НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ СІКОРСЬКОГО»

ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ

Комп'ютерний практикум

Багаторозрядна арифметика

Виконала:

Литвиненко Ю.С.

Перевірила:

Пекарчук Н.А.

Київ 2020

Зміст

1. Мета роботи

Отримання практичних навичок програмної реалізації багаторозрядної арифметики; ознайомлення з прийомами ефективної реалізації критичних по часу ділянок програмного коду таметодами оцінки їх ефективності.

2. Теоретичні відомості

2.1. Представлення багаторозрядних чисел

Арифметичні дії над цілими числами у обчислюваних середовищах не викликають жодних труднощів, поки розміри самих чисел не перевищують розміри регістрів процесору: всі відповідні операції реалізовані в інструкціях процесору. Однак в задачах математичного моделювання та криптографії часто потрібні числа, довжина яких в десятки та сотні разів перевищує розмір регістрів. В цьому випадку потрібні спеціальні типи даних та алгоритми, які дозволяють обробляти числа таких розмірів.

Натуральне число N завжди можна представити у системі числення із основою β:

$$N = \overline{a_{n-1}a_{n-2}\dots a_1a_0}_{\beta} = a_{n-1}\beta^{n-1} + \dots + a_1\beta + a_0$$

де $n = |\log_\beta N|$ – довжина числа у даній системі числення, $a_i \in 0{,}1, ..., \beta - 1$ – окремі цифри даного представлення. Зручним та природним способом зберігання таких чисел виявляються звичайні масиви.

Для такого запису натуральних чисел існують розроблені ще у Середньовіччя алгоритми – чіткі послідовності дій, виконання яких гарантовано призводить до потрібного результату. В наступних розділах будуть розглянуті алгоритми для основних арифметичних операцій.

У сучасних обчислювальних архітектурах, побудованих на двійковому принципі, зручно обирати в якості основи системи числення степені двійки: $\beta = 2^w$, де значення w залежить від конкретних особливостей архітектури та реалізації алгоритмів. Так, у 32-розрядних обчислювальних архітектурах найпопулярніші значення $w = 1$, $w = 16$ та $w = 32$. При $w = 32$ одна цифра займає один регістр процесора, тому обчислювальні ресурси використовуються максимально ефективно, а алгоритми працюють максимально швидко; однак якщо арифметичні алгоритми реалізуються без використання асемблеру, потрібні додаткові доволі хитромудрі засоби контролю переповнення регістрів. При $w = 1$ числа представляються у звичайній двійковій системі числення, для якої всі алгоритми мають напрочуд прості вигляд та реалізацію; однак в цьому випадку кожен регістр буде нести лише один значущий біт числа (проти 32-х у попередньому випадку), тобто виконання арифметичних алгоритмів значно уповільниться саме по собі. Випадок $w = 16$ (одна цифра займає половину регістру) дозволяє реалізовувати алгоритми без переповнення регістрів, а тому може розглядатись як компроміс між простотою та швидкістю реалізації.

Також зауважимо, що у бібліотеках роботи із багаторозрядними числами можуть використовуватись такі варіанти представлення чисел.

1) Числа можуть розглядатись *знакові* або *беззнакові*. В першому випадку алгоритми можуть оперувати від'ємними числами наряду із додатними, однак це потребує додаткових ресурсів; в другому випадку алгоритми зазвичай працюють швидше, однак якщо в процесі обчислення виникає від'ємне число, ситуація вважається помилковою та зупиняє всі обчислення.

2) Числа можуть розглядатись із *довільною* або *фіксованою довжиною*. Перший випадок більш гнучкий, оскільки на зберігання чисел виділяється лише необхідна кількість пам'яті, а результати обчислень завжди будуть коректними (хоча можуть виявитись дуже довгими); однак контроль довжин чисел та їх нормалізація (видалення старших нулів) в цілому уповільнюють роботу алгоритмів. В другому випадку на кожне число незалежно від конкретного значення виділяється фіксована кількість пам'яті, а алгоритми працюють максимально швидко, однак результати всіх обчислень автоматично «обрублюються» по довжині числа: скажімо, якщо бібліотека підтримує багаторозрядні числа довжиною до 2048 біт, то всі арифметичні операції фактично виконуються за модулем 2^{2048}. За неакуратного використання це може призвести до логічних помилок.

3) При зберіганні окремих цифр в масиві може використовуватись формат *найменшої* (little endian) або *найбільшої* (big endian) *значущої цифри*. В першому випадку цифри числа зберігаються в масиві від найменш значущої до найстаршої; зручність цього способу полягає в тому, що індекси елементів масиву збігаються із відповідними степенями основи, тобто цифра a_i лежатиме у i-тій комірці, причому ці індекси не змінюються під час обробки. В другому випадку цифри числа зберігаються від найстаршої до наймолодшої цифри, тобто нульова комірка містить цифру a_{n-1}, перша – a_{n-2} тощо. В такому підході цифри в пам'яті зберігаються в такий спосіб, в який їх сприймає людина, що спрощує строкову інтерпретацію, вивід на екран тощо; однак з точки зору реалізації алгоритмів формат big endian може нести додаткові ускладнення.

Хоча модель числа із фіксованою довжиною на перший погляд здається дивною, саме вона використовується у сучасних процесорах. Скажімо, якщо у 32-бітній програмі, написаною мовою C, перемножити два 32-бітних цілих числа, результат також буде 32-бітним.

У бібліотеках багаторозрядної арифметики загального призначення для більшої гнучкості майже завжди використовується представлення чисел як знакових із довільною довжиною. Однак для криптографічних застосувань арифметичні задачі є вужчими, а швидкість є критичним параметром, тому в криптографічних бібліотеках найчастіше використовується представлення чисел як беззнакових із фіксованою довжиною. Надалі ми будемо виходити саме із такого представлення чисел.

Також для спрощення всіх алгоритмів ми будемо вважати, що всі числа зберігаються у форматі little endian.

