KATEDRA INFORMATYKI TECHNICZNEJ
ZAKŁAD ARCHITEKTURY KOMPUTERÓW

ORGANIZACJA I ARCHITEKTURA KOMPUTERÓW
SPRAWOZDANIE PROJEKTOWE

# ROZKŁAD LICZBY NA CZYNNIKI PIERWSZE

STANISŁAW FRANCZYK, 248857

PROWADZĄCY - PROF. DR HAB. INŻ. JANUSZ BIERNAT

DN. 10 CZERWCA 2020

# Spis treści

# 1 Cele projektu

Celem projektu była implementacja skutecznego algorytmu rozkładu na czynniki pierwsze dla liczb 64 bitowych oraz liczb posiadających więcej niż 64 bity.

# 2 Wstęp

Faktoryzacja to zapisanie danej liczby naturalnej przez iloczyn jej czynników pierwszych. Jest to zgodne z zasadniczym twierdzeniem arytmetyki. Na dzień dzisiejszy nie istnieje powszechnie znany żaden szybki algorytm rozkładu dużych liczb pierwszych. Jednymi z wielu przykładów algorytmów rozkładu na czynniki pierwsze są:

- Bezpośrednie poszukiwanie rozkładu na czynniki pierwsze - przez sprawdzanie podzielności przez kolejne liczby pierwsze. Wadą rozwiązania jest wymóg posiadania zapisanych wszystkich liczb pierwszych co najmniej do pierwiastka z rozkładanej liczby.
- Algorytm wykorzystujący metodę Fermata.
- Sito kwadratowe.
- Algorytm Rho Pollarda.

W do implementacji w projekcie wybrany został algorytm Rho Pollarda.

Z powodu założenia, że program ma rozkładać na czynniki pierwsze, aby zapisać liczbę posiadającą więcej niż 64 bity długości w systemie binarnym potrzebny był sposób jej przechowywania. Założono że liczba jest przechowywana w tablicy liczb typu **unsigned char**, a jej długość jest zapisana w zmiennej typu **long int**. Element tablicy o indeksie 0 ma największą wagę, kolejne elementy posiadają niższe wagi. Liczba faktoryzowana jest więc reprezentowana przez liczbę o podstawie 256 i zapisanej w określonej zmiennej liczbie cyfr.

## 2.1 Algorytm Rho Pollarda

Jest algorytmem rozkładu liczby na czynniki pierwsze opracowanym przez Johna Pollarda w 1975 roku. Jest on szczególnie efektywny przy rozkładaniu liczb mających niewielki dzielniki. Algorytm stał się sławny po tym kiedy udało się za jego pomocą zfaktoryzować ósmą liczbę Fremata. Było to głównie spowodowane tym, że jeden z czynników tej liczby jest znacznie mniejszy niż drugi.

## 2.2 Działanie algorytmu Rho Pollarda

Algorytm w wyszukiwaniu liczb pierwszych wykorzystuje paradoks dnia urodzin. Aby zanieść dwie liczby $x$ i $y$ przystające modulo $p$ z prawdopodobieństwem większym niż $\frac{1}{2}$ wystarczy wylosować mniej więcej $1,177\sqrt{p}$ liczb. Jeśli $p$ jest szukanym dzielnikiem n to $NWD(|x-y|,n) > 1$, ponieważ $n$ i $|x-y|$ są podzielne prze $p$. Z tego powodu wystarczy losować kolejne liczby i sprawdzać, czy któraś różnica ma nietrywialne wspólne dzielniki z n.

Algorytm nie zapamiętuje wszystkich wylosowanych liczb i nie sprawdza każdej pary. Zamiast tego wykorzystuje metodę cyklu funkcji (cykl Floyda). Wybierana jest pseudolosowa funkcja modulo $n$ jako generator dwóch sekwencji. W czasie wykonania jednej iteracji pierwszej sekwencji druga wykonuje dwie iteracje. Pierwszą z sekwencji zwykle oznacza się literą $x$, natomiast drugą literą $y$. Następnie w każdym kroku sprawdzane jest $NWD(|x-y|,n) > 1$. Jeśli wynik jest równy n, algorytm kończy się błędem, ponieważ wtedy $x = y$ i dalsze powtarzanie kroków będzie powtarzaniem poprzednich obliczeń. Jeśli jednak wynik jest większy od 1 i mniejszy od n jest on dzielnikiem $n$.

Owo podwójne iterowanie jednej z sekwencji jest czasem obrazowane, jako żółwia i zająca startujących z jednego miejsca na owalnej bieżni, przy czym zając biegnie dwa razy szybciej od żółwia. Jeśli będą oni biegli ze stałymi prędkościami ponownie zrównają się w miejscu startu.

Sekwencja modulo szukanego dzielnika $p$ zawsze tworzy cykl. Diagram sekwencji przypomina grecką literę $\rho$, stąd nazwa algorytmu.

### 2.3 Problemy algorytmu

Z algorytmem Rho Pollarda związane są dwa zasadnicze problemy. Jeśli otrzyma do rozkładu na czynniki pierwsze liczbę pierwszą algorytm nigdy nie zakończy działania. Algorytm potrafi też zwracać liczby nie pierwsze jeśli dzielnikami liczby $n$ są też liczby złożone. Z tego powodu dodatkowo w programie jest wymagany algorytm sprawdzający pierwszość liczby.

## 3 Dodatkowe algorytmy

Algorytm do obliczenia największego wspólnego dzielnika (NWD) został napisany na podstawie klasycznego algorytmu Euklidesa w wersji modularnej.

Sprawdzenie pierwszości liczby zostało rozwiązane w następując sposób:

- W pierwszym programie rozkładającym liczbę pierwszą zapisaną w formacie **long long int** w celu sprawdzenia pierwszości danej liczby sprawdza się czy jest ona podzielna przez kolejne wielokrotności liczby 6, aż do wartości pierwiastka tej liczby. Jest to lekko zmodyfikowany trywialny algorytm naiwny.
- W drugim programie w celu sprawdzenia pierwszości liczby został zaimplementowany algorytm *chińskiego testu pierwszości.*

## 4 Wykorzystane narzędzia

Programy projektowe były pisane w językach *C++* i w języku assemblera x86 w konwencji AT&T.

