Organizacja i Architektura Komputerów

Implementacja biblioteki arytmetyki liczb zmiennoprzecinkowych dowolnej precyzji

Piotr Przybył 225952
Piotr Karoń 241626

Prowadzący:
dr inż. Tadeusz Tomczak

6 czerwca 2019

Spis treści

1. Cel projektu

Celem projektu jest stworzenie biblioteki arytmetyki zmiennoprzecinkowej dowolnej precyzji. Podstawowe typy danych reprezentujące liczby zmiennoprzecinkowe takie jak *float* czy *double* zapewniają stosunkowo niewielką precyzję, co dyskwalifikuje je w wielu zastosowaniach, np. bankowości, astronomii, fizyce. Ta wada wynika ze sposobu w jaki te typy danych są przechowywane w pamięci komputera. Ten projekt ma na celu stworzenie klasy obudowującej, która zapewni odpowiednią precyzję obliczeń.

2. Wykorzystane technologie

Do stworzenia biblioteki użyliśmy języka **C++** i jego biblioteki standardowej. Do testowania kodu wykorzystaliśmy bibliotekę **Google-Test**. Proces kompilacji jest zarządzany przez **CMake**.

3. Reprezentacja

Podstawą reprezentacji klasy liczb zmiennoprzecinkowych – $BigFloat$ – jest klasa wielkich liczb całkowitych – $BigInteger$ – która umożliwia przechowywanie liczb całkowitych większych niż pozwalają na to typy podstawowe języka C++.

$BigFloat$ składa się z:

- znaku, jako typ boolean, sprzężonego z mnożnikiem
- mnożnika, jako liczby całkowitej w postaci $BigInteger$
- wykładnika, jako liczby całkowitej w postaci $BigInteger$

4. Big Integer

$BigInteger$ jest reprezentowany jako wektor liczb całkowitych $uint32_t$ oraz zmienna typu *bool* jako znak. Wykorzystujemy cechę jaką jest ciągłość tablicy w pamięci. Przyjmując, że każda dodatkowa komórka tablicy to rozszerzenie liczby o kolejne 32 pozycje, możemy traktować całą tablicę jako jedną liczbę. Ograniczeniem staje się tylko rozmiar tej tablicy, ale biorąc pod uwagę szybkość zwiększania się zakresu względem rozszerzania tablicy, przekroczenie zasobów pamięciowych jest mało prawdopodobne (osiągnięcie wartości 10^{100} wymaga jedynie 333 komórek).

Tabela 1: Przykład reprezentacji. Jedna komórka szerokości 32 bitów.

Wartość	Rozmiar tablicy	Reprezentacja w pamięci
0	1	$[00\ldots000]$
1	1	$[00\ldots001]$
2	1	$[00\ldots010]$
$2^{32}-1$	1	$[11\ldots111]$
2^{32}	2	$[00\ldots001][00\ldots000]$
$2^{32}+1$	2	$[00\ldots001][00\ldots001]$
$2^{64}-1$	2	$[11\ldots111][11\ldots111]$
2^{64}	3	$[00\ldots001][00\ldots000][00\ldots000]$

Taka reprezentacja ułatwia operacje. Przykładowo, dodanie dwóch liczb $BigInteger$ sprowadza się do zsumowania odpowiednich komórek tablic. Maksymalnie otrzymamy wartość $2^{64}-2$ – wynik możemy bezpiecznie przechować jako typ $long$. Następnie obliczamy jaka część wyniku zostanie w komórce, w której przeprowadzamy operację oraz przeniesienie, które dodajemy do komórki następnej. Więcej o operacjach matematycznych w rozdziale 4.1.

Definicje pojęć:

- **storage** - wektor przechowujący liczbę
- **sign** - zmienna przechowująca znak
- **int** - typ $std :: int32_t$
- **uint** - typ $std :: uint32_t$
- **long** - typ $std :: int64_t$
- **ulong** - typ $std :: uint64_t$
- słowo - komórka $storage$

4.1. Funkcje

4.1.1. Konstruktor

Dostępne są konstruktory:

- BigInteger()
- BigInteger(std::int32_t)

- BigInteger(std::uint32_t)
- BigInteger(std::int64_t)
- BigInteger(std::uint64_t)
- BigInteger(const std::vector std::uint32_t &, bool = false)
- BigInteger(std::string)

Podstawowy konstruktor jako parametr przyjmuje łańcuch znaków *string* zawierający **liczbę o podstawie heksadecymalnej** z ewentualnym znakiem −. To założenie pozwala w prosty sposób przekonwertować ciąg znakowy na ciąg bitów.

