Projekt 2

do samodzielnego wykonania

Dane jest poniższa implementacja algorytmu badania czy zadana liczba jest pierwsza:

```
bool IsPrime(BigInteger Num)
{
if (Num < 2) return false;
else if (Num < 4) return true;
else if (Num % 2 == 0) return false;
else for (BigInteger u = 3; u < Num / 2; u += 2)
if (Num % u == 0) return false;
return true;
}
```

Celem projektu jest zaproponowanie bardziej efektywnego algorytmu przy zachowaniu niezmienionego interfejsu podprogramu. Przeprowadzić analizę za pomocą instrumentacji i pomiarów czasu. Przyjąć, że operacją dominującą jest dzielenie modulo (%).

W sprawozdaniu przedstawić dla obu algorytmów:

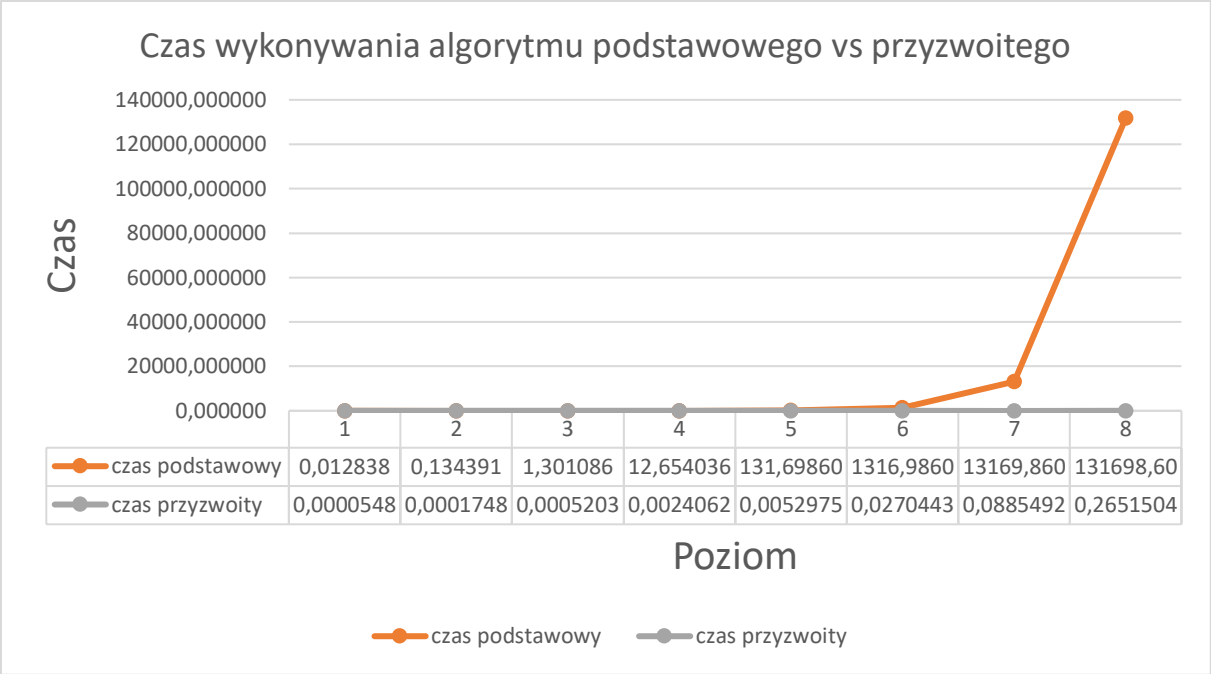
- kod źródłowy przed instrumentacją
- kod źródłowy po instrumentacji
- zebrane wyniki w postaci tekstu i wykresów
- wnioski z analizy zebranych danych (ocena złożoności)

Badanie przeprowadzić dla następującego zbioru punktów pomiarowych (liczb pierwszych):

{ 100913, 1009139, 10091401, 100914061, 1009140611, 10091406133, 100914061337, 1009140613399 }

Czas wykonywania algorytmu:

Poziom	Liczba	czas podstawowy	czas przyzwoity	Iloraz czasu Podst vs Przyzwoitego
1	100913	0,012838	0,00005481	234,228
2	1009139	0,134391	0,00017476	769,005
3	10091401	1,301086	0,00052033	2500,501
4	100914061	12,654036	0,00240621	5258,908
5	1009140611	131,698609	0,00529749	24860,568
6	10091406133	1316,986091	0,02704431	48697,345
7	100914061337	13169,860907	0,08854916	148729,371
8	1009140613399	131698,609073	0,26515042	496693,949



Analizując czas wykonywania algorytmu podstawowego oraz algorytmu przyzwoitego, można z łatwością zauważyć, że niewielka zmiana w pętli for zdecydowanie poprawiła wydajność. Czas wykonywania algorytmu podstawowego dla liczby na poziomie 8 (1009140613399) jest nawet 496694 razy dłuższy niż czas algorytmu przyzwoitego, a badając całą tablicę wyników możemy zauważyć, że im większa liczba jest poddawana sprawdzeniu tym przewaga algorytmu przyzwoitego rośnie.