Przy pisaniu kodu funkcji assemblerowych wykorzystywano *Visual Studio Code* na systemie operacyjnym Ubuntu. Do utworzenia całego projektu wykorzystano program *CLion* na komputerze o większej mocy z systemem operacyjnym Windows.

## 5 Przebieg projektu

Po wybraniu tematu i algorytmu napisany został program wykonujący algorytm Rho Pollarda dla liczb typu **long long int**. Program jest napisany w paradygmacie obiektowym. Została napisana specjalna klasa *RhoPollardsAlgorithm* która agreguje w sobie wszystkie funkcje. W funkcji głównej *int main()* jest wpisana liczba która ma zostać poddana faktoryzacji - liczba **n**, tablica agregująca uzyskane liczby pierwsze oraz funkcja, która najpierw dzieli daną liczbę **n** przez liczbę 2, najmniejszą znaną liczbę pierwszą, a następnie wykonuje na niej algorytm faktoryzacji, pobiera liczbę pierwszą i dzieli przez nią liczbę, dopóki otrzymana dzięki dzieleniu liczba nie będzie liczbą pierwszą. W ten sposób można uzyskać wszystkie czynniki pierwsze danej liczby **n**. Niestety program działa tylko dla liczb mniejszych od liczby 9 223 372 036 854 775 808 ($2^{63}$) co jest spowodowane ograniczeniami zmiennej typu **long long int**, która jest zapisywana na 64 bitach.

Następnym krokiem było napisanie algorytmu dla liczb większych. W tym celu został napisany szereg funkcji assemblerowych do obsługi programu. W pierwszej kolejności napisano funkcję **fullChar**, która odczytuje wartość liczby z ciągu kodów znaków danej liczby i zapisuje ją w tablicy jako liczbę o podstawie $2^8$ (jeden bajt). Dzięki temu oszczędzane jest miejsce i potem liczba wykonanych obliczeń, co poprawia szybkość wykonania algorytmu. Następnie napisana została funkcja odkodowania **fullCharToString** i potem pozostałe funkcje potrzebne do wykonania algorytmu. W końcu stworzono projekt programu i napisano wersję programu dla dużych liczb. W dalszej części sprawozdania elementy tablicy będą nazywane *cyframi*.

Potem próbowano jeszcze bardziej zoptymalizować kod tworząc projekt który miał zamieniać ciąg znaków na bloki 64 bitowe, ale zaprzestano tego po tym jak odkryto, że przez sposób w jaki realizowane jest dzielenie, czyli wielokrotne odejmowanie, przy dzieleniu przez niewielkie liczby lub liczby które posiadają niewielką "cyfrę" wiodącą działanie to bardzo się wydłuża. Np. Przy dzieleniu liczby $(1 \cdot 2^{64} + 2^{64} - 1)$ przez liczbę 9 należało by wykonać 4099276460824344803 odejmowań. Natomiast dla drugiej wersji programu wykonywane są 1133 odejmowania. Zaprzestano więc wdrażania trzeciego programu i skupiono się na naprawie błędów w poprzednim.

Ostatecznie dokonano pomiarów porównawczych pierwszego i drugiego programu, oraz pomiarów dużych liczb dla drugiego programu.

# 6 Pierwszy program

## 6.1 Klasa implementująca algorytm Rho Pollarda

Zawiera w sobie 6 metod:

1. long long **get_prime**(long long *number*) która zwraca czynnik liczby przekazanej
2. long long **f**(long long $x$, long long $c$) która jest funkcją generującą pseudolosowo liczby modulo n
3. long long **GCD**(long long $a$, long long $b$) która zwraca największy wspólny dzielnik, w programie używa zaimplementowanej metody **EuclideanAlgorithm**
4. bool **priorityTest**(long long $x$) która sprawdza czy liczba jest liczbą pierwszą, w programie używa zaimplementowanej metody **primary6_k_1**
5. bool **primary6_k_1**(long long $x$) która naiwnie sprawdza czy liczba jest liczbą pierwszą
6. long long **EuclideanAlgorithm**(long long $a$, long long $b$) która zwraca NWD wyliczając go zgodnie z algorytmem Euklidesa

Poniżej znajduje się zaimplementowany w funkcji **get_prime** algorytm Rho Pollarda.

```
/**Algorytm Rho Pollarda**/
long long RhoPollardsAlgorithm::get_prime(long long number)
{
    n = number;
    // Pierwsza sekwencja
    long long int x = (rand()%(n-2))+2;
    // Druga sekwencja
    long long int y = x;

    long long int d = 1;

    // Dodatkowa stała do funkcji
    long long int c = (rand()%(n-1))+1;

    while( d == 1)
    {
        // Pierwsza sekwencja
        x = f(x, c);
        // Druga sekwencja f(f(x))
        y = f(f(y, c), c);
        // Sprawdzenie podzielności
        d = GCD( llabs(x - y), number);
    }

    // Jeśli wyznaczony podzielnik n jest liczbą złożoną należy ją także rozłożyć
    if(!priorityTest(d))
        d = get_prime(d);

    // Sprawdzenie czy wyznaczona liczba nie jest liczbą zadaną
    return d == number ? get_prime(d) : d;
}
```

# 7 Drugi program

## 7.1 Funkcje assemblerowe

### 7.1.1 Funkcja fullChar

Po przekonaniu się o działaniu poprzedniego programu rozpoczęto pisanie programu faktorującego większe liczby. W celu przyspieszenia programu napisano w assemblerze funkcję **charFull**, która zamieniała ciąg kodów znaków char liczby w zapisie dziesiętnym na tablicę bajtów zawierającej wartość danej liczby. Przykładowo liczbę 140519 funkcja przekształci na tablicę 2, 36, 231. Tablicę można następnie przekształcić: $2 \cdot 256^2 + 36 \cdot 256 + 231 = 140519$.