4.1.2. Dodawanie

Dodawanie jest realizowane z pomocą przeciążonego operatora +. Przed dodawaniem krótsza liczba zostaje rozszerzona zerami lewostronnie, aby długości tablic były zgodne. Następnie, algorytm dodaje kolejno odpowiadające sobie komórki tablic obu liczb od najmłodszej. Ze względu na to, że jedna komórka ma długość 32 bitów, to w wyniku dodawania otrzymamy co najwyżej 64 bity – możemy więc go przechować w zmiennej typu *ulong*. Gdy wynik przekroczy maksymalną wartość, którą może przechować *unt* to ustawiana jest flaga przeniesienia, która zostanie uwzględniona w kolejnej iteracji dodawania komórek, a do obecnej komórki zostają wpisane młodsze 32 bity wyniku. Operacja kończy się w momencie osiągnięcia końca wektora.

4.1.3. Odejmowanie

Odejmowanie jest realizowane z pomocą przeciążonego operatora −. Operacja polega na iteratywnym odejmowaniu odpowiednich komórek *storage*. Odejmowanie dwóch liczb typu *uint* zwraca wynik mieszczący się w zakresie typu *long*. Jeśli jest on ujemny flaga pożyczki zostaje podniesiona i uwzględniona w kolejnym odejmowaniu. Do komórki wyniku wpisywane są młodsze 32 bity wyniku (korzystając z rzutowania *long* na *uint*). Operacja kończy się w momencie osiągnięcia końca wektora.

4.1.4. Mnożenie

Mnożenie realizowane jest z pomocą przeciążonego operatora ∗. Mnożenie dwóch liczb typu *uint* może zwrócić wynik maksymalnie 64 bitowy. Funkcja iteruje od najmłodszych pozycji operandów i mnoży ich odpowiednie komórki. Młodsze 32 bity wyniku zostają dodane do $storage[i]$, starsze 32 bity do $storage[i+1]$. Operacja kończy się w momencie osiągnięcia końca wektora.

4.1.5. Dzielenie

Dzielenie realizowane jest z pomocą przeciążonego operatora /, funkcji $divide(BigInteger)$, która jako parametry przyjmuje dzielnik (dzielną jest obiekt, na którym zostaje ona wywołana) oraz prywatnej funkcji pomocniczej *divmnu*. Jest to implementacja algorytmu dzielenia całkowitego bez znaku zaproponowanego przez D.E. Knutha w książce *The Art of Programming. Volume 2: Seminumerical Algorithms, Chapter 4.3: Multiple-Precision Arithmetic* [1]. W tym projekcie wykorzystano zmodyfikowaną implementację przedstawioną przez Henry. S. Warrena na stronie z materiałami dodatkowymi do książki $Hacker's Delight$ [2] [3]. Funkcja *divide* zwraca resztę z dzielenia typu $BigInteger$

5. BigFloat

Podstawowym założeniem liczby typu $BigFloat$ jest ustalenie, że wykładnik ma wartość maksymalną pod warunkiem zachowania całkowitości mnożnika. Innymi słowy, w binarnej reprezentacji mnożnik jako najmłodszy bit zawsze posiada 1. Po operacjach dodawania, odejmowania i mnożenia jest wywoływania procedura mnożenia, która odpowiednio skaluje mnożnik i dostosowuje wykładnik.

5.1. Funkcje

5.1.1. Konstruktor

- BigFloat(BigInteger significand, BigInteger exponent, bool sign = false)
- BigFloat(double value)
- BigFloat(std::string str)

Podstawowy konstruktor przyjmuje liczbę szesnastkową, zmiennoprzecinkową w formacie z kropką oddzielającą część całkowitą od ułamkowej.

5.1.2. Dodawanie

Dodawanie jest zrealizowane z wykorzystaniem przeciążonego operatora +. Przed operacją lewy operand jest skalowany tak, aby wykładnik był równy drugiemu składnikowi. Następnie mnożniki są sumowane. Na koniec wynik jest normalizowany – skalowany w prawo, tak aby usunąć zera.

5.1.3. Odejmowanie

Odejmowanie jest zrealizowane z wykorzystaniem przeciążonego operatora $-$. Przed operacją lewy operand jest skalowany tak, aby jego wykładnik był równy wykładnikowi odjemnika. Następnie mnożniki są odejmowane. Na koniec wynik jest normalizowany – skalowany w prawo, tak aby usunąć zera.

5.1.4. Mnożenie

Mnożenie wykorzystuje przeciążony operator $*$. Podczas operacji mnożniki są przez siebie mnożone a wykładniki dodawane. Na koniec wynik jest normalizowany.

5.1.5. Dzielenie

W dzieleniu mnożniki operandów są dzielone, a wykładniki odejmowane. Realizowane jest z pomocą przeciążonego operatora / oraz funkcji $divide(BigInteger, BigInteger, Round)$, która jako parametry przyjmuje kolejno dzielnik, precyzję oraz tryb zaokrąglania. Dzielną jest obiekt, na którym ta metoda jest wywołana, a zwracany jest nowy obiek typu $BigInteger$ zawierający iloraz.

Precyzja określa liczbę cyfr znaczących w zapisie szesnastkowym. Zadana precyzja jest uzyskana przez odpowiednie przeskalowanie dzielnej przed dzieleniem.

