Politechnika Wrocławska,
18.04.2024

Organizacja i architektura komputerów

Analiza i porównanie algorytmów Montgomery’ego w kontekście wydajności czasowej i pamięciowej

Marek Tutka 276314,
Małgorzata Skowron 272963

Projekt grupa 6, czwartek nieparzysty 11:15

Spis treści

1. Cel projektu

Celem projektu jest przeprowadzenie kompleksowej analizy algorytmów Montgomery'ego, z uwzględnieniem algorytmu podstawowego MonPro oraz algorytmów Separated Operand Scanning (SOS) oraz Coarsely Integrated Operand Scanning (CIOS). Analiza ta ma na celu porównanie wydajności tych algorytmów z punktu widzenia wydajności czasowej oraz pamięciowej. Poprzez badanie czasu wykonania oraz ilości zajmowanej pamięci dla różnych wielkości danych wejściowych, projekt ma na celu dostarczenie głębokiego zrozumienia różnic między tymi algorytmami oraz identyfikację ich mocnych i słabych stron.

2. Wstęp teoretyczny

2.1. Arytmetyka modulo

Arytmetyka modulo jest oparta na resztach z dzielenia. Zamiast zwykłych liczb całkowitych, używamy klas reszt modulo *m,* gdzie *m* jest dodatnią liczbą całkowitą zwaną modułem. Dla dowolnych liczba całkowitych a i m (m 0), istnieje taka liczba r ($0 \leq r < m$), że zachodzi:

$$a = q * m + r$$

Gdzie q jest pewną liczbą, r jest resztą z dzielenia a przez m. Oznaczamy to $a\ (mod\ m)$

Dwie liczby całkowite a i b są kongruentne modulo m (zapisujemy $a \equiv b\ (mod\ m)$), jesli mają taką samą resztę z dzielenia, czyli $a\ (mod\ m) = b\ (mod\ m)$. Przez kongruencję rozumie się relację między dwoma liczbami, które mają tę samą resztę z dzielenia przez określony moduł.

Arytmetyka modulo ma wiele zastosowań, spotykamy się z nią codziennie, np. w zegarku: godziny liczymy od 0 do 23 (mod 24), minuty od 0 do 59 (mod 60).

2.2. Redukcja dzielenia modulo

Redukcja modularna jest czasochłonną operacją arytmetyczną. Klasyczna metoda realizacji polega na dzieleniu z resztą argumentów będących wynikiem potęgowania dużych liczb, czyli najpierw wykonujemy mnożenie, następnie redukujemy ten wynik dzieląc go przez moduł. Przy takim podejściu, wyniki potęgowania bardzo szybko rosną, a cała operacja staje się powolna. Znacznie efektywniejszą metodę przedstawił P. L. Montgomery.

2.3. Arytmetyka Montgomery'ego

Arytmetyka Montgomery'ego to udoskonalona wersja arytmetyki modulo, która pozwala na efektywne wykonywanie obliczeń modulo m, nawet gdy m jest dużą liczbą pierwszą. Jest ona szczególnie przydatna w kryptografii (np. RSA, wymiana kluczy Diffie-Hellman), gdzie wymaga się potęgowania modulo dużych liczb.

Załóżmy, że moduł n jest k-bitową liczbą taką, że $2^{k-1} \leq n < 2^k$, oraz $r = 2^k$
(algorytm działa dla każdego r, które jest relatywnie pierwsze do n, $NWD(n,r) = 1$, natomiast najefektywniej jest wybrać potęgi liczby 2)
Warunek $NWD(n,r) = 1$ jest spełniony, gdy n jest nieparzyste.

Jeśli spełniony jest warunek $a < n$, oraz resztę modulo określimy jako

$$a' = a * r(mod\ n),$$

to zbiór:

$$\{a * r\ (mod\ n) \mid 0 \leq a \leq n-1\}$$

jest ***zredukowanym układem reszt modulo n*** (tzn. dla dowolnego $a' \in \{0,\dots,n-1\}$ istnieje dokładnie jeden element a, że $a' \equiv a * r(mod\ n)$)