Funkcja przyjmuje jako parametr wskaźnik na tablicę znaków, wynik konwersji zapisuje w tej samej tablicy, zwraca długość tablicy.

```
countsub2:
  movq %rbx, %rsi
  movq $3, %rcx
  clc

  subtract2:                # Dzielenie przez odejmowanie
    decq %rsi
    decq %rcx
    movb cbase(, %rcx, 1), %al      # cbase[] = {2, 5, 6}
    sbbb %al, (%r8, %rsi, 1)
    jnc then3
      cmpq %rdi, %rsi
        je checkcredit
      addb $BASE, (%r8 , %rsi, 1)   # BASE = 256
      stc
      jmp subtract2
    then3:
      cmpq %rdi, %rsi
        jg subtract2
      incb %dl              # Zwiększ licznik
      jmp countsub2

  checkcredit:              # Sprawdzenie czy można uzyskać pożyczkę z poprzedniej cyfry
    dec %rsi
    movb (%r8 , %rsi, 1), %al
    cmpb $0, %al
      jng restore           # Jeśli nie można uzyskać pożyczki należy cofnąć działanie

  iscredit:                 # Można uzyskać pożyczkę
    decb (%r8 , %rsi, 1)    # to zmniejszenie cyfry o 1 (wartość pożyczki)
    inc %rsi
    addb $BASE, (%r8 , %rsi, 1)
    incb %dl
    jmp countsub2
```

Listing 1: Fragment funkcji **fullChar**

### 7.1.2 Funkcja zamieniająca fullChar na string

Do prezentacji wyników wymagana była funkcja zwracająca ciąg kodów znaków otrzymanej liczby.

```
movb cbase, %bl          # .byte cbase = 10
divfullChar:
  movb (%r8, %rsi, 1), %al  # Konwersja
  div %bl
  movb %al, (%r8, %rsi, 1)
  inc %rsi
  cmpq %rsi, %r9
    jg divfullChar

  movb %ah, tab(, %rdi, 1)  # Kolejna reszta z dzielenia
  inc %rdi
```

```

  cmpb $0, (%r8, %r10, 1)
    jne dalej

  incq %r10

dalej:
  cmpq %r10, %r9    # Koniec konwersji jeśli w liczbie pozostało tylko zero
    je end

  xorq %rax, %rax
  movq %r10, %rsi

  jmp divfullChar

end:
  xor %rsi, %rsi
  mov %rdi, %rax

toChar:             # Zamiana wartości cyfr na kody ich znaków
  dec %rdi          #    jednocześnie odwracanie kolejności ustawienia cyfr
  movb tab(, %rdi, 1), %dl
  orb $0x30, %dl    # Tutaj zmiana wartości cyfry na kod znaku
  movb %dl, (%r8, %rsi, 1)
  inc %rsi
  cmpq $0, %rdi
    jg toChar


  movb $0, (%r8, %rsi, 1)
  inc %rax          # W RAX długość ciągu znaków
```

Listing 2: Fragment funkcji **fullCharToString**

### 7.1.3 Funkcja dzieląca liczby

W celu dzielenia wielkich liczb została napisana funkcja **divFullChar**, która przyjmuje jako wartości parametry w tej kolejności: wskaźnik na początek tablicy liczby dzielnika, długość dzielnika, wskaźnik na początek tablicy liczby dzielnej, długość tablicy dzielnej, wskaźnik na tablicę wyniku. Funkcja zwraca długość wyniku. Zamienia także liczbę dzielną na resztę z dzielenia dzielnej przez dzielnik. Przy tym w reszcie z dzielenia bardzo często pierwsze elementy tablicy posiadają wartość zera. Jeśli istnieje potrzeba zachowania dzielnej, należy ją najpierw skopiować, jeśli zależy nam na reszcie z dzielenia należy po dzieleniu usunąć zera wiodące, do czego służy kilejna funkcja **femovezeros**. Dzielenie jest zrealizowane przez wielokrotne odejmowanie.

```
restore:                        # Przywrócenie ostatniego odejmowania
  movq %rbx, %rsi
  movq %r10, %rcx
  decq %rcx
  cmpq $0, %rcx
    je afterRes
  clc

  addlastcredit:
    decq %rsi
    movb (%rdi , %rcx, 1), %al
    adcb %al, (%rdx , %rsi, 1)
    loop addlastcredit
    afterRes:
    decq %rsi
    movb (%rdi , %rcx, 1), %al
    adcb %al, (%rdx , %rsi, 1)

slide:                          # Skalowanie dzielnika
  movb %r11b, (%r8, %r12, 1)

  xorq %r11, %r11
  incq %r12
```

```
  incq %rbx
  cmpq %rbx, %r13
    jl out

countsub2:                        # Liczenie odejmowań
  movq %rbx, %rsi
  movq %r10, %rcx
  decq %rcx
  cmpq $0, %rcx
    je after2
  clc

  subtract2:                      # Odjęcie liczby
    decq %rsi
    movb (%rdi , %rcx, 1), %al
    sbbb %al, (%rdx , %rsi, 1)   # Odjęcie cyfry
  loop subtract2
  after2:
  decq %rsi
  movb (%rdi , %rcx, 1), %al
  sbbb %al, (%rdx , %rsi, 1)
  jc checkcredit
  incb %r11b
  jmp countsub2

  checkcredit:                    # Sprawdzenie czy można uzyskać pożyczkę
    decq %rsi
    movb (%rdx , %rsi, 1), %al
    cmpb $0, %al
      je restore

  iscredit:
    decb (%rdx , %rsi, 1)
    incb %r11b
    jmp countsub2

out:
  movq %r12, %rax                 # Długość reszty z dzielenia
```

Listing 3: Fragment funkcji **divFullChar** bez pierwszego odejmowania przy którym nie ma pożyczki, liczby w rejestrarch RDI (dzielnik), RDX (dzielna/reszta z dzielenia), w rejestrze R8 wynik

### 7.1.4 Funkcja redukująca liczby

W wyniku niektórych działań ”wiodącymi cyframi” liczb były zera. W celu ich redukcji powstała funkcja **removezeros**. Funkcja przyjmuje za parametry wskaźnik na liczbę i wartość długości liczby. Przenosi wszystkie ”cyfry” występujące za zerami i zwraca nową wartość długości liczby.