Następnie przeprowadzane jest zaokrąglanie.

W przypadku trybu zaokrąglania przez obcięcie wynik zostaje zwrócone bez obróbki. W przypadku trybu zaokrąglania w dół, gdy wynik jest ujemny, dodawany jest jeden bit do ostatniego *sowastorage*. Kiedy dodatni – wynik zostaje niezmieniony. W przypadku trybu zaokrąglania w górę, gdy wynik jest dodatni, dodawany jest jeden bit do ostatniego *sowastorage*. Kiedy dodatni – wynik zostaje niezmieniony. W przypadku trybu zaokrąglania symetrycznie do parzystej, w wyniku dzielenia, otrzymujemy mnożnik z zadaną liczbą bitów znaczących oraz resztę z dzielenia, na podstawie której jest przeprowadzanie zaokrąglanie. Resztę skalujemy tak, aby otrzymać kolejne 32 cyfry heksadecymalne (128 bitów) przy kolejnym dzieleniu. Na podstawie ostatniego słowa w *storage* wyliczane są bity R i S, z użyciem których wykonywane jest zaokrąglenie.

6. Realizacja projektu

Cały zrealizowany przez nas projekt wraz ze wskazówkami jak go zbudować znajduje się na stronie

https://github.com/Piteropeter/OiAK

6.1. Struktura

Projekt zdecydowaliśmy się realizować w języku C++. Jest on kompilowany do postaci biblioteki statycznej którą użytkownik końcowy dołącza do swojej aplikacji. Nasza biblioteka wykorzystuje narzędzie do zarządzania strukturą projektu takie jak *CMake*. Projekt jest cross-kompilowalny pomiędzy Windowsem i Linuxem. Testowaliśmy go wykorzystując Windowsa 10 (1809) oraz Ubuntu 18.04. Wszystkie pliki związane z naszą biblioteką znajdują się w katalogu *src/*.

6.2. Testy jednostkowe

```
TEST(big_integer_constructor_tests, int8min_constructor) {
    auto x = BigInteger(INT8_MIN);
    EXPECT_EQ(x.size(), 1);
    EXPECT_EQ(x[0], 128);
    EXPECT_TRUE(x.get_sign());
}

TEST(big_integer_constructor_tests, int8max_constructor) {
    auto x = BigInteger(INT8_MAX);
    EXPECT_EQ(x.size(), 1);
    EXPECT_EQ(x[0], 127);
    EXPECT_FALSE(x.get_sign());
}
```

Rysunek 1: Przykładowe testy jednostkowe

Nie mielibyśmy pewności czy nasza biblioteka działa prawidłowo gdyby nie testy jednostkowe. Staraliśmy się pokryć każdy pisany przez nas kawałek kodu testami. Poskutkowało to tym, że w ostatecznej wersji mamy 123 testy jednostkowe, które dbają

o to czy któraś z wprowadzanych przez nas zmian nie psuje tego co napisaliśmy do tej pory. Testy jednostkowe zostały zrealizowane przez nas przy pomocy frameworku testowego *googletest* autorstwa Google. Wszystkie pliki związane z naszymi testami jednostkowymi są zlokalizowane w katalogu *ut/*.

6.3. Testy wydajnościowe

```
TEST(big_integer_constructor_tests, int8min_constructor) {
    auto x = BigInteger(INT8_MIN);
    EXPECT_EQ(x.size(), 1);
    EXPECT_EQ(x[0], 128);
    EXPECT_TRUE(x.get_sign());
}

TEST(big_integer_constructor_tests, int8max_constructor) {
    auto x = BigInteger(INT8_MAX);
    EXPECT_EQ(x.size(), 1);
    EXPECT_EQ(x[0], 127);
    EXPECT_FALSE(x.get_sign());
}
```

Rysunek 2: Przykładowy test wydajnościowy

Na samym końcu dodaliśmy do naszego projektu testy wydajnościowe. Dzięki temu mogliśmy zebrać dane na temat rzeczywistej wydajności naszego kodu. Testy wydajnościowe zostały zrealizowane przy pomocy frameworku *benchmark* autorstwa Google. Stworzyliśmy 60 testów. Wszystkie pliki związane z naszymi testami wydajnościowymi znajdują się w katalogu *benchmark/*. Na końcu dokumentacji znajdują się pomiary dokonane przy pomocy tych testów.

6.3.1. Pomiary

Pomiary zostały dokonane na Ubuntu 18.04 przy wykorzystaniu procesora i9-9900k pracującego z częstotliwością 5.00 GHz. Pomiary w innych warunkach mogą odstawać od tych zawartych na końcu sprawozdania.