Mając dwie reszty modulo a', oraz $\boldsymbol{b'}$, *Montgomery product* określany jest jako

$$MonPro(a',b') = u' = a' * b' * r^{-1}\ (mod\ n)$$

Gdzie liczby a, b, n są k-bitowymi liczbami binarnymi, $a, b < n$, natomiast r^{-1} jest odwrotnością modularną $r\ (mod\ n)$, więc spełniona jest własność:

$$r^{-1} * r = 1\ (mod\ n)$$

Do modularnej redukcji Montgomery'ego potrzebujemy dodatkowo wartości n', spełniającej następujący warunek:

$$r * r^{-1} - n * n' = 1$$

2.4. Pseudokod funkcji MonPro(a',b')

funkcja MonPro(a',b')
Krok1. t := a' * b'
Krok2. m := t * n'(mod r)
Krok3. u' := (t + m * n) / r
Krok4: **jeśli** u' ≥ n **wykonaj zwróć** u' – n
zwróć u'

Ponieważ r jest potęgą liczby 2, operacje modulo oraz operacje dzielenia są szybkie, w porównaniu z normalnym obliczaniem $a * b\ (mod\ n)$, które wymaga dzielenia przez n (będące dowolną liczbą).

2.5. Pseudokod funkcji ModExp(M,e,n)

funkcja ModExp(M,e,n) {n jest nieparzyste}
Krok1. Oblicz n'
Krok2. M' := M * r (mod n)
Krok3. u' := 1 * r (mod n)
Krok4. **dla** i = k - 1 **w dół do** 0 **zrób**
Krok5. u' := MonPro(u', u')
Krok6. **jeśli** e_i = 1 **wykonaj** u' := MonPro(M', x')
Krok7. u := MonPro(x',1)
Krok8. **zwróć** u

Powyższy algorytm prezentuje binarny sposób obliczania potęg.
Wykładnik przedstawiany jest w postaci binarnej, a proces potęgowania zastąpiony jest przez podnoszenie do kwadratu oraz mnożenie przez początkową wartość (jeśli na *i*-tym miejscu binarnej reprezentacji wykładnika występuje 1)

Po zakończeniu pętli zmienna u' przechowuje wartość kongruentną do wartości $u = M^e (mod\ n)$, aby uzyskać wartość pierwotną, wykorzystujemy następującą własność algorytmu Montgomery'ego:

$$MonPro(u', 1) = u' * 1 * r^{-1}\ (mod\ n) = u * r * r^{-1} (mod\ n) = u\ (mod\ n)$$

Operacje na dużych liczbach często wykonuje się dzieląc te liczby na mniejsze kawałki zwane "słowami" (oznaczane w), gdzie:

$$k = s * w, \quad W = 2^w, \quad r = 2^{s*w}$$

W naszych algorytmach, wielkość "słowa", będzie wynosiła 1 ($w = 1$)

3. Algorytmy

Poniżej zostaną przedstawione 2 algorytmy, Separated Operand Scanning (SOS), oraz Coarsely Integrated Operand Scanning (CIOS). Wykonamy na nich analizę statyczną, określającą ilość operacji dodawania, mnożenia, zapisu oraz odczytu.

Należy pamiętać, że owe algorytmy dotyczą jedynie implementacji funkcji MonPro(a',b').

3.1. Separated Operand Scanning (SOS)

Krok 1, czyli obliczenie a*b odbywa się w następujący sposób:

```
for i=0 to s-1
        C := 0
        for j=0 to s-1
                (C,S) := t[i+j] + a[j]*b[i] + C
                t[i+j] := S
```

```
        t[i+s] := C
```

Gdzie t ma rozmiar 2*s oraz jest zainicjowane wartoscią “0”, a wynik stanowi liczba przechowywana “słowami” $t[0], t[1], \ldots, t[2s-1]$

Następnie w kroku 2 oraz 3 obliczamy $u' := (t + m * n)/r$. Aby to zrobić do t dodaje my $m * n$. Dzielenie przez $r = 2^{s*w}$ odbywa się poprzez przesunięcie w prawo o s “słów”.

```
for i=0 to s-1
        C := 0
        m := t[i]*n'[0] mod W
        for j=0 to s-1
                (C,S) := t[i+j] + m*n[j] + C
                t[i+j] := S
        ADD (t[i+s],C)
for j=0 to s
        u[j] := t[j+s]
```

Po wykonaniu przesunięcia, okazuje się, że wynik w zmiennej u posiada rozmiar $s + 1$

Funkcja “ADD” wykonuje propagacja przeniesienia od miejsca $t[i + s]$ do momentu, gdy przeniesienie przestanie być generowane. Ponieważ przeniesienie jest propagowane aż do ostatniego słowa, zwiększa rozmiar zmiennej t do rozmiaru $2 * s + i$.