### 7.1.5 Funkcja dodająca liczby

W celu dodawania liczb napisana została funkcja **addFullChar**. Przyjmuje ona 5 parametrów: wskaźnik na pierwszą liczbę, jej długość, wskaźnik na drugą liczbę, długość drugiej liczby i wskaźnik na miejsce w którym ma zostać zapisana liczba wyniku. Funkcja najpierw wyszukuje która z liczb jest mniejsza, następnie umieszcza adresy liczb w odpowiednich rejestrach. Potem funkcja przenosi wartość sumy kolejnych *cyfr* obu liczb do odpowiedniego miejsca *cyfry* w liczbie wyniku. Kiedy mniejsza z liczb nie ma już kolejnych *cyfr* do dodania funkcja przenosi kolejne *cyfry* większej liczby wraz z dodaniem przeniesień. Jeśli nastąpiło przepełnienie ostatecznie funkcja przenosi wszystkie *cyfry* o jedno miejsce, a w miejscu *cyfry wiodącej* wstawia wartość 1. Funkcja zwraca długość liczby wyniku.

### 7.1.6 Wartość bezwzględna z różnicy liczb

W algorytmie Rho Pollarda wymagana była funkcja obliczająca wartość bezwzględną różnicy liczb. W tym celu napisana została funkcja **absFullChar**. Przyjmuje ona 5 parametrów: wskaźnik na pierwszą liczbę, jej długość, wskaźnik na drugą liczbę, długość drugiej liczby i wskaźnik na miejsce w którym ma zostać zapisana liczba wyniku. Funkcja najpierw wyszukuje która z liczb jest mniejsza, następnie umieszcza adresy liczb w odpowiednich rejestrach. Potem funkcja w pętli przenosi wartość *cyfry* większej liczby do odpowiedniego miejsca na *cyfrę* w liczbie wyniku, a następnie odejmuje od niej *cyfrę* drugiej mniejszej liczby wraz z pożyczką. Jeśli liczba mniejsza zużyje już wszystkie *cyfry* funkcja przenosi kolejne *cyfry* liczby większej do liczby wyniku odejmując pożyczkę. Ostatecznie funkcja wywołuje funkcję **removezeros** na liczbie wyniku i zwraca długość otrzymanej liczby.

### 7.1.7 Funkcja mnożąca liczby

Do mnożenia liczb napisana została funkcja **mulFullChar**. Przyjmuje ona 5 parametrów: wskaźnik na pierwszą liczbę, jej długość, wskaźnik na drugą liczbę, długość drugiej liczby i wskaźnik na miejsce w którym ma zostać zapisana liczba wyniku. Funkcja najpierw wylicza możliwą długość liczby wyniku (długość pierwszej liczby + długość drugiej liczby) i zapełnia całą tablicę zerami. Następnie następuje mnożenie *cyfr* drugiej liczby przez pierwszą liczbę, wyniki są kolejno dodawane do liczby wyniku. Funkcja zwraca długość wyniku.

### 7.1.8 Funkcja iloczynu logicznego

W algorytmie *Chińskiego testu pierwszości* jest wykorzystywana dodatkowa funkcja. Powinna ona wykazywać czy mnożenie logiczne (operacja **AND**) liczb zwraca wartość różną od zera. W tym celu napisano funkcję **andFullChar**. Przyjmuje ona 4 parametry: wskaźnik na pierwszą liczbę, jej długość, wskaźnik na drugą liczbę i długość drugiej liczby. Funkcja sprawdza czy kolejne wykonywanie działań mnożenia logicznego na wartościach odpowiednich *cyfr* zwraca 0. Jeśli przy którymś działaniu otrzymana wartość będzie różna od 0 funkcja zwróci 1 w przeciwnym wypadku zwracana wartość jest równa 0. W programie wartość ta jest rozumiana jako typ **bool**.

```
[...]
decq %rcx           # Zmniejszenie licznika mniejszej liczby (size - 1 = lastindex)
  cmpq $0, %rcx
    je after
  cmpq $0, %rdx
     je after

  andNumbers:
    decq %rdx
    movb (%rdi, %rdx, 1), %al
    andb (%r8, %rcx, 1), %al

    cmpb $0, %al
      jne end1
    cmpq $0, %rdx
      je end0
  loop andNumbers
    after:
    decq %rdx
    movb (%rdi, %rdx, 1), %al
    andb (%r8, %rcx, 1), %al
    cmpb $0, %al
      jne end1

end0:
    xorq %rax, %rax
    jmp exit

end1:
  movq $1, %rax

  [...]
```

Listing 4: Fragment funkcji **andFullChar**, liczby w rejestrach RDI i R8

#### 7.1.9 Rozwiązane problemy

Funkcje assemblerowe zostały napisane kodem 64-bit w konwencji AT&T początkowo w systemie Linux Ubuntu. Następnie przeniesiono je do systemu operacyjnego Windows w celu złączenia ich z resztą projektu. Niestety funkcje assemblera w systemie Windows posiadają inny sposób przesyłania argumentów niż na systemach Linuxowych. Z tego powodu w części funkcji została napisana specjalny kawałek kodu, który przekazywał argumenty, które znalazły się w rejestrach w konwencji Microsoftowej do rejestrów w których powinny się znaleźć w programie na systemie operacyjnym Ubuntu.

```
movq %rcx, %rdi        # Pierwszy argument
movq %rdx, %rsi        # Drugi argument
movq %r8, %rdx         # Trzeci argument
movq %r9, %rcx         # Czwarty argument
movq 48(%rbp), %r8     # Piąty argument przez stos
```

### 7.2 Funkcje drugiego programu

W programie głównym znajduje się 10 różnych funkcji oraz funkcja main. Są to:

1. int **get_prime**(unsigned char* *n*, int n_len, unsigned char* d) - funkcja wyszukująca dzielnik liczby n będący liczbą pierwszą zgodnie z algorytmem Rho Pollarda
2. int **randX**(unsigned char* x, const unsigned char* n, int n_len) - funkcja która wylosowuje wartość pierwszej liczby sekwencji $x$ do algorytmu Rho Pollarda
3. int **randC**(unsigned char* c, const unsigned char* n, int n_len) - która wylosowuje wartość stałej $c$ do funkcji pseudolosowej
4. int **copy**(const unsigned char* x, int x_size, unsigned char* y) - funkcja kopiuje liczbę x do liczby y
5. int **f**(unsigned char* x, int x_len, unsigned char* c, int c_len, unsigned char* n, int n_len) - funkcja generująca modularnie liczby pseudolosowe
6. int **GCD**(unsigned char* a, int size_a, unsigned char* b, int size_b, unsigned char* gdc) - funkcja wyliczająca NWD dwóch liczb, używa funkcji **EuclideanAlgorithm**
7. int **EuclideanAlgorithm**(unsigned char* a, int size_a, unsigned char* b, int size_b, unsigned char* gdc) - funkcja licząca NWD zgodnie z algorytmem Euklidesa
8. bool **priorityTest**(unsigned char* a, int size_a) - test pierwszości, zwraca czy liczba jest liczbą pierwszą, testując liczbę *Chińskim testem pierwszości*, wykorzystuje funkcję **PowMod**
9. int **MulMod**(unsigned char* a, int size_a, unsigned char* b, int size_b, unsigned char* c, int size_c, unsigned char* w) - funkcja wykonująca modularne mnożenie liczb
10. int **PowMod**(unsigned char* e, int size_e, unsigned char* u, int size_u, unsigned char* w) - funkcja wykonująca modularne potęgowanie liczb w celu wykrycia pierwszości liczby

### 7.3 Główna funkcja programu

Głowna funkcja **main()** pobiera od użytkownika liczbę do rozłożenia na liczby pierwsze liczbę dowolnej długość (w programie ograniczeniem jest długość 200 znaków, ale można tę liczbę zwiększyć lub zmniejszyć w kodzie źródłowym). Następnie dzieli ją przez 2 jeśli jest ona podzielna przez 2. W następnej kolejności sprawdza, czy jest ona liczbą pierwszą funkcją **priorityTest**, jeśli nie to wywołuje na niej funkcję **get_prime**, która powinna zwracać dzielnik pierwszy liczby. Następnie dzieli faktoryzowaną liczbę przez otrzymany dzielnik, wypisuje dzielnik i sprawdza czy liczba wynikająca z dzielenia jest liczbą pierwszą. Jeśli nie działania są powtarzane, w przeciwnym wypadku wypisuje otrzymaną z dzielenia liczbę, ponieważ jest ona ostatnim dzielnikiem pierwszym dzielonej liczby. Kończy się program.

```
int main() {
    /* initialize random seed */
    srand (time(nullptr));
    auto *number = new unsigned char[200];
    auto *prime = new unsigned char[150];
    auto *c = new unsigned char[200];

    printf("Wpisz liczbe: ");
    scanf("%s", number);

    int num_len = fullChar(number);

    unsigned char two = 2;
    while(number[num_len - 1] % 2 == 0)
    {
        num_len = divFullChars(&two, 1, number, num_len, c);
        for (int i = 0; i < num_len; ++i)
            number[i] = c[i];
        if(number[0] == 0)
            num_len = removezeros(number, num_len);
        printf("%d ", 2);
    }

    while(!(number[0] == 1 && num_len == 1)){
        if( !priorityTest(number, num_len) ){
            int size_p = get_prime(number, num_len, prime);
            for (int i = 0; i < num_len; ++i)
                c[i] = number[i];
            num_len = divFullChars(prime, size_p, c, num_len, number);
            num_len = removezeros(number, num_len);
            fullCharToString(prime, size_p);
            printf("%s ", prime);
        }
        else{
            fullCharToString(number, num_len);
            printf("%s ", number);
            break;
        }
    }

    delete [] prime;
    delete [] number;
    return 0;
}
```

### 7.4 Funkcja zwracająca dzielnik liczby - algorytm Rho Pollarda

```
/// Algorytm Rho Pollarda
int get_prime(unsigned char* n, int n_len, unsigned char* d)
{

    //    long long int x = (rand()%(n-2)) + 2;
    auto *x = new unsigned char [n_len];
    int x_len = randX(x, n, n_len);

    //    long long int y = x;
    auto *y = new unsigned char [n_len];
    int y_len = copy(x, x_len, y);

    //    long long int c = (rand()%(n - 1)) + 1;
    auto *c = new unsigned char [n_len];
    int c_len = randC(c, n, n_len);

    //    long long int d = 1;
    d[0] = 1;
    int d_len = 1;

    auto* pom = new unsigned char[n_len];
```

```
    auto n_copy = new unsigned char[n_len];

    bool keep_looking = true;
    while( keep_looking)
    {
        //        x = (f(x) + c) % n;
        x_len = f (x, x_len, c, c_len, n, n_len);

        //        y = (f((f(y) + c ) % n) + c) % n;
        y_len = f(y, y_len, c, c_len, n, n_len);
        y_len = f(y, y_len, c, c_len, n, n_len);

        //        d = GCD( llabs(x - y), number);
        int size = absFullChars(x, x_len, y, y_len, pom);
        copy(n, n_len, n_copy);
        d_len = GCD(pom, size, n_copy, n_len, d);
        keep_looking = d[0] == 1 && d_len == 1;
    }

    delete [] n_copy;
    delete [] c;
    delete [] x;
    delete [] y;

    if(n_len == d_len){
        int size = absFullChars(n, n_len, d, d_len, pom);
        if(size == 1 && pom[0] == 0)
            if(priorityTest(n, n_len)){
                delete [] pom;
                return 0;
            }
    }
    delete [] pom;

    if(!priorityTest(d, d_len))
    {
        auto *d_next = new unsigned char [d_len];
        d_len = get_prime(d, d_len, d_next);
        for (int i = 0; i < d_len; ++i)
            d[i] = d_next[i];
        delete [] d_next;
    }

    return d_len;
}
```

### 7.5 Funkcja losująca pierwszą liczbę sekwencji $X$

Funkcja **randX** losuje najpierw długość liczby, która nie może być większa od długości liczby $n$. Następnie losuje wartość każdej "cyfry" liczby (każdego elementu tablicy). Jeśli liczba $x$ ma podobną długość to jest dzielona modulo $n$. Liczba jest zapisywana w tablicy której wskaźnik jest przekazywany w parametrze **x** funkcji. Funkcja zwraca długość liczby $x$.