7. Podsumowanie i wnioski

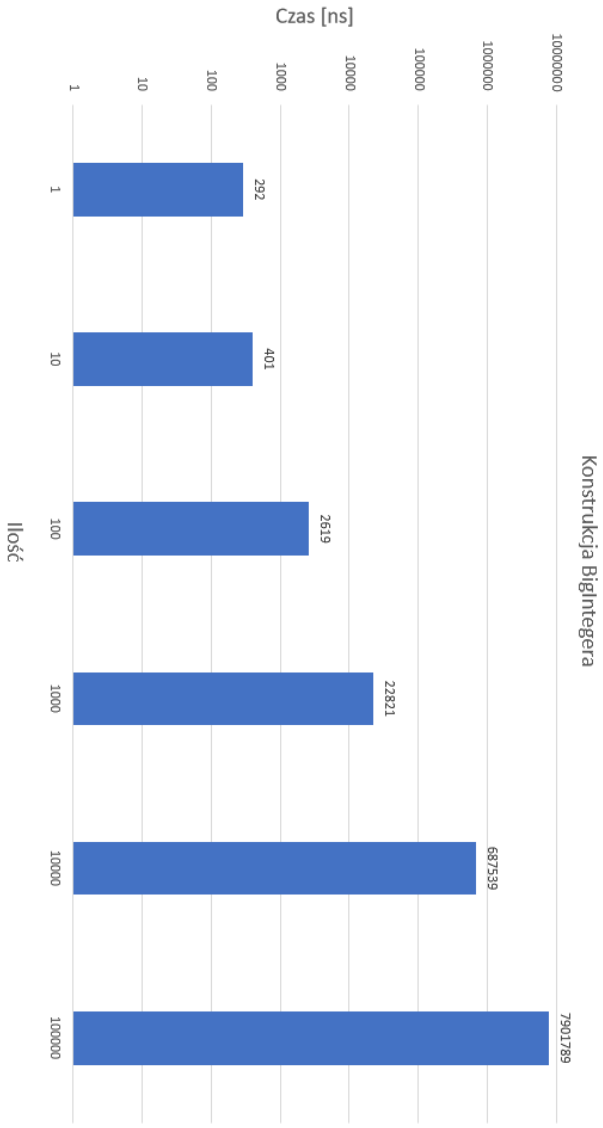
Udało nam się skończyć projekt i jesteśmy z niego bardzo dumni. Niestety z braku czasu nie udało nam się dodać wszystkich rzeczy, które dodać chcieliśmy. Brakuje sprawdzania czy wprowadzone Stringi są poprawne. Brakuje prawidłowej obsługi wyjątków. Brakuje dodatkowych metod (przesunięcia bitowe itp), które w innych implementacjach są obecne. Jedyna rzecz z której w naszej realizacji zadowoleni nie jesteśmy to wydajność dzielenia BigFloata. Zdajemy sobie sprawę z tego że jest to najbardziej wymagający obliczeniowo element jednakże zdecydowanie odstaje on od innych zaimplementowanych przez nas działań arytmetycznych, niemniej działa prawidłowo. Projekt wymagał on od nas przypomnienia sobie jak niskopoziomowo przebiega mnożenie liczb zmiennoprzecinkowych oraz wymyślenia własnego sposobu liczenia tychże. Wymagał od nas wymyślenia i zaimplementowania skalującego się w nieskończoność integera, który nie tylko nie byłby problemem w naszej implementacji BigFloata ale i stanowił solidną podstawę dla dalszej pracy. I w końcu dał nam możliwość popracowania z różnymi okołoprogramistycznymi narzędziami takimi jak CMake, Make, Git, GoogleTest i Google Benchmark. Wiele się podczas niego nauczyliśmy, dlatego projekt uważamy za bardzo ciekawy i rozwijający.

Literatura

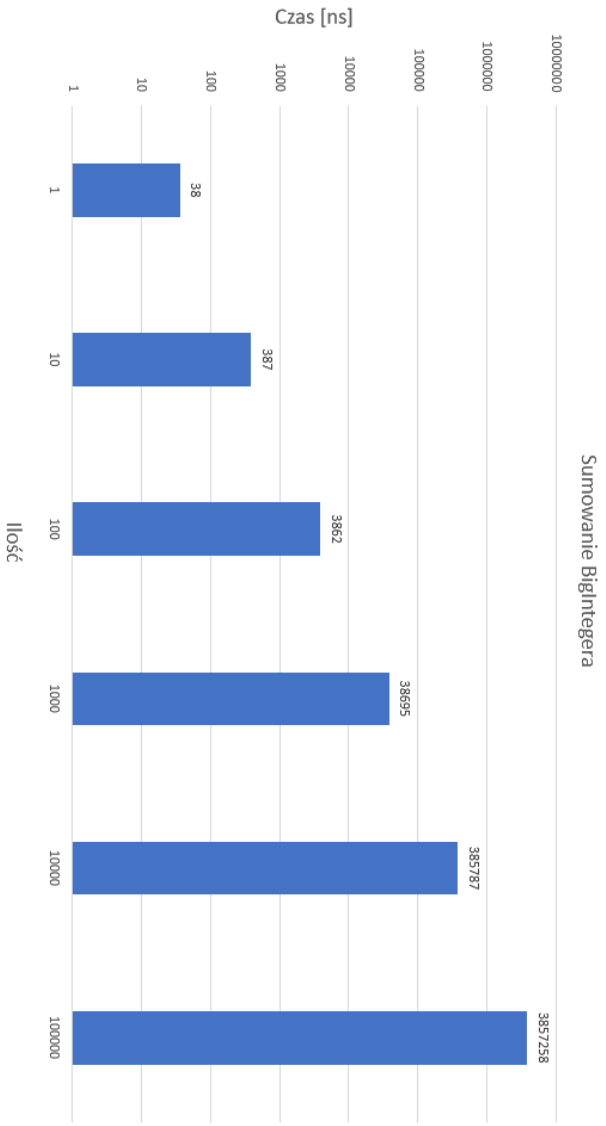
[1] D.E. Knuth `The Art of Computer Programming, Volume 2: Seminumerical`, Addison-Wesley, 1997