Badanie czasu było wykonywane na kodzie przed instrumentacją:

Kod podstawowy:

```
static bool IsPrime(BigInteger Num)
        {
            if (Num < 2) return false;
            else if (Num < 4) return true;
            else if (Num % 2 == 0) return false;
            else
                for (BigInteger u = 3; u < Num / 2; u += 2)
                    if (Num % u == 0) return false;
            return true;
        }
```

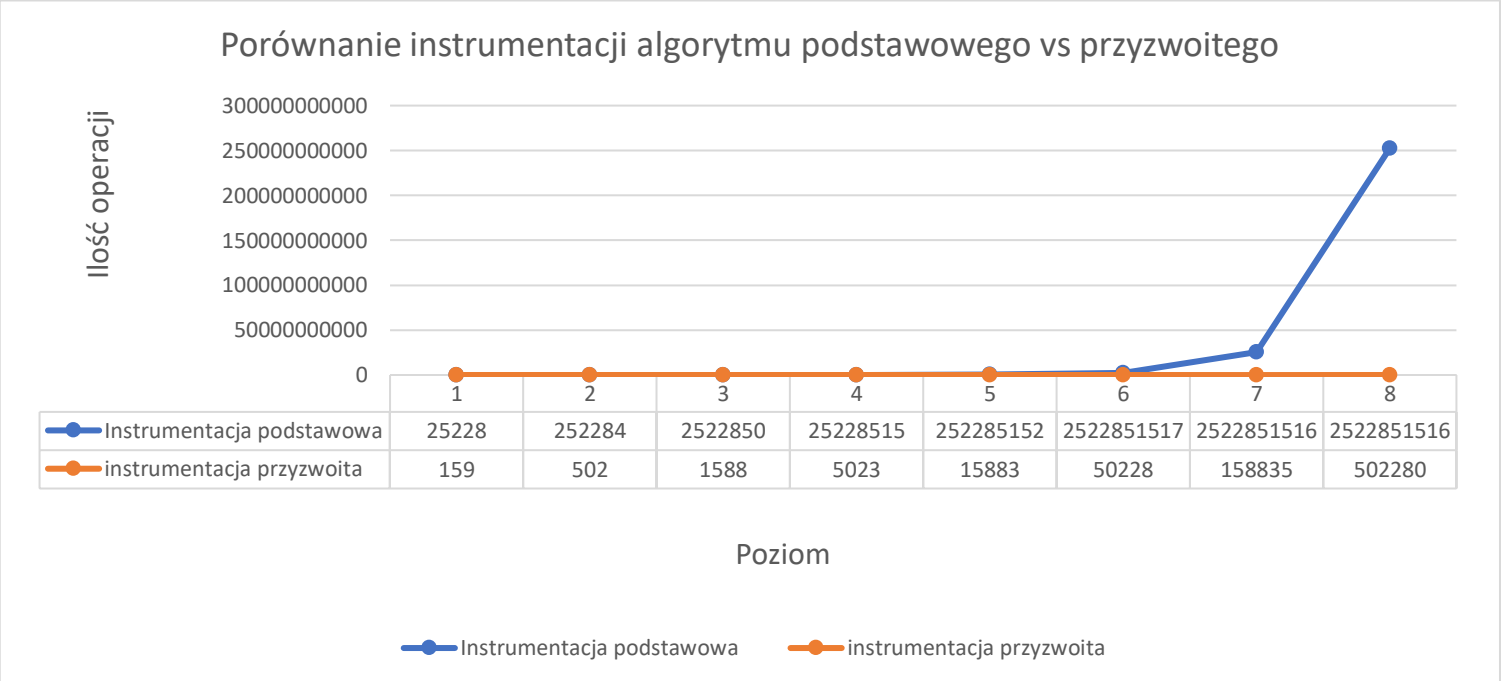
Kod poprawiony – przyzwoity:

```
static bool IsPrimeFaster(BigInteger Num)
        {
            if (Num < 2) return false;
            else if (Num < 4) return true;
            else if (Num % 2 == 0) return false;
            else
                for (BigInteger u = 3; u * u <= Num; u += 2)

                    if (Num % u == 0) return false;

            return true;
        }
```

Zmiana w kodzie została zaznaczona czerwoną, pogrubioną czcionką.

Skrócenie czasu jest wynikiem zmniejszenia liczby operacji potrzebnych do wykonania algorytmu co możemy zauważyć w tabelce oraz na wykresie przedstawiającym instrumentację dla obu algorytmów.