### 7.6 Funkcja generująca liczby pseudolosowe

Funkcja podnosi do drugiej potęgi liczbę $x$, dodaje stałą $c$ i dzieli powstałą liczbę przez $n$. Następnie resztę z dzielenia zapisuje do liczby $x$ i zwraca długość liczby $x$.

```
int f(unsigned char* x, int x_len, unsigned char* c, int c_len, unsigned char* n, int n_len)
{
    int size_sqrX = (x_len + x_len +  1) > c_len ? x_len + x_len +  1 : c_len + 1;
    auto * sqr_x = new unsigned char[size_sqrX];
    auto * d = new unsigned char[size_sqrX];
    size_sqrX = mulFullChars(x, x_len, x, x_len, sqr_x);
    size_sqrX = addFullChars(sqr_x, size_sqrX, c, c_len, sqr_x);
    divFullChars(n, n_len, sqr_x, size_sqrX, d);
    size_sqrX = removezeros(sqr_x, size_sqrX);
```

```
    for (int i = 0; i < size_sqrX; ++i)
        x[i] = sqr_x[i];
    delete [] sqr_x;
    delete [] d;
    return size_sqrX;
}
```

## 7.7 Funkcje sprawdzające pierwszość liczby algorytmem chińskiego testu pierwszości

Funkcja sprawdza czy po wykonaniu testu (przez funkcją **PowMod**) zwracaną wartością jest liczba 2. Jeśli nie oznacza to że liczba nie jest liczbą pierwszą. Jeśli jest to liczba pierwsza funkcja zwraca wartość **true**, w przeciwnym wypadku **false**.

```
//// Test pierwszości - Chiński test pierwszości
bool priorityTest(unsigned char* a, int size_a) {
    if(a[0] == 2 && size_a == 1)
        return true;
    auto c = new unsigned char[size_a+size_a];
    int size_c = PowMod(a, size_a, a, size_a, c);
    bool isPriority = false;
    if( c[0] == 2 && size_c == 1)
        isPriority = true;
    delete [] c;
    return isPriority;
}
```

Funkcja wykonuje modularne mnożenie na podanych liczbach. Wynik jest zapisywany w tablizy wskazywanej parametrem *w*.

```
//// Funkcja mnoży a i b mod n
int MulMod(unsigned char* a, int size_a, unsigned char* b, int size_b, unsigned char* c, int
    size_c, unsigned char* w)
{
    // Tablice pomocnicze
    auto pom = new unsigned char[size_c+1];
    auto mask = new unsigned char[size_b];
    unsigned char two = 2;

    int mask_len = 1;
    int size_w = 1;
    mask[0] = 1;
    w[0] = 0;
    for (int j = 0; j < size_b*8; ++j)
    {
        if(andFullChars(mask, mask_len, b, size_b))
        {
            int size = addFullChars(a, size_a, w, size_w, w);
            divFullChars(c, size_c, w, size, pom);
            size_w = removezeros(w, size);
        }
        pom[0] = 1;
        if(j == size_b*8 - 1)
            break;

        size_a = mulFullChars(&two, 1, a, size_a, pom);
        int l = pom[0] == 0 ? 1 : 0;
        if(l == 1) size_a--;
        for (int i = 0; i < size_a; ++i, ++l)
            a[i] = pom[l];
        divFullChars(c, size_c, a, size_a, pom);

        size_a = removezeros(a, size_a);
        mask_len = mulFullChars(&two, 1, mask, mask_len, pom);
        int k = pom[0] == 0 ? 1 : 0;
        if(k == 1) mask_len--;
        for (int i = 0; i < mask_len; ++i, ++k)
            mask[i] = pom[k];
```

```
    }
    delete [] mask;
    delete [] pom;
    return size_w;
}
```

Funkcja wykonuje modularne potęgowanie na podanych liczbach. W przypadku chińskiego testu pierwszości wskaźniki $e$ i $u$ powinny wskazywać ten sam adres. Wynik jest zapisywany w tablicy wskazywanej parametrem $w$.

```
//// Funkcja oblicza 2^e mod n
int PowMod(unsigned char* e, int size_e, unsigned char* u, int size_u, unsigned char* w)
{
    // Tablice pomocnicze
    auto m = new unsigned char[size_e];
    auto p = new unsigned char[size_e];
    auto p2 = new unsigned char[size_e+size_e];
    auto pom = new unsigned char[size_e+size_e];
    unsigned char two = 2;

    int size_m = 1, size_p = 1, size_w = 1;
    p[0] = 2; w[0] = m[0] = 1;

    for (int j = 0; j < size_e*8; ++j)
    {
        if(andFullChars(e, size_e, m, size_m))
        {
            size_w = MulMod(w, size_w, p, size_p, u, size_u, pom);
            for (int i = 0; i < size_w; ++i)
                w[i] = pom[i];
        }
        for (int i = 0; i < size_p; ++i)
            p2[i] = p[i];
        size_p = MulMod(p2, size_p, p, size_p, u, size_u, pom);
        for (int i = 0; i < size_p; ++i)
            p[i] = pom[i];

        if(j == size_e*8 - 1)
            break;

        size_m = mulFullChars(&two, 1, m, size_m, pom);
        int k = pom[0] == 0 ? 1 : 0;
        if(k == 1) size_m--;
        for (int i = 0; i < size_m; ++i, ++k)
            m[i] = pom[k];
    }
    delete [] m;
    delete [] p;
    delete [] p2;
    delete [] pom;
    return size_w;
}
```

## 8 Pomiary

Pomiary wykonano używając *QueryPerformanceCounter*. Pomiary wykonano 30 razy dla jednanej kategorii liczby, następnie wyniki uśredniono i obliczono odchylenie standardowe. Wyniki wykonano dla 4 kategorii w obu programach, oraz dla jednej w programie dla liczb wielkich. W pierwszym przypadku program losował odpowiednią liczbę liczb pierwszych z tablicy liczb pierwszych danego przedziału, następnie mnożył je i sprawdzał czy liczba złożona mieści się w zakresie, następnie zapisywał liczby. Po otrzymaniu 5 odpowiednich liczb wykonywał sześciokrotnie faktoryzację w pierwszym programie. Następnie wykonywano pomiary dla tych samych liczb w drugim programie.