W kroku 4 wykonujemy redukcje u, jeśli jest to potrzebne:

```
B := 0
for i=0 to s-1
        (B,D) := u[i] - n[i] - B
        t[i] := D
(B,D) := u[s] - B
        t[s] := D
if B=0 then return t[0], t[1], ... , t[s-1]
        else return u[0], u[1], ... , u[s-1]
```

3.2. Coarsely Integrated Operand Scanning (CIOS)

Te metoda w dużym stopniu bazuje na poprzedniej, główną różnicą jest połączenie mnożenia z redukcją. Zamiast obliczenia najpierw obliczenia a*b, a następnie redukcji, jest to wykonywane na zmianę. Taka operacja jest możliwa, ponieważ wartość m w i-tej operacji redukcji zależy od wartości $t[i]$, która jest obliczona w i-tej operacji multiplikacji (poprzedzającą operację redukcji). Warto zaznaczyć, że ze względu na przesunięcie w prawo w każdej i-tej operacji, odnosimy się do $t[j]$ oraz $t[0]$ zamiast do $t[i + j]$ lub $t[i]$

```
for i=0 to s-1
```

```
        C := 0
        for j=0 to s-1
                (C,S) := t[j] + a[j]*b[i] + C
                t[j] := S
        (C,S) := t[s] + C
        t[s] := S
        t[s+1] := C
        C := 0
        m := t[0]*n'[0] (mod W)
        for j=0 to s-1
                (C,S) := t[j] + m*n[j] + C
                t[j] := S
        (C,S) := t[s] + C
        t[s] := S
        t[s+1] := t[s+1] + C
        for j=0 to s
                t[j] := t[j+1]
B := 0
for i=0 to s-1
        (B,D) := u[i] - n[i] - B
        t[i] := D
(B,D) := u[s] - B
        t[s] := D
if B=0 then return t[0], t[1], ... , t[s-1]
        else return u[0], u[1], ... , u[s-1]
```

4. Założenia projektowe

Zaprezentowane algorytmy zaimplementowaliśmy w języku Python 3.12.0 i porównujemy je w zakresie wydajności czasowej i pamięciowej. Ostateczne wartości to wyniki dla funkcji MonExp, które otrzymywane są dzięki algorytmowi podstawowemu, SOS lub CIOS. Testy zostały przeprowadzone dla 1000 powtórzeń dla różnej ilości bitów (1, 2, 4, 8, 16, 32, 64, 128). Liczby o zadanej ilości bitów były losowane z uwzględnieniem warunków dla obliczenia wartości MonExp ($n > 0$, $n \epsilon niep$, $e \geq 0$, $0 < a < n-1$). Losowanie odbywało się za pomocą funkcji getrandbits(bits), która generuje losowe bity dla ilości bitów (bits).

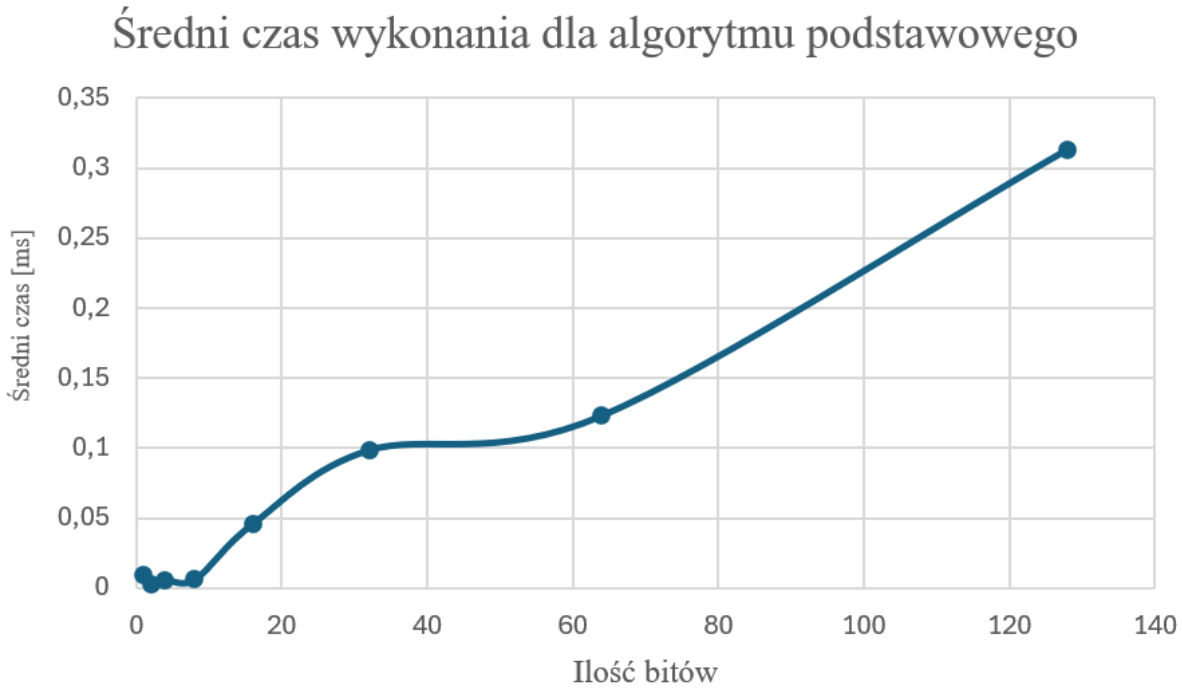
Każdy algorytm został przetestowany dla tych samych zestawów liczb losowych. Czas wykonania danych algorytmów mierzony jest przy użyciu biblioteki time. Program po zakończonym testowaniu wyświetla średni czas wykonania każdego z algorytmów.