Poziom	Liczba	Instrumentacja podstawowa	instrumentacja przyzwoita	Iloraz Instr. Podst vs Przyzwoita
1	100913	25228	159	159
2	1009139	252284	502	503
3	10091401	2522850	1588	1589
4	100914061	25228515	5023	5023
5	1009140611	252285152	15883	15884
6	10091406133	2522851517	50228	50228
7	100914061337	25228515160	158835	158835
8	1009140613399	252285151601	502280	502280



Podczas wykonywania instrumentacji również łatwo zauważyć o wiele mniejszą liczbę wykonywanych operacji w algorytmie przyzwoitym w porównaniu do algorytmu podstawowego. Liczba operacji w alg. podstawowym potrafi być nawet 502280 razy większa względem przyzwoitego podczas badania liczby z poziomu 8 (1009140613399). Tak samo jak w przypadku badania czasu, przewaga algorytmu przyzwoitego nad podstawowym rośnie w miarę badania coraz większych liczb. Dodatkowo należy zwrócić uwagę na to, że ilość operacji w alg. przyzwoitym jest pierwiastkiem liczby operacji wykonywanych w alg. podstawowym.

Kod źródłowy po instrumentacji dla algorytmu:

Podstawowego:

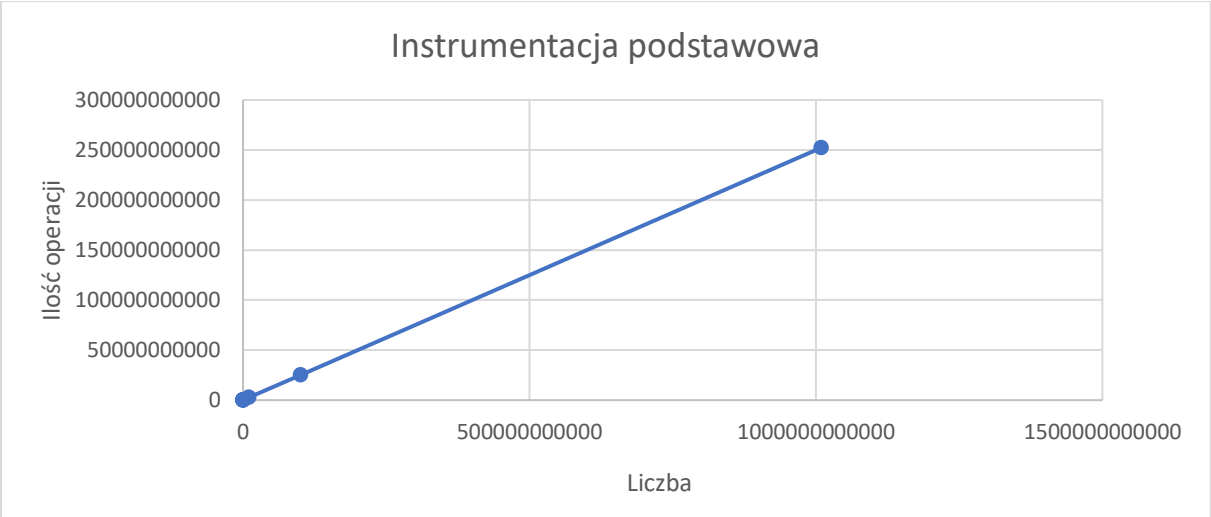
```
static bool IsPrimeInstr(BigInteger Num)
        {
            Licz = 1;
            if (Num < 2) return false;
            else if (Num < 4) return true;
            else if (Num % 2 == 0) return false;
            else
                for (BigInteger u = 3; u < Num / 2; u += 2)
                {
                    Licz++;

                    if (Num % u == 0)
                    {


                        return false;
                    }
                }
            return true;
        }
        static void IsPrimeInstrOut(BigInteger Num)
        {

            bool result = IsPrimeInstr(Num);
            Console.WriteLine("\t" + Licz);

        }
```

Przyzwoitego:

```
static bool IsPrimeFasterInstr(BigInteger Num)
        {
            Licz = 1;
            if (Num < 2) return false;
            else if (Num < 4) return true;
            else if (Num % 2 == 0) return false;
            else
                for (BigInteger u = 3; u * u <= Num; u += 2)
                {
                    Licz++;
                    if (Num % u == 0) return false;
                }
            return true;
        }
        static void IsPrimeFasterInstrOut(BigInteger Num)
        {

            bool result = IsPrimeFasterInstr(Num);
            Console.WriteLine("\t" + Licz);

        }
```

Wnioski – ocena złożoności

Złożoność obliczeniowa dla algorytmu podstawowego jest – O(n), ponieważ algorytm sprawdza nieparzyste wartości z zakresu [3, Num /2).



Po optymalizacji kodu i ograniczeniu zakresu poszukiwań przez zastosowanie u * u <= Num, ograniczamy liczbę dzielników. Co za tym idzie, w przypadku algorytmu przyzwoitego mamy do czynienia ze złożonością pierwiastkową – O(√n).