Pomiary czasu wykonania algorytmu całkowitej faktoryzacji wykonano dla:

1. Liczb rozkładających się na trzy liczy pierwsze z przedziału [100987, 104729] (w przedziale 329 liczb pierwszych), przykładowa liczba 1097488704594187.

2. Liczb rozkładających się na 3 liczby pierwsze z przedziału [3, 1009].
3. Liczb rozkładających się na 7 liczb pierwszych z przedziału [3, 1009] mniejszych od $2^{63}$.
4. Liczb rozkładających się na 4 liczby pierwsze z przedziału [1009, 50023] mniejszych od $2^{63}$.

Następnie wykonano pomiary dla liczb złożonych, które miały 6 dzielników pierwszych z zakresu [999983, 1299709], wykonano również 30 pomiarów.

Czas liczono w $\mu s$ ( mikrosekunda; $1\mu s = 0,000001 s = 1 \cdot 10^{-6} s$ ).

| l.p. | Wyniki pierwszego programu | Wyniki drugiego programu |
|---|---|---|
| | [$\mu s$] | [$\mu s$] |
| 1 | 17860 | 19515 |
| 2 | 41243 | 77489 |
| 3 | 40679 | 70713 |
| 4 | 22114 | 12125 |
| 5 | 25081 | 97263 |
| 6 | 18713 | 54345 |
| 7 | 13979 | 50056 |
| 8 | 14086 | 11330 |
| 9 | 23348 | 32647 |
| 10 | 21447 | 34002 |
| 11 | 44428 | 81426 |
| 12 | 46789 | 5624 |
| 13 | 42263 | 47297 |
| 14 | 43273 | 27205 |
| 15 | 31460 | 13345 |
| 16 | 42168 | 79155 |
| 17 | 35839 | 20713 |
| 18 | 22360 | 25672 |
| 19 | 10765 | 24953 |
| 20 | 18040 | 12076 |
| 21 | 11886 | 44588 |
| 22 | 24175 | 47818 |
| 23 | 34381 | 49819 |
| 24 | 75164 | 50748 |
| 25 | 17211 | 11544 |
| 26 | 27797 | 1872 |
| 27 | 13695 | 9171 |
| 28 | 23372 | 13385 |
| 29 | 21490 | 140138 |
| 30 | 17186 | 12884 |
| Odchylenie st. | 13935,62 | 31596 |
| Średnia | 28076,4 | 39297,27 |

Tablica 1: Pomiary dla pierwszej kategorii [100987, 104729]

| l.p. | Wyniki pierwszego programu | Wyniki drugiego programu |
|---|---|---|
| | [$\mu s$] | [$\mu s$] |
| 1 | 12 | 1084 |
| 2 | 15 | 1485 |
| 3 | 14 | 785 |
| 4 | 17 | 1341 |
| 5 | 8 | 598 |
| 6 | 15 | 688 |
| 7 | 13 | 634 |
| 8 | 9 | 1506 |
| 9 | 22 | 723 |
| 10 | 10 | 467 |
| 11 | 5 | 382 |
| 12 | 7 | 500 |
| 13 | 5 | 353 |
| 14 | 13 | 414 |
| 15 | 11 | 365 |
| 16 | 9 | 404 |
| 17 | 18 | 340 |
| 18 | 7 | 461 |
| 19 | 14 | 718 |
| 20 | 18 | 279 |
| 21 | 14 | 300 |
| 22 | 10 | 406 |
| 23 | 7 | 441 |
| 24 | 4 | 479 |
| 25 | 3 | 240 |
| 26 | 5 | 145 |
| 27 | 6 | 263 |
| 28 | 4 | 336 |
| 29 | 7 | 149 |
| 30 | 28 | 115 |
| Odchylenie st. | 5,74 | 363,08 |
| Średnia | 11 | 546,7 |

Tablica 2: Pomiary dla drugiej kategorii rozkładu na 3 liczby pierwsze z przedziału [3, 1009]

| l.p. | Wyniki pierwszego programu | Wyniki drugiego programu |
|---|---|---|
| | [$\mu s$] | [$\mu s$] |
| 1 | 125 | 7084 |
| 2 | 246 | 11896 |
| 3 | 85 | 11731 |
| 4 | 86 | 9252 |
| 5 | 274 | 12099 |
| 6 | 876 | 8260 |
| 7 | 308 | 9317 |
| 8 | 214 | 11293 |
| 9 | 282 | 8986 |
| 10 | 209 | 9425 |
| 11 | 134 | 9625 |
| 12 | 553 | 11566 |
| 13 | 433 | 7604 |
| 14 | 268 | 8557 |
| 15 | 49 | 6627 |
| 16 | 139 | 8581 |
| 17 | 142 | 9147 |
| 18 | 227 | 11163 |
| 19 | 222 | 6514 |
| 20 | 96 | 7325 |
| 21 | 124 | 11654 |
| 22 | 102 | 8561 |
| 23 | 93 | 6893 |
| 24 | 736 | 9259 |
| 25 | 163 | 12108 |
| 26 | 87 | 8844 |
| 27 | 245 | 12147 |
| 28 | 243 | 6856 |
| 29 | 86 | 13683 |
| 30 | 73 | 10573 |
| Odchylenie st. | 9 | 190 |
| Średnia | 15,5 | 230,6667 |