2.2. Додавання та віднімання багаторозрядних чисел

Для додавання та віднімання багаторозрядних чисел використовуються алгоритми, які ми ще зі школи знаємо як алгоритми «в стовпчик». Отже, нехай дано два числа A та B фіксованої довжини n у системі числення із основою $\beta = 2^w$:

$$A = a_{n-1}\beta^{n-1} + \cdots + a_1\beta + a_0$$

$$B = b_{n-1}\beta^{n-1} + \cdots + b_1\beta + b_0$$

і необхідно обчислити їх суму $C = A + B$. Зазвичай сума двох чисел може бути на одну цифру довша за довжину аргументів (з'являється значуща цифра n_c); у моделі із фіксованою довжиною ця цифра або нехтується, або повертається окремим аргументом.

Додавання чисел виконується поцифрово, із використанням додаткової змінної carry, що містить так званий *біт переносу*: частину суми двох цифр, що виходить за допустимі межі для цифри, а тому повинен додаватись до наступної цифри результату. Алгоритм додавання можна подати у вигляді такої процедури.

Процедура LongAdd (A, B, C, carry)

Вхід: багаторозрядні числа A, B

Вихід: багаторозрядне число $C = A + B$; біт переносу carry, що виходить за довжину C.

```
carry := 0;
for i := 0 to n-1 do:
    temp := a[i] + b[i] + carry;
    c[i] := temp mod 2^w;
    carry := temp / 2^w;
return C, carry
```

Аналогічним чином будується процедура віднімання двох багаторозрядних чисел; вона буде використовувати *біт запозичення* borrow, який показує, що у молодших розрядах виникла від'ємна різниця, яку потрібно компенсувати за рахунок даного розряду. Якщо по закінченню процедури borrow не дорівнює нулю, то в процедурі віднімалось більше число від меншого; в залежності від реалізації ця ситуація або повинна оброблятись штатним чином, або викликати помилку.

Процедура LongSub (A, B, C, borrow)

Вхід: багаторозрядні числа A, B

Вихід: багаторозрядне число $C = A + B$; фінальний біт запозичення borrow.

```
borrow := 0;
for i := 0 to n-1 do:
    temp := a[i] - b[i] - borrow;
    if temp >= 0 then:
        c[i] := temp;
```

```
            borrow := 0;
        else:
            c[i] := 2^w + temp;
            borrow := 1;
    return C, borrow
```

2.3. Порівняння багаторозрядних чисел

Операція порівняння чисел часто виникає в багатьох алгоритмах; від її ефективної реалізації також залежить загальна швидкість роботи арифметичних бібліотек. Реалізація операції порівняння може бути виконана в різний спосіб. Найпростіший варіант – це використання описаної вище процедури LongSub: дійсно, після виконання цієї процедури за значенням біту borrow можна зробити висновок про відносні значення аргументів: якщо borrow дорівнює нулю, то $A \geq B$, а якщо одиниці, то $A < B$.

Часто, однак, за результатом порівняння необхідно чітко виокремити всі три випадки $A > B$, $A = B$ та $A < B$. У моделі з фіксованою довжиною іноді простіше виконувати порівняння чисел згідно «шкільного» визначення.

Процедура LongCmp(A, B, r)

Вхід: багаторозрядні числа A, B

Вихід: результат порівняння r: 0, якщо числа рівні; 1, якщо $A > B$; –1, якщо $A < B$.

```
    i := n-1;
    while (a[i] = b[i]) do:
        i := i-1;
    if (i = -1) then:
        return 0;
    else:
        if a[i] > b[i] then:
            return 1
        else:
            return -1;
```

2.4. Множення та піднесення до квадрату багаторозрядних чисел

Множення багаторозрядних чисел відбувається дуже подібно до додавання. Дійсно, розглянемо добуток двох багаторозрядних чисел:

$$A \cdot B = (A \cdot b_{n-1})\beta^{n-1} \ + \ \cdots \ + \ (A \cdot b_1)\beta \ + \ A \cdot b_0$$

Бачимо, що для обчислення добутку необхідно навчитись множити багаторозрядні числа на одну цифру та на степінь β. Однак в нашій моделі чисел множення на степені β відбувається майже миттєво: по суті, це лише зсув комірок відповідного масиву цифр!

Таким чином, ми можемо зосередитись на першій підпроцедурі – множенню на одну цифру. Для спрощення опису відійдемо від моделі із фіксованою довжиною числа та будемо вважати, що, в загальному випадку, множення двох n-розрядних чисел дає 2n-розрядний результат.

Процедура LongMulOneDigit (A, b, C)

Вхід: багаторозрядне число A довжини n, цифра b.

Вихід: багаторозрядне число $C = A \cdot b$ довжини $n + 1$.

```
carry := 0;
for i := 0 to n-1 do:
    temp := a[i] * b + carry;
    c[i] := temp & (2w – 1);
    carry := temp » w;
C[n] := carry;
return C
```

Відповідно, процедура множення двох багаторозрядних чисел LongMul буде використовувати допоміжні підпроцедури LongMulOneDigit, описану вище, та LongShiftDigitsToHigh, яка зсуває комірки масиву цифр у бік старших індексів.

Процедура LongMul (A, B, C)

Вхід: багаторозрядні числа A, B довжини n.

Вихід: багаторозрядне число $C = A \cdot B$ довжини 2n .

```
C := 0;
for i := 0 to n-1 do:
    temp := LongMulOneDigit(A, b[i]);
    LongShiftDigitsToHigh(temp, i);
    C := C + temp;
return C
```

Піднесення чисел до квадрату, з одного боку, може бути обчислене як звичайне множення числа на само себе. З іншого боку, під час піднесення до квадрату серед одноцифрових добутків майже половина буде дублюватись; відштовхуючись від цього, можна побудувати реалізацію процедури LongSquare, що буде майже вдвічі швидшою за LongMul.

2.5. Ділення багаторозрядних чисел

Ділення є однією з самих складних арифметичних операцій, оскільки навіть у простому варіанті «в стовпчик» вимагає виконання багатьох інших дій та пошукових евристик (знаходження цифр частки). В криптографічних застосуваннях намагаються зменшити кількість використовуваних операцій ділення, задля чого розробляються спеціальні алгоритми медулярної арифметики.