[2] Henry S. Warren, `https://hackersdelight.com`, 2012

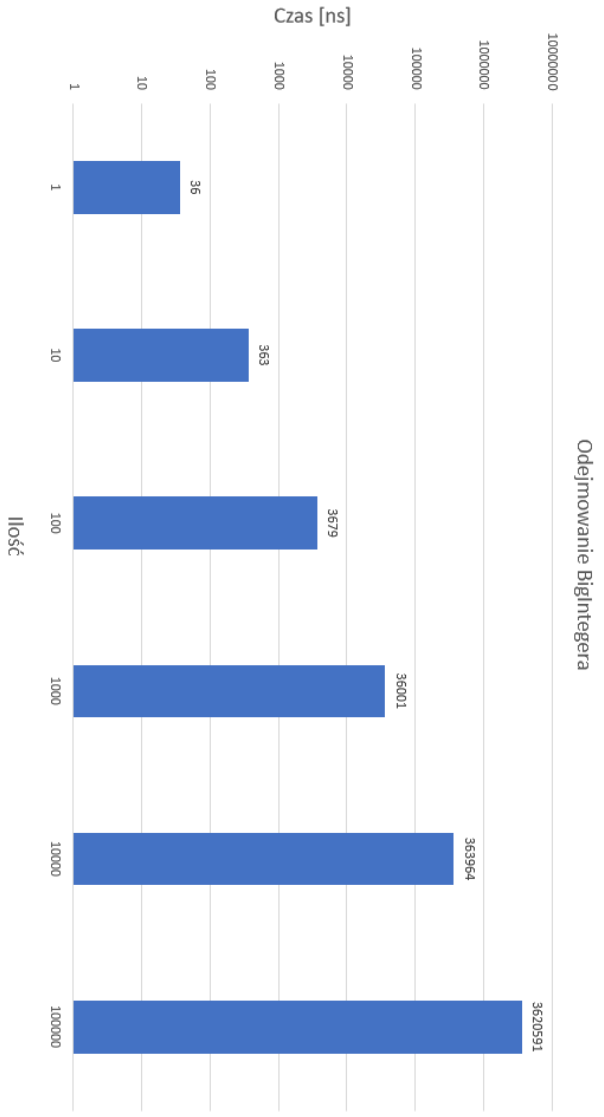
[3] Henry S. Warren, `Hacker's Delight`, Addison-Wesley, 2012