5. Analiza czasowa

5.1. Tabela wyników

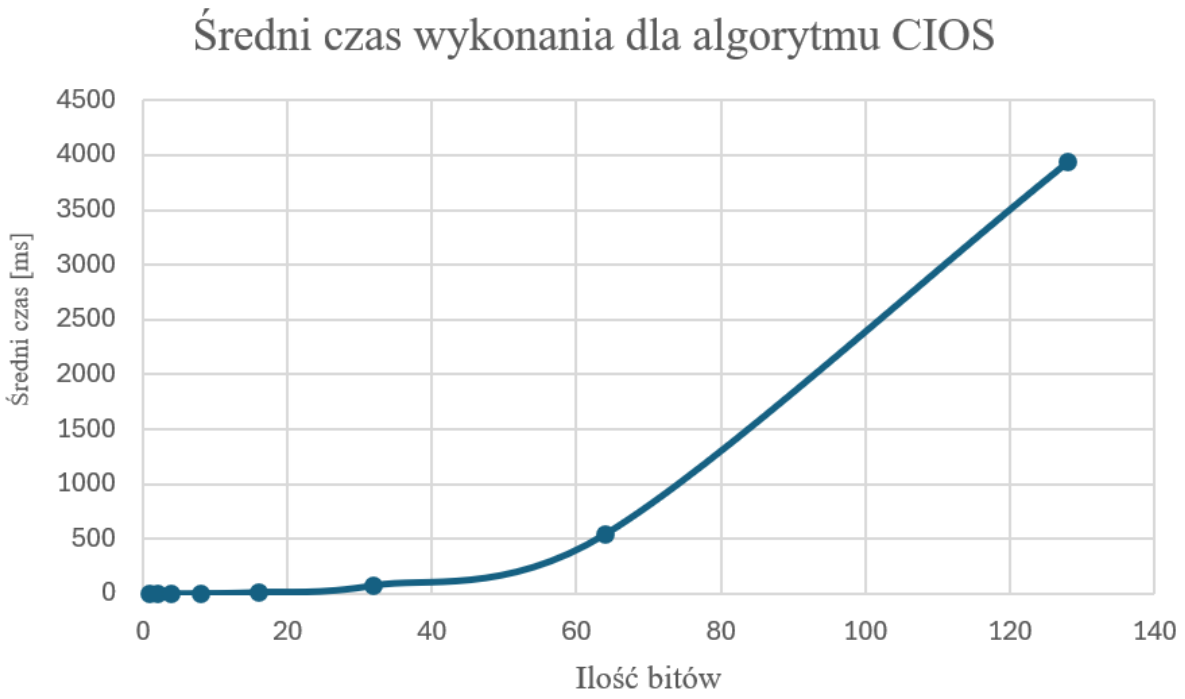
Tab. 1 - wyniki dla różnych algorytmów potęgowania Montgomery'ego			
	średni czas wykonania algorytmu [ms]		
ilość bitów	algorytm podstawowy	algorytm SOS	algorytm CIOS
1	0,0095	0,057	0,0616
2	0,0029	0,0954	0,0649
4	0,0058	0,2787	0,2869
8	0,0063	1,7304	1,6913
16	0,0462	10,857	12,377
32	0,0987	65,648	74,864
64	0,1233	471,8	541,19
128	0,3132	3405,5	3932,8