Однак повністю позбавитись ділення майже неможливо. Нижче наводиться один з найпростіших варіантів алгоритму ділення, що оперує із багаторозрядними числами у двійковій формі запису; зауважимо, що числа у системі числення із основою $\beta = 2^w$ легко переводяться у двійкову форму: достатньо кожну цифру перевести у двійковий запис, виділивши на неї w біт, а потім склеїти результати у порядку старшинства. Обернений перехід також не вимагає особливих зусиль.

Алгоритм, що пропонується, має неофіційну назву «зсувай@віднімай», оскільки основними його кроками є зсув дільника на максимально можливу кількість біт в сторону старших розрядів та віднімання його від подільного. Фактично цей алгоритм описує ділення в стовпчик у двійковій системі числення, однак використовує всі переваги останнього – зокрема, спрощення всіх операцій.

Процедура LongDivMod (A, B, Q, R)

Вхід: багаторозрядні числа A, B.

Вихід: багаторозрядні частка Q та остача від ділення R: $A = B \cdot Q + R, 0 \leq R < B$.

```
k := BitLength(B);
R := A;
Q := 0;
while R >= B do:
    t := BitLength(R);
    C := LongShiftBitsToHigh(B, t – k);
    if R < C then:
        t := t – 1;
        C := LongShiftBitsToHigh(B, t – k);
    R := R – C;
    Q := Q + 2^(t˘k);
return Q, R
```

2.6. Піднесення багаторозрядних чисел до багаторозрядного степеня

Остання базова арифметична операція, яку ми ще не розглядали – це піднесення до степеня. Для зручності в цьому розділі ми також відходимо від моделі із фіксованою довжиною числа і

вважатимемо, що результат піднесення до степеню повертається потрібної (порівняно великої) довжини.

Прямолінійний спосіб обчислення виразу A^B передбачає B раз перемножити майбутній результат на число A; цей спосіб вимагає B множень багаторозрядних чисел (де само число B чималеньке), а тому він не підходить для практичного застосування.

Значно ефективніший метод піднесення до степеня дає так звана *схема Горнера*, перенесена на цю задачу з задачі ефективного обчислення поліномів у точці. Схема Горнера має декілька можливих реалізацій та працює із двійковим записом степеня B.

Нехай $B = b_{m-1}2^{m-1} + \cdots + b_1 2 + b_0$, де $b_i \in \{0,1\}$ – окремі біти числа B. Тоді можемо записати:

$$A^B = \prod_{i=0}^{m-1} \left(A^{2^i}\right)^{b_i}$$

Бачимо, що результат можна одержати шляхом множення послідовних возведень числа A до квадрату, причому біти b_i фактично вказують, включається поточна степінь до результату чи не включається. Звідси маємо такий алгоритм піднесення до степеня.

Процедура LongPower1 (A, B, C)

Вхід: багаторозрядні числа A, B; B задане двійковим записом $B = b_{m-1}2^{m-1} + \cdots + b_1 2 + b_0$.
Вихід: багаторозрядне число $C = A^B$.

```
C := 1;
for i := 0 to m-1 do:
    if b[i] = 1 then:
        C := C * A;
    A := A * A;
return C
```

Бачимо, що схема Горнера виконує m піднесень до квадрату та не більш ніж m множень, тобто усього $\leq 2logB$ багаторозрядних множень (на відміну від прямолінійного способу, яке вимагає B множень).

Інший варіант схеми Горнера базується на такому представленні піднесення до степеня:

$$A^B = \left(\ldots\left(\left(A^{b_{m-1}}\right)^2 \cdot A^{b_{m-2}}\right)^2 \cdot \ldots\ldots A^{b_1}\right)^2 \cdot A^{b_0}$$

В даному варіанті ми йдемо не від молодших бітів B, а від старших, на кожному кроці підносимо до квадрату результат та, якщо потрібно, множимо його на A. Маємо таку процедуру.

Процедура LongPower2 (A, B, C)

Вхід: багаторозрядні числа A, B; B задане двійковим записом $B = b_{m-1}2^{m-1} + \cdots + b_1 2 + b_0$.
Вихід: багаторозрядне число $C = A^B$.

```
C := 1;
```

```
for i := m-1 to 0 do:
    if b[i] = 1 then:
        C := C * A;
    if i <> 0 then:
        C := C * C;
return C
```

Складність цієї схеми Горнера така само, як і попередньої; однак процедура LongPower2 має дві потенційні переваги над LongPower1. По-перше, вона використовує множення на число A, що є незмінним в ході роботи, і цей крок за певних умов можна оптимізувати. По-друге, саме цей варіант схеми Горнера узагальнюється до ще більш ефективних *віконних методів* піднесення до степеня.

Дійсно, представимо число B у системі числення із основою $\beta' = 2^t$ для деякого невеликого значення t («вікна») наприклад, $t = 4$. Тоді

$$A^B = \left(\ldots \left(\left(A^{b_{m-1}} \right)^{2^t} \cdot A^{b_{m-2}} \right)^{2^t} \cdot \ldots \cdot A^{b_1} \right)^{2^t} \cdot A^{b_0}, b_i \in \{0, \ \ldots, \ 2^t - 1\}$$

Таблицю степенів A до $(2^t - 1)$-того легко обчислити заздалегідь. Користуючись нею, ми будемо виконувати не більше одного множення на кожні t бітів числа B – в той час як в звичайній схемі Горнера у нас було в середньому одне множення на два біти. Таким чином, абсолютна складність віконного методу зменшується за рахунок використання додаткової пам'яті та передобчислень.

Процедура LongPowerWindow (A, B, C)

Вхід: багаторозрядні числа A, B; B задане у системі числення із основою $\beta' = 2^t$,
$B = b_{m-1}2^{t(m-1)} + \cdots + b_1 2^t + b_0$
Вихід: багаторозрядне число $C = A^B$.

```
C := 1;
D[0] := 1; D[1] := A;
for i := 2 to (2t – 1) do:
    D[i] := D[i – 1] * A;
for i := m-1 to 0 do:
    C := C * D[b[i]];
    if i <> 0 then:
        for k := 1 to t do:
            C := C * C;
return C
```

Віконні методи надалі були розвинені у методи із використанням динамічних вікон, адаптивні віконні методи, а також найбільш ефективні (але, на жаль, поки що не застосовні на практиці в загальному випадку) методи із використанням адитивних ланцюгів.