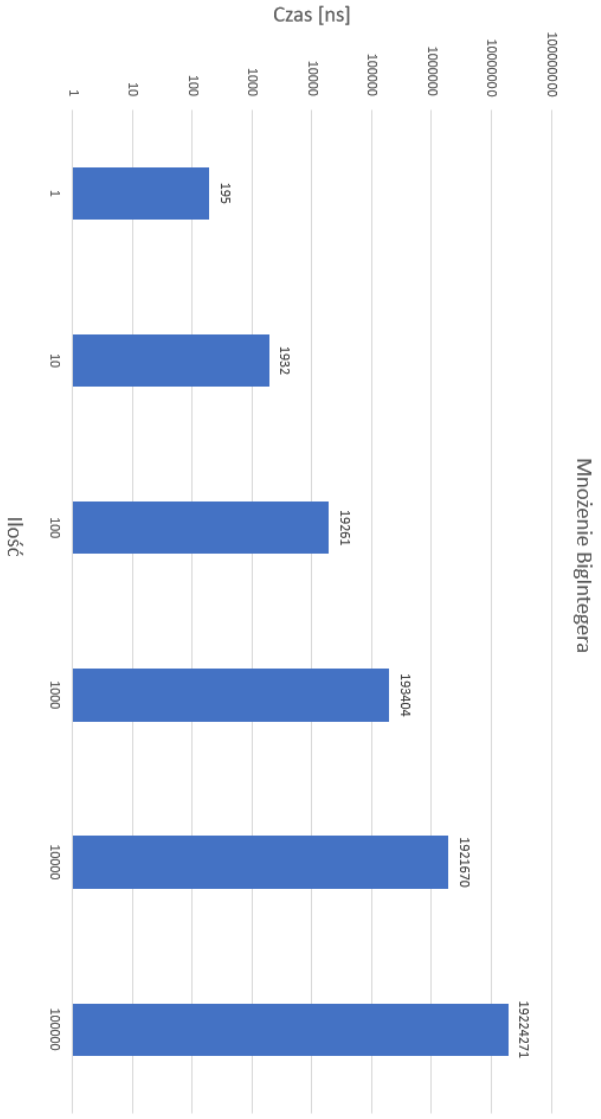
Rysunek 3: Konstrukcja BigIntegera



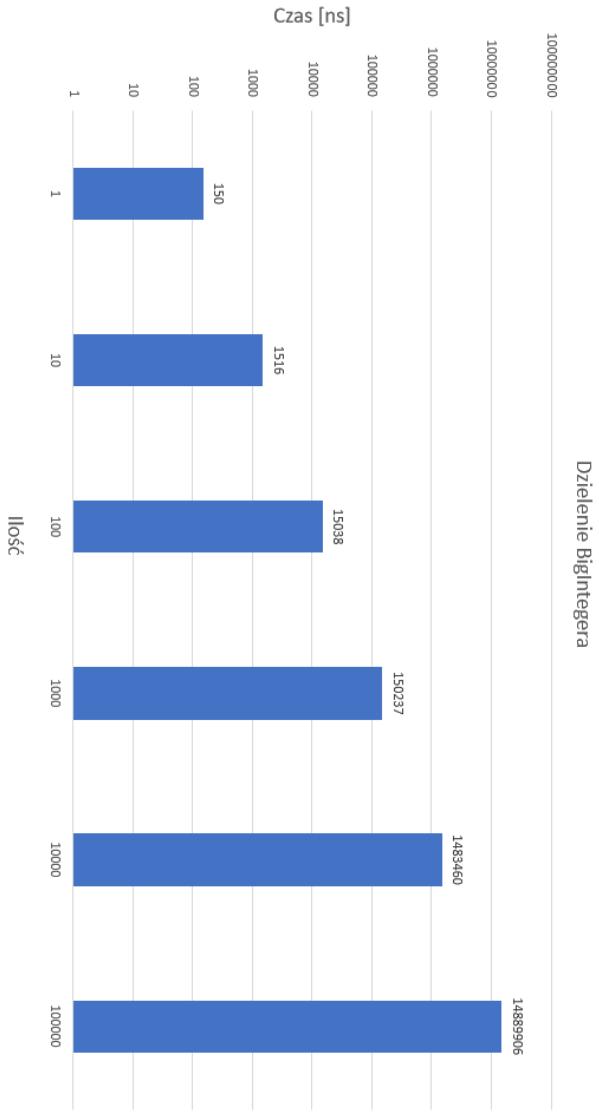
Rysunek 4: Dodawanie BigIntegera



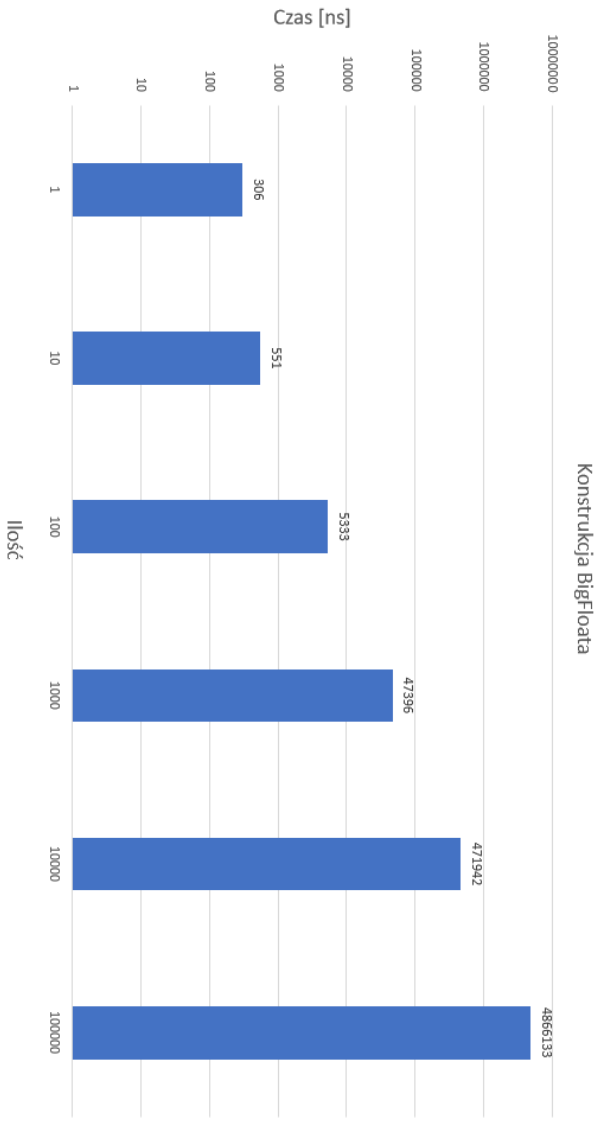