5.2. Wykresy

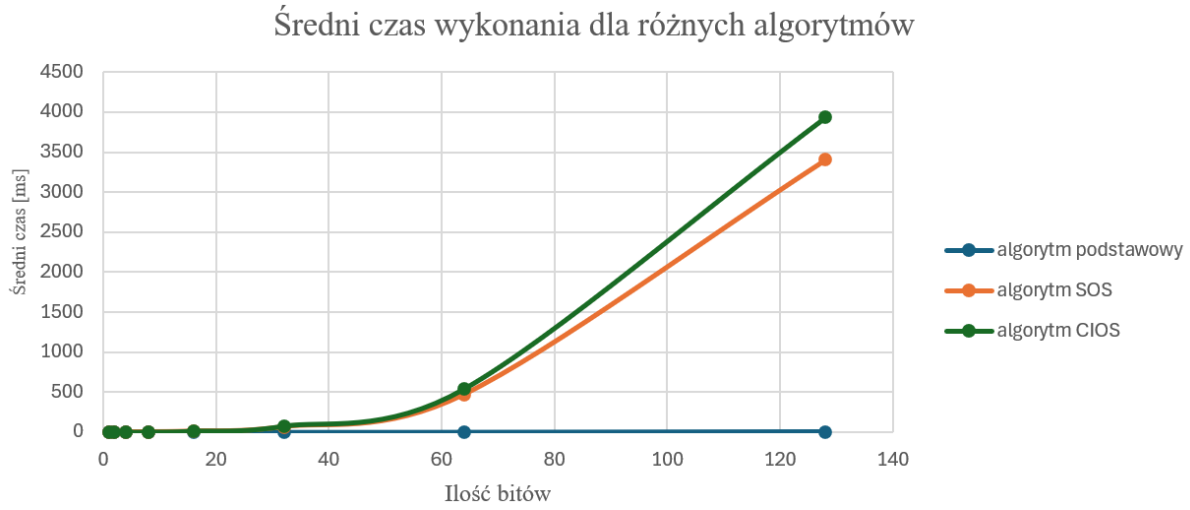


Rys. 1 – Średni czas wykonania dla algorytmu podstawowego na podstawie ilości bitów

Rys. 2 – Średni czas wykonania dla algorytmu SOS na podstawie ilości bitów



Rys. 3 – Średni czas wykonania dla algorytmu CIOS na podstawie ilości bitów



Rys. 4 – Średni czas wykonania dla różnych algorytmów na podstawie ilości bitów

6. Analiza wymaganych operacji oraz przestrzeni

6.1. Separated Operand Scanning (SOS)

	mult	add	read	write	iterations
for i=0 to s-1	-	-		-	-
C := 0	0	0	0	0	s
for j=0 to s-1	-	-	-	-	-
(C,S) := t[i+j] + a[j]*b[i] + C	1	2	3	0	s^2
t[i+j] := S	0	0	0	1	s^2
t[i+s] := C	0	0	0	1	s
for i=0 to s-1	-	-	-	-	-
C := 0	0	0	0	0	s
m := t[i]*n'[0] mod W	1	0	2	1	s
for j=0 to s-1	-	-	-	-	-
(C,S) := t[i+j] + m*n[j] + C	1	2	3	0	s^2
t[i+j] := S	0	0	0	1	s^2
ADD (t[i+s],C)	0	2	2	2	s
for j=0 to s	-	-	-	-	-
u[j] := t[j+s]	0	0	1	1	s
Final substraction	0	2(s+1)	2(s+1)	s+1	1
	$2s^2 + s$	$4s^2 + 4s + 2$	$6s^2 + 7s + 2$	$2s^2 + 6s + 2$	

Rys. 5- Operacje dla algorytmu SOS

Algorytm SOS potrzebuje $2s + 2$ tymczasowej przestrzeni do przechowywania wyniku.