2.7. Алгоритм Евкліда

Алгоритм Евкліда обчислює найбільший спільний дільник двох чисел $d = gcd(a,b)$ шляхом ітеративної процедури, яка ґрунтується на такому факті: якщо $a \geq b$, то $gcd(a,b) = gcd(b,a-b)$. Звідси одразу випливає, що $gcd(a,b) = gcd(b,amodb)$, і процедура обчислення НСД задається наступним чином.

Нехай $r_0 = a, r_1 = b$; обчислюємо послідовність (r_i) для $i \geq 2$ шляхом ділення з остачею:

$$r_0 = r_1 \cdot q_1 + r_2,$$
$$r_1 = r_2 \cdot q_2 + r_3,$$
$$\dots$$
$$r_{s-2} = r_{s-1} \cdot q_{s-1} + r_s$$
$$r_{s-1} = r_s \cdot q_s$$

Якщо на відповідному кроці виявилось, що $r_{s+1} = 0$, то $d = r_s$.

Складність алгоритму Евкліда є лінійною по відношенню до бітової довжини n аргументів. Дійсно, найгірший випадок для роботи алгоритму – коли всі q_i =1. Цей випадок відповідає ситуації, коли a та b – два послідовні числа Фібоначчі. З формули Біне випливає, що число Фібоначчі асимптотично веде себе як $f^n \sim \phi^n$, де $\phi = \frac{\sqrt{5}+1}{2}$ – відношення «золотого перерізу», а тому алгоритм Евкліда виконає не більше ніж $|\log_\phi a| = O(\log a) = O(n)$ операцій.

Розширений алгоритм Евкліда обчислює дві додаткові послідовності (u_i) та (v_i) такі, що на кожному кроці виконується рівність $ri = u_i \cdot a + v_i \cdot b$; зокрема, для найбільшого спільного дільника матимемо $d = r_s = u_s a + v_s b$. Ці послідовності також можна обчислити рекурентно за допомогою часток q_i:

$$u_0 = 1,\ u_1 = 0,\ u_{i+1} = u_{i-1} - q_i u_i;$$
$$v_0 = 0,\ v_1 = 1,\ v_{i+1} = v_{i-1} - q_i v_i.$$

Зауважимо, що хоча алгоритм Евкліда є лінійним за кількістю операцій, він використовує операції ділення з остачею, які самі по собі є важкими. Для запобігання цього був розроблений інший алгоритм, що має назву бінарний алгоритм обчислення НСД або алгоритм Стейна. Цей алгоритм базується на таких спостереженнях:

1) якщо і a, і b – парні, то $\gcd(a,\ b) = 2\gcd(\frac{a}{2},\ \frac{b}{2})$;

2) якщо a – парне, а b – непарне, то $\gcd(a,\ b) = \gcd(\frac{a}{2},\ b)$;

3) якщо і a, і b – непарні, то $\gcd(a,b) = \gcd(min\{a,\ b\},\ |a-b|)$, причому різниця є парним числом.

На відміну від алгоритму Евкліда, бінарний алгоритм використовує лише віднімання та ділення на два, яке у двійкових архітектурах ефективно реалізується як бітовий зсув.

Отже, бінарний алгоритм можна подати у вигляді такої процедури.

```
d := 1;
while (a – парне) and (b парне) do:
    a := a / 2;
    b := b / 2;
    d := d * 2;
while (a – парне) do:
    a := a / 2;
while (b <> 0) do:
    while (b – парне) do:
        b := b / 2;
    (a, b) := (mina, b, abs(a – b))
d := d * a;
return d;
```

На кожному кроці одне з двох оброблюваних чисел скорочується на один біт (шляхом ділення на два), тому бінарний алгоритм виконає не більш ніж $2 \log a$ кроків. Це більше, ніж в класичному алгоритмі Евкліда, але використання віднімання та зсувів замість ділення на практиці робить бінарний алгоритм суттєво швидшим.

2.8. Редукція за Барреттом

В багатьох асиметричних криптографічних алгоритмах потрібно виконувати обчислення за модулем деякого натурального числа. Прямолінійний підхід полягає в тому, щоб застосовувати ділення з остачею після кожної арифметичної операції; однак, як зазначалось, ділення є дуже складною операцією, тому були розроблені спеціальні алгоритми, які б дозволяли замінити ділення на інші, більш прості операції.

Перевага таких алгоритмів відчутна тоді, коли потрібно виконувати багато обчислень за одним й тим самим модулем. В цьому випадку за рахунок деяких передобчислень вдається суттєво зекономити на обчисленнях в кожному конкретному випадку, що призводить до значного пришвидшення обчислень в цілому.

Розглянемо перший з таких методів – *алгоритм модулярної редукції Барретта.*

Отже, нехай дано багаторозрядні числа n та x у системі числення із основою β, причому довжина x вдвічі більша за n: $|n| = k$, $|x| = 2k$. Необхідно знайти частку q та остачу r від ділення x на n: $x = q \cdot n + r, 0 \leq r < n$.

Алгоритм Барретта передбачає «вгадування» частки q – точніше, її оцінку. Дійсно, можемо записати:

$$\frac{x}{n} = \frac{x}{\beta^{k-1}} \cdot \frac{\beta^{2k}}{n} \cdot \frac{1}{\beta^{k+1}}$$

$$q = \left\lfloor \frac{x}{n} \right\rfloor = \left\lfloor \frac{\frac{x}{\beta^{k-1}} \cdot \frac{\beta^{2k}}{n}}{\beta^{k+1}} \right\rfloor \geq \left\lfloor \frac{\left\lfloor \frac{x}{\beta^{k-1}} \right\rfloor \cdot \left\lfloor \frac{\beta^{2k}}{n} \right\rfloor}{\beta^{k+1}} \right\rfloor = \widetilde{q}$$

причому у виразі для $\widetilde{q}$ ділення на степені β насправді є лише відкиданням останніх цифр числа (тобто ніякого ділення там не відбувається), а множник $\mu = \left\lfloor \frac{\beta^{2k}}{n} \right\rfloor$ не залежить від x і тому може бути передобчислений. Барретт довів, що якщо $x \leq n^2$, то знайдена таким чином частка $\widetilde{q}$ відрізняється від справжнього значення q не більш ніж на 2, причому в 90% випадків $\widetilde{q}$ та q взагалі збігаються.