Tablica 3: Pomiary dla trzeciej kategorii rozkładu na 7 liczb pierwszych z przedziału [3, 1009]

| l.p. | Wyniki pierwszego programu | Wyniki drugiego programu |
|---|---|---|
| | [$\mu s$] | [$\mu s$] |
| 1 | 1757 | 20830 |
| 2 | 2582 | 15887 |
| 3 | 520 | 10100 |
| 4 | 6503 | 14791 |
| 5 | 919 | 14123 |
| 6 | 2169 | 9982 |
| 7 | 1569 | 10652 |
| 8 | 4336 | 11254 |
| 9 | 5264 | 9120 |
| 10 | 5604 | 11902 |
| 11 | 1107 | 13827 |
| 12 | 1201 | 12095 |
| 13 | 8367 | 12091 |
| 14 | 2698 | 13560 |
| 15 | 4686 | 14803 |
| 16 | 1035 | 10692 |
| 17 | 3119 | 13546 |
| 18 | 2294 | 14253 |
| 19 | 4476 | 7174 |
| 20 | 1107 | 15356 |
| 21 | 8596 | 20344 |
| 22 | 15915 | 14360 |
| 23 | 4017 | 14009 |
| 24 | 2374 | 11091 |
| 25 | 2223 | 18281 |
| 26 | 5453 | 7891 |
| 27 | 1733 | 13688 |
| 28 | 5539 | 16008 |
| 29 | 1001 | 10854 |
| 30 | 3800 | 11287 |
| Odchylenie st. | 3162,48 | 3122 |
| Średnia | 13128,37 | 3732,133 |

Tablica 4: Pomiary dla czwartej kategorii rozkładu na 4 liczby pierwsze z przedziału [1009, 50023]

| l.p. | [$\mu s$] |
|---|---|
| 1 | 242815 |
| 2 | 340632 |
| 3 | 292008 |
| 4 | 255278 |
| 5 | 294139 |
| 6 | 395835 |
| 7 | 620928 |
| 8 | 419070 |
| 9 | 428960 |
| 10 | 121322 |
| 11 | 394968 |
| 12 | 337373 |
| 13 | 333330 |
| 14 | 460193 |
| 15 | 405117 |
| 16 | 718937 |
| 17 | 709962 |
| 18 | 549555 |
| 19 | 331016 |
| 20 | 284078 |
| 21 | 402656 |
| 22 | 325715 |
| 23 | 216535 |
| 24 | 343147 |
| 25 | 413736 |
| 26 | 567419 |
| 27 | 278272 |
| 28 | 369024 |
| 29 | 439174 |
| 30 | 210712 |
| Odchylenie st. | 137312,82 |
| Średnia | 383396,87 |

Tablica 5: Pomiary dla drugiego programu

# 9 Zakończenie

## 9.1 Wnioski

Oba programy działały poprawnie i zgodnie z założeniami. Zwracały poprawne czynniki przykładowych liczb.

Po analizie pomiarów czasu faktoryzacji wywnioskowane zostały dwie rzeczy. Po pierwsze każdy program rozkłada liczby nawet o takiej samej liczbie czynników w różnym czasie, co ukazuje duże odchylenie standardowe wyników. Po drugie pierwszy z programów wykonuje algorytm znacznie szybciej od drugiego, kiedy czynniki są niewielkie, kiedy czynniki stają się większe wtedy to drugi z programów zaczyna według wskazań średniej działać w lepszym czasie. Odchylenie standardowe w tym przypadku przekracza jednak 80% średniej, więc wynik ten może nie być wiarygodny.

Pierwsze spostrzeżenie jest interpretowane jako następstwo oparcia algorytm Rho Pollarda o probabilistykę. Wartości początkowe sekwencji oraz stała do funkcji pseudolosowej są generowane w trakcie wykonywania algorytmu. Z tego powodu czas w jakim wyszukiwane są kolejne dzielniki może się różnić, nawet dla takiej samej liczby.

Po drugie to, że pierwszy z programów wykonywany jest szybciej jest spowodowane wykonywaniem działań na typach elementarnych. Natomiast drugi z programów wykonuje cały szereg funkcji i choć są one bardzo szybkie zajmują więcej czasu, niż pojedyncze instrukcje. Dobrze widać to w drugiej serii pomiarów, gdzie pierwsze wyniki są mniejsze od 20 [$\mu s$], natomiast drugie czasem przekraczają nawet 1000 [$\mu s$]. Wynika to prawdopodobnie z tego, że pierwszy z programów od razu przechodzi do działania, natomiast drugi najpierw "przekodowuje" liczbę, następnie wykonuje funkcje, które przy niewielkiej wielkości liczby są często zbędne, a potem odkodowuje się celem prezentacji wyników.

Dzięki specjalnemu zapisywaniu drugi z programów rozkładał na czynniki pierwsze nawet bardzo długie liczby w czasie nie przekraczającym sekundy.

### 9.2 Podsumowanie

Przy pisaniu projektu nauczono się bardzo dużo o działaniu assemblera, wyszukiwania błędów w programach assemblerowych za pomocą debuggerów, pisaniu kodu assemblerowego, możliwościach łączenia go z językiem C++. Poznano wiele wartościowych informacji o liczbach pierwszych i problemach rozkładu liczb złożonych na czynniki pierwsze. Zapoznano się ze sposobami faktoryzacji liczb, a także nauczono się implementacji algorytmu faktoryzacji Rho Pollarda. Opanowano jeden ze sposobów zapisu wielkich liczb w komputerze.

## 10 Literatura


[1] *https://www.cs.colorado.edu/ srirams/courses/csci2824-spr14/pollardsRho.html*

[2] *https://homes.cerias.purdue.edu/ ssw/cun/*

[3] *https://www.geeksforgeeks.org/pollards-rho-algorithm-prime-factorization/*

[4] *http://zak.ict.pwr.wroc.pl/materials/architektura/laboratorium%20AK2/Linux-AK2-lab-2019%20May.pdf*

[5] Przekazywanie argumentów Microsoft Assembly: *https://docs.microsoft.com/pl-pl/cpp/build/x64-calling-convention?view=vs-2019*

[6] *https://en.wikipedia.org/wiki/Pollard%27s_ rho_ algorithm*

[7] Query Performance Counter: *http://staff.iiar.pwr.wroc.pl/antoni.sterna/pea/PEA_ time.pdf*