6.2. Coarsely Integrated Operand Scanning (CIOS)

	mult	add	read	write	iterations
for i=0 to s-1	-	-	-	-	-
C := 0	0	0	0	0	s
for j=0 to s-1	-	-	-	-	s^2
(C,S) := t[j] + a[j]*b[i] + C	1	2	3	0	s^2
t[j] := S	0	0	0	1	s
(C,S) := t[s] + C	0	1	1	0	s
t[s] := S	0	0	0	1	s
t[s+1] := C	0	0	0	1	s
C := 0	1	0	2	0	s
m := t[0]*n'[0] (mod W)	-	-	-	-	-
for j=0 to s-1	1	2	3	0	s^2
(C,S) := t[j] + m*n[j] + C	0	0	0	1	s^2
t[j] := S	0	1	1	0	s
(C,S) := t[s] + C	0	0	0	1	s
t[s] := S	0	1	1	1	s
t[s+1] := t[s+1] + C	-	-	-	-	-
for j=0 to s	0	0	1	1	s^2
t[j] := t[j+1]					
Final substraction	0	2(s+1)	2(s+1)	s+1	1
	$2s^2 + s$	$4s^2 + 5s + 2$	$6s^2 + 7s + 2$	$2s^2 + 6s + 2$	

Rys. 6 -Operacje dla algorytmu CIOS

Algorytm CIOS potrzebuje $s + 2$ tymczasowej przestrzeni do przechowywania wyniku.

7. Analiza wyników

Programy zostały zaimplementowane w języku Python. Algorytmy MonPro znajdują się w dodatku A, a pozostałe kody są dostępne w załączniku 1.

Algorytm podstawowy to bezpośrednia implementacja pseudokodu funkcji MonPro. Działania arytmetyczne w tej implementacji są wykonywane przez język Python dla liczb większych niż słowa, przy założeniu, że słowa mają rozmiar 64 bitów, co odpowiada architekturze 64-bitowej.

Implementacje algorytmów SOS oraz CIOS zostały wykonane dla słów o wielkości 1 bit, co prowadzi do znacznej różnicy w wynikach w porównaniu z algorytmem podstawowym. To potwierdza istotność wielkości słowa w kontekście wydajności algorytmów Montgomery'ego.

W przypadku algorytmu CIOS nie zintegrowaliśmy przesunięcia w prawo z pętlą redukcyjną, co skutkuje dodatkowymi operacjami dodawania s, prowadzącymi do nieznacznego wydłużenia czasu wykonania w porównaniu z algorytmem SOS.

Powyższe wyniki wskazują na istotność wyboru odpowiedniej implementacji algorytmów Montgomery'ego w zależności od konkretnych wymagań. Dla niektórych zastosowań większe znaczenie może mieć szybkość wykonania, podczas gdy dla innych kluczowa może być minimalizacja zużycia pamięci. Dlatego analiza i porównanie różnych implementacji algorytmów Montgomery'ego są kluczowe dla wyboru optymalnej strategii w zależności od wymagań projektowych oraz charakterystyki środowiska wykonawczego.

8. Bibliografia

- C. Kaya Koc, T. Acar, and B. S. Kaliski, ”Analyzing and comparing Montgomery multiplication algorithms,”IEEE Micro, vol. 16, no. 3, ss. 26-33, June 1996, doi: 10.1109/40.502403

- Computerphile “Square & Multiply Algorithm” https://www.youtube.com/watch?v=cbGB__V8MNk

- Henk C. A. van Tilborg, Sushil Jajodia, “Encyclopedia of Cryptography and Security”, https://link.springer.com/referencework/10.1007/978-1-4419-5906-5

9. Dodatek A

9.1. Funkcja podstawowa MonPro

```
def monpronaive(ap, bp, n, r, np):
    t = ap * bp # obliczenie t
    m = t * np % r # obliczenie m
    u = (t + m * n) // r # obliczenie u
    if u >= n:
        return u - n
    return u
```