Зауваження: для коректного обчислення μ необхідно правильно визначити степінь β^{2k}, яка має $2k + 1$ цифру в записі (на одну більше, ніж в записі x); це може бути критичним при реалізації у моделі із фіксованою довжиною числа.

Алгоритм редукції за Барреттом можна подати у вигляді такої процедури.

Процедура BarrettReduction (x, n, μ, r)
Вхід: багаторозрядні числа x, n, передобчислене значення $\mu = \left\lfloor \frac{\beta^{2k}}{n} \right\rfloor$.
Вихід: багаторозрядне число $r = x \mod n$.

```
q := KillLastDigits(x, k-1);
q := q * μ;
q := KillLastDigits(q, k+1);
r := x – q * n;
while (r >= n) do:
    r := r – n;
return r;
```

Бачимо, що редукція за Барретом для обчислення остачі використовує не ділення, а множення (та декілька зсувів та віднімань), що значно пришвидшує обчислення за модулем.

Покажемо застосування редукції за Барреттом на прикладі схеми Горнера. Саме в схемі Горнера під час піднесення до степеня потрібно виконувати багато операцій множення за одним модулем, що є необхідною передумовою для ефективного застосування редукції за Барреттом.

Процедура LongModPowerBarrett (A, B, N, C)
Вхід: багаторозрядні числа A, B, N; B задане двійковим записом $B = b_{m-1}2^{m-1} + \cdots + b_1 2 + b_0$.
Вихід: багаторозрядне число $C = A^B \mod N$.

```
C := 1;
μ := LongShiftDigitsToHigh(1, 2*k) / n;
for i := 0 to m-1 do:
    if b[i] = 1 then:
        C := BarrettReduction(C * A, N, μ);
    A := BarrettReduction (A * A, N, μ);
return C
```

2.9. Редукція за Монтгомері

На відміну від метода Барретта, Монтгомері запропонував для обчислення лишків взагалі перейти у іншу арифметичну систему, в якій редукція сама по собі виконується значно швидше.

Нехай n – непарне число, а $R > n$ – число, взаємно просте із n (зазвичай для зручності обирають $R = 2^t$). *Лишком Монтгомері* числа x називають вираз $mont(x) = xR \mod n$. *Функцією редукції Монтгомері* називають функцію $redc(x) = x \cdot R^{-1} \mod n$, де R^{-1} – число, обернене до R за модулем n. Система лишків Монтгомері утворює арифметику із такими операціями:

$$mont(x \pm y) = mont(x) \pm mont(y),$$

$$mont(x \cdot y) = redc(mont(x) \cdot mont(y)),$$

і треба мати на увазі, що $redc(mont(x)) = x \mod n$. Таким чином, довільний арифметичний алгоритм, який використовує лише додавання, віднімання та множення, може бути переписаний для системи лишків Монтгомері таким чином:

1) всі вхідні змінні x та константи замінюються на лишки Монтгомері $mont(x)$;

2) всі множення $x \cdot y$ замінюються на $redc(x \cdot y)$;

3) всі вихідні змінні z замінюються на $redc(z)$.

Наприклад, схема Горнера у системі лишків Монтгомері буде виглядати так:

Процедура LongModPowerMontgomery (A, B, N, C)

Вхід: багаторозрядні числа A, B, N; B задане двійковим записом $B = b_{m-1}2^{m-1} + \cdots + b_1 2 + b_0$.
Вихід: багаторозрядне число $C = A^B \mod N$.

```
A := mont(A);
C := mont(1);
for i := 0 to m-1 do:
    if b[i] = 1 then:
        C := redc(C * A);
    A := redc(A * A);
return redc(C)
```

Головною перевагою методу Монтгомері є обчислення функції $redc(x)$, яке може бути виконане суттєво швидше, ніж звичайне ділення з остачею. Операція ділення залишається лише при початкових обчисленнях $mont(x)$.

Покажемо схематично, як швидко обчислювати значення $redc(x)$.

1) За допомогою розширеного алгоритму Евкліда знаходяться такі числа R^{-1} та n', що $RR^{-1} - nn' = 1$ (зауважимо, що R^{-1} – це обернений до R за модулем n). Цей крок є передобчисленням.

2) Обчислити $u = x + (x \cdot n' \mod R) \cdot n$. Оскільки внутрішнє множення береться за модулем $R = 2^t$, то для його обчислення достатньо взяти по t біт аргументів, що прискорює обчислення; також відмітимо, що операція $\mod R$ – це просто одержання останніх t біт числа.

3) Обчислити $u = u/R$ (тобто у числа u відкидаються останні t біт). Якщо $u \geq n$, то $u = u - n$. Повернути u як $redc(x)$.

Цю процедуру також можна пришвидшити, якщо звести відповідні обчислення до окремих цифр результату.

3. Завдання до комп'ютерного практикуму

А) Згідно варіанту розробити клас чи бібліотеку функцій для роботи з m-бітними цілими числами. Бібліотека повинна підтримувати числа довжини до 2048 біт.

Повинні бути реалізовані такі операції:

1) переведення малих констант у формат великого числа (зокрема, 0 та 1);

2) додавання чисел;

3) віднімання чисел;

4) множення чисел, піднесення чисел до квадрату;

5) ділення чисел, знаходження остачі від ділення;

6) піднесення числа до багаторозрядного степеня;

7) конвертування (переведення) числа в символьну строку та обернене перетворення символьної строки у число; обов'язкова підтримка шістнадцяткового представлення, бажана – десяткового та двійкового.

8)обчислення НСД та НСК двох чисел;

9) додавання чисел за модулем;

10) віднімання чисел за модулем;

11) множення чисел та піднесення чисел до квадрату за модулем;

12) піднесення числа до багаторозрядного степеня d по модулю n.

Бажано реалізувати такі операції:

1) визначення номеру старшого ненульового біта числа;

2) бітові зсуви (вправо та вліво), які відповідають діленню та множенню на степені двійки.

Мова програмування, семантика функцій та спосіб реалізації можуть обиратись довільним чином.