Rysunek 5: Odejmowanie BigIntegera



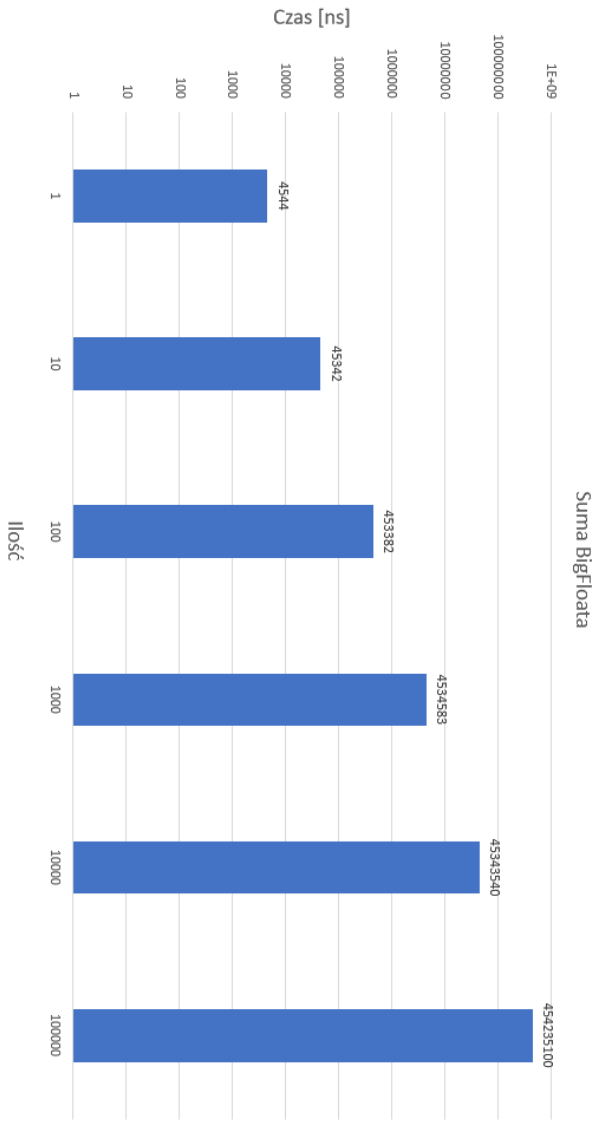
Rysunek 6: Mnożenie BigIntegera



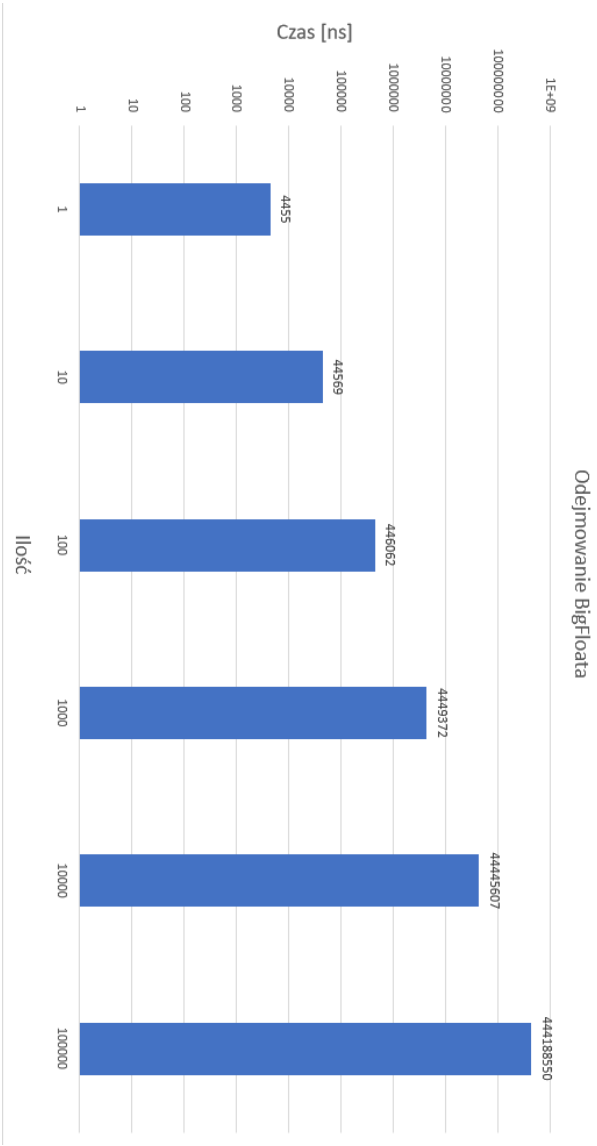
Rysunek 7: Dzielenie BigIntegera



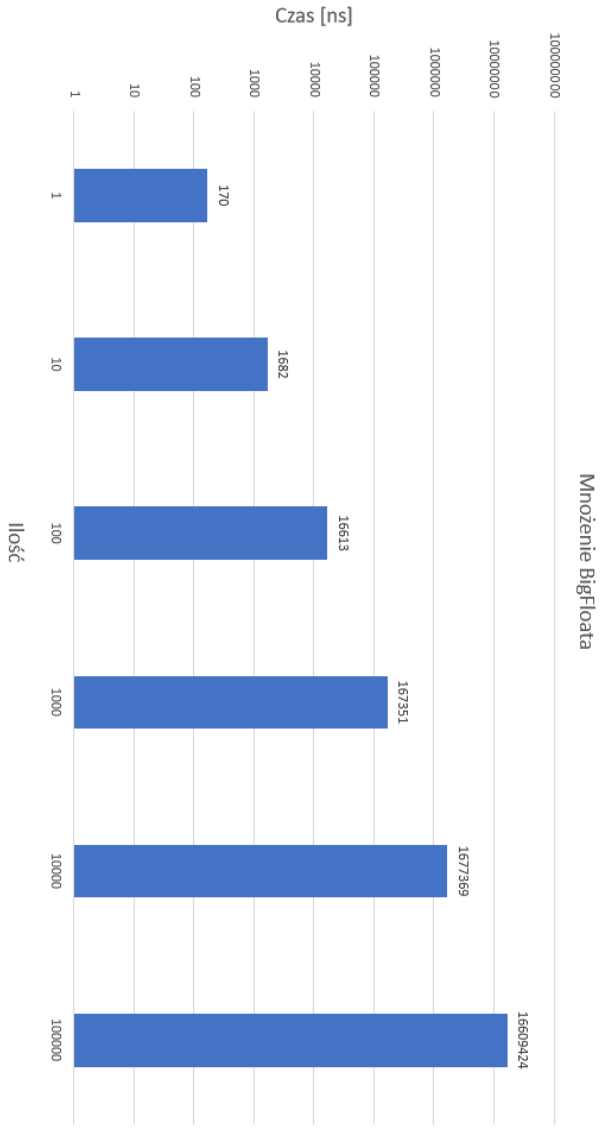