9.2. Przykład przejść podst. MonPro $7^{10}(mod\ 13)$

```
t = 9 = 3 * 3
m = 3 = 9 * 11 % 16
u = 3 = 9 + 3 * 13 // 16
t: 9 m: 3 u: 3 n: 13 ap: 3 bp: 3
t = 24 = 3 * 8
m = 8 = 24 * 11 % 16
u = 8 = 24 + 8 * 13 // 16
t: 24 m: 8 u: 8 n: 13 ap: 3 bp: 8
t = 64 = 8 * 8
m = 0 = 64 * 11 % 16
u = 4 = 64 + 0 * 13 // 16
t: 64 m: 0 u: 4 n: 13 ap: 8 bp: 8
t = 16 = 4 * 4
m = 0 = 16 * 11 % 16
u = 1 = 16 + 0 * 13 // 16
t: 16 m: 0 u: 1 n: 13 ap: 4 bp: 4
t = 8 = 1 * 8
m = 8 = 8 * 11 % 16
u = 7 = 8 + 8 * 13 // 16
t: 8 m: 8 u: 7 n: 13 ap: 1 bp: 8
t = 49 = 7 * 7
m = 11 = 49 * 11 % 16
u = 12 = 49 + 11 * 13 // 16
t: 49 m: 11 u: 12 n: 13 ap: 7 bp: 7
t = 12 = 12 * 1
m = 4 = 12 * 11 % 16
u = 4 = 12 + 4 * 13 // 16
t: 12 m: 4 u: 4 n: 13 ap: 12 bp: 1
Wynik: 7^10 (mod 13) = 4
```

9.3. Algorytm SOS

```
def CIOS(ap_bin, bp_bin, n_bin, np,s, w=1):
    # inicjalizacja t oraz u
    t = [0] * (2*s+1)
    u = t.copy()
```

```
    # algorytm, w ktorym krok mnozenia i redukcji jest zintegrowany
    for i in range(s):
        carry = 0
        for j in range(s):
            carry,sum = BinaryHelper.addc(int(t[-1-j]) , int(ap_bin[-1-j])*int(bp_bin[-1-i]) ,carry)
            t[-1-j] = sum

        carry, sum = BinaryHelper.addc(int(t[-1-s]),carry)
        t[-1-s] = sum
        t[-1-(s+1)] = carry
        carry = 0

        m = (int(t[-1])*int(np[-1])) % (2 ** w)

        for j in range(s):
            carry, sum = BinaryHelper.addc(int(t[-1-j]), int(m)*int(n_bin[-1-j]), carry)
            t[-1-j] = sum

        carry, sum = BinaryHelper.addc(int(t[-1-s]), carry)
        t[-1-s] = sum
        t[-1-(s+1)] = t[-1-(s+1)] + carry

        for j in range(s+1):
            t[-1-j] = t[-1-(j+1)]
            u[-1-j] = t[-1-(j+1)]

    borrow = 0
    for i in range (s):
        borrow, diff = BinaryHelper.subc(u[-1-i],int(n_bin[-1-i]),borrow)
        t[-1-i] = diff
    borrow, diff = BinaryHelper.subc(u[-1-s],0,borrow)
    t[-1-s] = diff

    # zwrocenie wyniku algorytmu CIOS
    if borrow == 0:
        return t
    else:
        return u
```

9.4. Algorytm CIOS

```
def CIOS(ap_bin, bp_bin, n_bin, np,s, w=1):
    # inicjalizacja t oraz u
    t = [0] * (2*s+1)
    u = t.copy()

    # algorytm, w ktorym krok mnozenia i redukcji jest zintegrowany
    for i in range(s):
        carry = 0
        for j in range(s):
            carry,sum = BinaryHelper.addc(int(t[-1-j]) , int(ap_bin[-1-j])*int(bp_bin[-1-i]) ,carry)
            t[-1-j] = sum

        carry, sum = BinaryHelper.addc(int(t[-1-s]),carry)
        t[-1-s] = sum
```

```
    t[-1-(s+1)] = carry
    carry = 0

    m = (int(t[-1])*int(np[-1])) % (2 ** w)

    for j in range(s):
        carry, sum = BinaryHelper.addc(int(t[-1-j]), int(m)*int(n_bin[-1-j]), carry)
        t[-1-j] = sum

    carry, sum = BinaryHelper.addc(int(t[-1-s]), carry)
    t[-1-s] = sum
    t[-1-(s+1)] = t[-1-(s+1)] + carry

    for j in range(s+1):
        t[-1-j] = t[-1-(j+1)]
        u[-1-j] = t[-1-(j+1)]

borrow = 0
for i in range (s):
    borrow, diff = BinaryHelper.subc(u[-1-i],int(n_bin[-1-i]),borrow)
    t[-1-i] = diff
borrow, diff = BinaryHelper.subc(u[-1-s],0,borrow)
t[-1-s] = diff

# zwrocenie wyniku algorytmu CIOS
if borrow == 0:
    return t
else:
    return u
```