Модулярну арифметику рекомендовано реалізовувати на базі редукції Баррета, піднесення до степеня – на базі схеми Горнера. Мова програмування, семантика функцій та спосіб реалізації можуть обиратись довільним чином.

Б) Проконтролювати коректність реалізації алгоритмів; наприклад, для декількох багаторозрядних a,b,c,n. перевірити тотожності:

1) $(a + b) \cdot c = c \cdot (a + b) = a \cdot c + b \cdot c$

2) $n \cdot a = \underbrace{a + a + \cdots + a}_{n}$, де n повинно бути не менш за 100; і таке інше.

Продумати та реалізувати свої тести на коректність.

В) Обчислити середній час виконання реалізованих арифметичних операцій. Підрахувати кількість тактів процесора (або інших одиниць виміру часу) на кожну операцію. Результати подати у вигляді таблиць або діаграм.

Окрім основного завдання, ви також можете виконати додаткове завдання згідно варіанту.

Варіант	Додаткове завдання
6	Реалізація модулярного піднесення до степеню віконним методом (4 біти), порівняння ефективності із методом Горнера

4. Посилання на github

https://github.com/LytvynenkoJ/Lytvynenko_FI84_lab1

5. Результати тестів

Були проведені деякі додаткові перевірки:

1) $(A \cdot B)/B = A$

2) $A = Q * B + R$

3) $(A + B) \cdot C = A \cdot B + B \cdot C$

Тест 1

A	$458AED51BC84868D64E84698A777AD161E1CBBA4984620150AE20065$ $C104465B155AD54F65E1651DA1E1FA16E1615698484BC846E4F84865$ $4D8F4F8484655440$
B	$FE684D0565C4B986D61A61CB48984D94F89E486A841650062DF0E654$ $98CB051495A4894ED65F1641D56E64116511C65B68A08961549510DFE69$ $8A489D44E894F$
$A + B$	$143f33a57224940143b02a863f00ffaab16bb040f1c5c701b38d2e6ba59cf4b6f$ $aaff5e9e3c407b5f77505e2846731cf3b0ec51a8398d59453427f40e58b3dd8f$
$A - B$	Negative number
$A \cdot B$	$451c2cd3c2fd8c0876f4d22584501bd13260bcf22191b2f75a1fd6524b4e$ $e716544397841d439a7a2f6c339835f89356f90379b082ac9df7a35d503bba94f$ $df5bfa20ec402ea319ef05c825943d6d3c5b45ec20e7ca11511df60a497d45091$ $7d737164ba66bae43a9c4eb29a396c95ad84b89806c381599fe2fecba3c0db3fc0$
A/B	0
$A\%B$	$458aed51bc84868d64e84698a777ad161e1cbba4984620150ae20065c10$ $4465b155ad54f65e1651da1e1fa16e1615698484bc846e4f848654d8f4f$ 8484655440
$(A \cdot B)/B$	$458aed51bc84868d64e84698a777ad161e1cbba4984620150ae20065c104465b1$ $55ad54f65e1651da1e1fa16e1615698484bc846e4f848654d8f4f8484655440$
$(A + B) \cdot B$	$141ef5029247edb5c3f1e4f459b423f276bf91f039bd96163a3dac0db0293dd1$ $fa105c3b47b73d35d1e18521cc4e7b6f69858fa4ea7961b63018c1dfcc856a4ca$ $611508533536aca1da1f46b6316e36e23c31267743a4e0e7f39f293da3d4daa$ $05cb6fd830f4872100ef9dd1e95cb2992e7ee66c19a5e8f7870dd25b48ea4e621$
$B^2 = B \cdot B$	$fcd3235561814f53c8297d2016f22356399862117a47ae6c49baea88b744f6094$ $cc22c305e3038e2eeac1e848eef239f9f55809e24e97d6b5e2ecdc10dc1a6d4a17$ $2f98f324c7b02e9c2c45ced97631c87d26468c703cbd6143e84a5cf844922e94598$ $c8a88d8dd572ab2a845c5e93e56335cebad6dd35d88dde5a10cdc9a661$
$A \cdot B + B \cdot B$	$141ef5029247edb5c3f1e4f459b423f276bf91f039bd96163a3dac0db0293dd1$ $fa105c3b47b73d35d1e18521cc4e7b6f69858fa4ea7961b63018c1dfcc856a4ca$ $611508533536aca1da1f46b6316e36e23c31267743a4e0e7f39f293da3d4daa$ $05cb6fd830f4872100ef9dd1e95cb2992e7ee66c19a5e8f7870dd25b48ea4e621$