Rysunek 8: Konstrukcja BigFloata



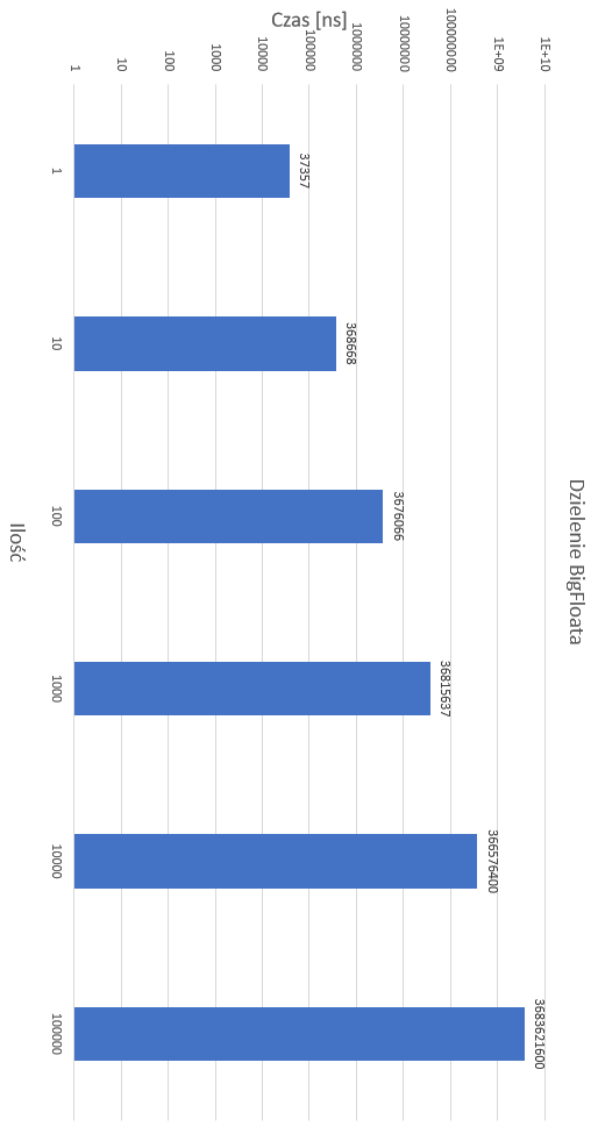
Rysunek 9: Dodawanie BigFloata



Rysunek 10: Odejmowanie BigFloata



Rysunek 11: Mnożenie BigFloata



Rysunek 12: Dzielenie BigFloata