
  if (E & M) != 0 than    -- Czy dany bit wykładnika ma wartość 1
    W := (W * P) mod N    -- jeżeli tak, to wynik mnożymy przez potęgę
  end
  P := (P * P) mod N      -- Wartość kolejnej potęgi
  M := M<<1               -- przesunięcie maski o 1 bit w lewo
end
```

Algorytm korzysta z faktu, że wykładnik potęgi modulo można rozłożyć na iloczyn wielu mniejszych potęg modulo. W ten sposób przykładowo a^{52} mod n można zapisać:

$$a^{52} mod\ n = (a^4\ mod\ n * \ a^{16}\ mod\ n * \ a^{32}\ mod\ n)\ mod\ n$$

Implementacja algorytmu dla wielkich liczb, podobnie jak w przypadku wyliczania iloczynu modulo, nie wymaga dużych modyfikacji. Funkcja wyliczania mnożenia modulo została przygotowana wcześniej. Należy jedynie zaimplementować przesunięcie w lewo, oraz test równości dla liczb składających się z wielu słów 32 bitowych. Pierwsze z wymienionych jest już zaimplementowane w funkcji shiftOneLeft.

3.4.4 Przebieg testu

Przebieg testu jest również bardzo zbliżony do algorytmu, który był przedstawiony na samym początku punktu 3.4. Należy jednak zwrócić szczególną uwagę na test większości/mniejszości i dodawanie/odejmowanie liczb. Jednak przykładowo dla etapu obliczania D:=P-1 operacje są wykonywane jedynie na najmniej znaczącym słowie 32 bitowym. Wynika to z faktu, że na tym etapie jesteśmy pewni, że liczba jest nieparzysta, co za tym idzie, odjęcie 1 wymaga od nas tylko zmiany ostatniego bitu na 0. Aby zapewnić warunek, że liczba jest nieparzysta, na początku sprawdzane jest, czy liczba nie jest podzielna przez 2. Użytkownik informowany jest o tym, z jaką dokładnością test zostanie przeprowadzony, a następnie, jeżeli wszystkie testy zostały wykonane pomyślnie, zwrócony zostanie komunikat o tym, że liczba jest pierwsza. W przeciwnym wypadku komunikatem będzie informacja, że liczba nie jest pierwsza.

Z racji tego, że test Millera-Rabina jest testem probabilistycznym, nigdy nie można być pewnym, że liczba jest w 100% pierwsza. O liczbach, które pomyślnie mogą przejść test Millera-Rabina mówi się, że są silnie pseudopierwsze. Zachowanie takie można zauważyć, gdy po uruchomieniu programu zostanie wpisana liczba testowana 15, oraz ilość testów 1. Wtedy liczba ta przejdzie pomyślnie 1 na 4 testy:

Wynik poprawny:

```
jellek@jellek-Virtual-Machine:~/Pulpit/programowanie/AK2-project-gas$ ./prime
Podaj liczbe szesnastkowo (0-9, A-F), ktorej pierwszenstwo chcesz sprawdzic: 15
Podaj liczbe szesnastkowo (0-9, A-F), ilości testów <0;FFFFFFFF> (reszta znaków zostanie obcięta): 1

Test Millera-Rabina:
Prawdopodobienstwo bledu: 1/4^00000001
Podana liczba nie jest piewsza

Sprawdzanie pierwszosci metoda nawina:
Podana liczba nie jest piewsza
```

Wynik błędny:

```
jellek@jellek-Virtual-Machine:~/Pulpit/programowanie/AK2-project-gas$ ./prime
Podaj liczbe szesnastkowo (0-9, A-F), ktorej pierwszenstwo chcesz sprawdzic: 15
Podaj liczbe szesnastkowo (0-9, A-F), ilości testów <0;FFFFFFFF> (reszta znaków zostanie obcięta): 1

Test Millera-Rabina:
Prawdopodobienstwo bledu: 1/4^00000001
Podana liczba jest piewsza

Sprawdzanie pierwszosci metoda nawina:
Podana liczba nie jest piewsza
```

4 HARMONOGRAM PRACY

1. Stworzenie pliku makefile
   W pliku zawarty jest skrypt, który automatycznie łączy wszystkie pliki .s zawarte w folderze src i na ich podstawie generuje najpierw pliki obiektów .o, a następnie plik wynikowy prime.
2. Implementacja wczytywania danych z terminala i dynamiczna alokacja pamięci
   Prawdopodobnie element ten zajął nam najwięcej czasu, z uwagi na początki nauki asemblera. Bardzo pomocne okazały się materiały
   https://en.wikipedia.org/wiki/X86_instruction_listings - spis instrukcji procesora
   http://unixwiz.net/techtips/x86-jumps.html - porównania signed/unsigned
   https://syscalls.kernelgrok.com/ - spis wywołań systemowych Linux (niestety link chwilowo jest nieaktywny)
   https://baptiste-wicht.com/posts/2011/11/dynamic-memory-allocation-intel-assembly-linux.html?fbclid=IwAR1c7-6X86Cb57plnFbFrJfZFjHJ3keyqWE6qOsVCTRBgz1zYj-Rh7_F2wo – przykład dynamicznej alokacji pamięci za pomocą sys_brk
3. Obliczanie pierwiastka podanej liczby
   pomocne materiały:
   https://en.wikipedia.org/wiki/Methods_of_computing_square_roots - algorytmy obliczania pierwiastka kwadratowego
4. Wyznaczanie reszty z dzielenia dużych liczb
   pomocne materiały:
   https://en.wikipedia.org/wiki/Division_algorithm - algorytmy obliczania wyniku dzielenia
5. Implementacja naiwnego testu pierwszości
   pomocne materiały:
   https://eduinf.waw.pl/inf/alg/001_search/0016.php

6. Implementacja testu Millera-Rabina
   Pomocne materiały:
   https://eduinf.waw.pl/inf/alg/001_search/0019.php - algorytm testu Millera-Rabina
   https://pl.wikipedia.org/wiki/Test_Millera-Rabina - algorytm testu Millera-Rabina
   https://eduinf.waw.pl/inf/alg/001_search/0017.php - algorytm wyznaczania wartości mnożenia modulo, oraz potęgi modulo.
   https://wiki.osdev.org/Random_Number_Generator?fbclid=IwAR0vR4awybxiL5-VFupUdmKORqIsE0gNTcXVVjWSnKun1NuFI4LDC-OHUko – wyznaczanie liczby pseudolosowej
7. Testowanie algorytmów za pomocą narzędzia gdbtui