A	$D4D2110984907B5625309D956521BAB4157B8B1ECE04043249A3D379AC112E$ $5B9AF44E721E148D88A942744CF56A06B92D28A0DB950FE4CED2B41A0B$ $D38BCE7D0BE1055CF5DE38F2A588C2C9A79A75011058C320A7B661C6CE$ $1C36C7D870758307E5D2CF07D9B6E8D529779B6B2910DD17B6766A7EFE$ $E215A98CAC300F2827DB$
B	$3A7EF2554E8940FA9B93B2A5E822CC7BB262F4A14159E4318CAE3ABF5$ $AEB1022EC6D01DEFAB48B528868679D649B445A753684C13F6C3ADBAB05$ $9D635A882090FC166EA9F0AAACD16A062149E4A0952F7FAAB14A0E9D3CB$ $0BE92020DBD3B034249642182691914 8E617AF1DB66978B1FCD28F840850$ $6B79979CCBCC7F7E5FDE7$
$A + B$	$10f51035ed319bc50c0c4503b4d44872fc7de7fc00f5de863d6520e3906fc$ $3e7e8761505118c918db31aadbea5a054b13a25f259cd47c1faa7db9b76f$ $2db450861ba26c4794e8e3bfbc2924de45e47e5408536e3548a03591da05$ $56d595ab78c55149f45170f2cb7736a46976d1c09bfce4df6eab040599af$ $235968f8070e25c2$
$A - B$	$9a531eb436073a5b899ceaef7cfeee386318967d8caa2000bcf598ba51261e$ $38ae874c932360023620da0caf90cec25eb7f21c1a55a3a9f327ae7ca8796$ $34c73fc1f9e7256d38e258ee860b509506bae185e180c06cc8dfbc23316ba$ $1b357240be81b14c9ec0a25a73ae85c0049185bd4a8d7e29f9f82a7c2fbfe$ $f68174229f4$
$A \cdot B$	$30a120b609dcbe28b09ca92e12dd29d77ae6400dc22b026afb5fb945aaf$ $62b57f4e48bd299261f02bbb35dd2495b5cd2713bf0e30192dae1b33465916$ $0c8552423f0ad7fb82870920df4e9b57980ead2ada9f3ef4b5d0718ab7f105$ $3700395278998cb9ad48498d65150e3e837b0bb169d432b441424557061f83$ $8a17c65f90a31105f599bf69b87485bf9c70f51d37a417e476e372558c2678$ $2ac8c8f35c3d1227e851d8a72cd708700fd90c5e17f22c4ea15730345e56bd$ $76f04b54580813cbe306b4404c6f34bcd9840d2911e6b3cf6de3ee428c274e$ $df0a97335d8256da26fcd67ba5450593a15f6b527ece76fbbe20f7a8823476$ $14af4b7faf55086659d$
$Q = A/B$	3
$R = A\%B$	$25553a0998f4b866527585a3acb95540fe52ad3b09f6579da399233b9b4ffdf2$ $d5ad48d52df6eb9110093d74c79839a9cd851297d6cb343bd1a341e1c512486$ $1dc9cd09d18be388b61a79c8bccbc59082868c1e2c4f8e665aa60d69ea0bf6b$ $bc2bb96e47cc8e793db0b8a20937626f7db03b8da8efeabb493c2675d827762e26$
$Q * B + R$	$d4d2110984907b5625309d956521bab4157b8b1ece04043249a3d379ac112e5b9af44e$ $721e148d88a942744cf56a06b92d28a0db950fe4ced2b41a0bd38bce7d0be105$ $5cf5de38f2a588c2c9a79a75011058c320a7b661c6ce1c36c7d870758307e5d2$ $cf07d9b6e8d529779b6b2910dd17b6766a7efee215a98cac300f2827db$

N	$269D7722EA018F2AC35C5A3517AA06EAA1949059AE8240428BBFD0A8B$ $E6E2EBF91223991F80D7413D6B2EB213E7122710EDEC617460FA01$ $91F3901604619972018EBEF22D81AED9C56424014CADCC2CCDEE67D$ $36A54BFC500230CA6693ABA057B374746622341ED6D52FE5A79E686$ $0F54F197791B3FEF49FD534CB2C675B6BDB$
$(A+B)modN$	$102c16a6d0ed225693dd8c7a79e56c55cce8d4c49ce26920413599bd1f8f7418f$ $71bd53506aec5052c66e01a4ed59fc3a47baf9ea0ebefaa32aadcd4360cd2d5$ $983ea12fdcc667db3bf23820031b69677b607ec313fb8477d58b0078bfa165f$ $691ac5868185f397625521e17ce5f547bb2d29af180ce9504ee7dac1338e32c5$
$(A-B)modN$	$267ab94b78028cdb3f87dc503600d9787e5ae57081235f3919b626c015db91f9f$ $b209fdd3b37a5fa9cc14b4bd57b5b0b8b55c9d48374c9a7ca037887a6b6e88bc$ $506c94a7ce2c34c38bbe1416263976af3fa9a32a935302608f7d58660329139a$ $424541a382445d5f41ed4c9295862dd5d411774ddfb4c9cb0218de2e12fe663$
$(A\cdot B)modN$	$53fd232f16874e3ad51d332fe50c9046d77564ae7387269da227c4fb5bd50ecb55$ $64cbde924914008ed8cffa4e5cb26b405db83013b98bca2b261f5bd8554f37b55$ $f359aaac6efc5502b40a842215c8d141697d10fe389328a64ed94f366a313813$ $45b3822d0fe35598e8e7399325857283ecffe4c33bc1e66778398b7e657e$
НСД(A,B)	1
НСК(A,B)	$30a120b609dcbe28b09ca92e12dd29d77ae6400dc22b026afb5fb945aaf$ $62b57f4e48bd299261f02bbb35dd2495b5cd2713bf0e30192dae1b33465916$ $0c8552423f0ad7fb82870920df4e9b57980ead2ada9f3ef4b5d0718ab7f105$ $3700395278998cb9ad48498d65150e3e837b0bb169d432b441424557061f83$ $8a17c65f90a31105f599bf69b87485bf9c70f51d37a417e476e372558c2678$ $2ac8c8f35c3d1227e851d8a72cd708700fd90c5e17f22c4ea15730345e56bd$ $76f04b54580813cbe306b4404c6f34bcd9840d2911e6b3cf6de3ee428c274e$ $df0a97335d8256da26fcd67ba5450593a15f6b527ece76fbbe20f7a8823476$ $14af4b7faf55086659d$

A	$54E894C13F6C3ADBAB059D63C7BB262F4A14159E435A2882090FC16$ $6E952F7FAAB14A0E9D3C40FA9B93B2A5E822C18CAE3ABF5AEB1022EC$ $6D68679AF1DB66978B1FCD28F8408501DEFAB48B5288D649B445A75368$ $A9F0AAACD16A06217E5FDE749E4A0B0BED3B034249642182691914 8E61$ $706B79979CCBCC7F9200DB3A7EF25$
B	$5009E1A03EE0EF87A0BFD3C8CE0F5DD5130F81C1AEB2557F32607121$ $B233ADB260EE4E42FC13AB57BFDCC637D0BF051C001D59D302D4F71F$ $6F61FDEB49329774EF75FAA1DCE5A7A6ADCBC58E341AE70F54DFB354A8$ $7A4D86A1FA630B7FA8A400C3758674FB99D7ADD17936651B48B8D4A94A4$ $5D2758E3C35B37956BAC9BAD7EC$
$A+B$	$a4f276617e4d2a634bc5712c95ca84045d23975ff20c7e013b7032$ $889b86a5ad0c02ef2ccfd7bb017917f09652eb1de6e3c94f81b3d725e$ $645e8779a66e900eda172cd3160edf7c49d770e435ca84baa993a288b$ $331958316f11036d978ea1e80d5a2725ba6d87e1f60f787d41da4a1d8$ $f614c8a0f0808f27b7276c87d62c711$
$A-B$	$4deb321008b4b540a45c99af9abc85a370493dc94a7d302d6af5045$ $371f4a484a2652a6d7b06451f95e6426b16d13aee38e9bdbae2d37a767$ $247bc3d483d203c286d7eda722a87741df8326f4727d8bef7ac1e1e224$ $bd242b1c3d56983d59e6866f1a3bc339d886531d0bb30b48d8743cccc0$ $e523eb9087147fc952e9ed1739$
$A \cdot B$	$1a8bf57f2f92cdb38fd62ccdcae467311800e195e8692055f06070d06a60$ $ffb445862d81f07128c3af4cc776020554d225c85a887c32fdac4b3d18ceeb$ $eed14d1acb03612b824929d80886288249fe7b05fc60e4543c17c90251ea8d$ $dc041da5a4ea2bd1f49add7d398ea437a6aefe1995aad8b3460f4618086d8e$ $30eed4b2d2031b951f8812961848318dc32ddaf7392e9b2ad011e3ece6ae48cd02$ $ee0239670cd4084c291494eccb540aa06e9515e0b4d59b422b41d66883591ac2cf$ $4c1c1831696d66e5f66c10053625dec4c61aa5f8256603db4f671cae9f8961cd66c$ $deadf9818638639026457 1b88e75f2581a2b4cdcfa4959c98b5796fa02d768a891c$
$(A*B)/B$	$54e894c13f6c3adbab059d63c7bb262f4a14159e435a2882090fc166e952f7fa$ $ab14a0e9d3c40fa9b93b2a5e822c18cae3abf5aeb1022ec6d68679af1db66978b1fc$ $d28f8408501defab48b5288d649b445a75368a9f0aaacd16a06217e5fde749e4a0b0$ $bed3b034249642182691914 8e61706b79979ccbcc7f9200db3a7ef25$
N	$3A7EF2554E8940FA9B93B2A5E822CC7BB262F4A14159E4318CAE3ABF5AE$ $B1022EC6D01DEFAB48B528868679D649B445A753684C13F6C3ADBAB059D$ $635A2882090FC166EA9F0AAACD16A062149E4A0952F7FAAB14A0E9D3CB0$ $BE9200DBD3B034249642182691914 8E617AF1DB66978B1FCD28F8408506$ $B79979CCBCC7F7E5FDE7$

$(A+B) mod N$	$2ff491b6e13aa86e149e0be0c584eb0cf85dae1d6f58b59e2213bd09e5b0856$ $73328eb6eda6ea45c6847215b89b49531f95c45ff34feb02eefdd3cd3b297fcd$ $b81efffff5c22d8a22a70364a1a20143904a944d261f145b09b573ec3521d189b63$ $7a91e420e83b5ec5331994c674ab33ddf50f5c0904fa99bf87d8fd388d96cb43$
$(A-B) mod N$	$4deb321008b4b540a45c99af9abc85a370493dc94a7d302d6af5045371f4a484a26$ $52a6d7b06451f95e6426b16d13aee38e9bdbae2d37a767247bc3d483d203c286d$ $7eda722a87741df8326f4727d8bef7ac1e1e224bd242b1c3d56983d59e6866f$ $1a3bc339d886531d0bb30b48d8743cccc0e523eb9087147fc952e9ed1739$
$(A \cdot B) mod N$	$3a15c1d10e51aa22fd52ea838a2cca8e45fdc4659edecb05ab412de80c7a3e18b30$ $cf384ae9f45e189dba5f72530b1d7f73d012b0e1197128a33929368dc1259e71$ $5e9626ce242ab460ff5f8f20dbc638e48fa7989757849f7b0d16f9009a51275d$ $4bf0fe8e86996ffec00c94dc98ad032e99cd778cfff0beef18d893622b2538$
НСД(A,B)	1
НСК(A,B)	$1a8bf57f2f92cdb38fd62ccdcae467311800e195e8692055f06070d06a60$ $ffb445862d81f07128c3af4cc776020554d225c85a887c32fdac4b3d18ceeb$ $eed14d1acb03612b824929d80886288249fe7b05fc60e4543c17c90251ea8d$ $dc041da5a4ea2bd1f49add7d398ea437a6aefe1995aad8b3460f4618086d8e$ $30eed4b2d2031b951f8812961848318dc32ddaf7392e9b2ad011e3ece6ae48cd02$ $ee0239670cd4084c291494eccb540aa06e9515e0b4d59b422b41d66883591ac2cf$ $4c1c1831696d66e5f66c10053625dec4c61aa5f8256603db4f671cae9f8961cd66c$ $deadf98186386390264571b88e75f2581a2b4cdcfa4959c98b5796fa02d768a891c$

Тест 4 (перевірка НОД та НОК на малих числах)

A	$A9F7C272DC9$
B	$388AF80A806$
$A \cdot B$	$258a76c6ef72e1ea76fab6$
НСД(A, B)	$c1578ad$
НСК(A,B)	$31b5030fd35ce4e$

Тест 5 (перевірка роботи степіня)

A	125
B	40
A^B(Горнер)	$1613949fac142718b9aa8be04dcbff5929e289e9ec38165aa485eb2ece04b2302f9c85$ $63419d57e0b8ae36540fede68465efa8726d1b2f047347f2f1a218fc5f4701$
A^B(віконний)	$1613949fac142718b9aa8be04dcbff5929e289e9ec38165aa485eb2ece04b2302f9c85$ $63419d57e0b8ae36540fede68465efa8726d1b2f047347f2f1a218fc5f4701$
C	$1cd7$
$A^B \mod C$	$ad8$

6. Результати вимірів часу

У таблицях нижче наведений середній час роботи кожної операції в наносекундах.

5.1. Основне завдання

Операції з першого практикума

кількість біт	сума	різниця	добуток	ділення
$n = 128$	4500	2300	3900	1100
$n = 512$	3907	2185	5932	2152
$n = 1024$	9600	3900	15000	2530

Операції з другого практикума

кількість біт	сума	різниця	добуток	НСД	НСК
$n = 128$	18131600	11586433	34812833	200879467	1563212533
$n = 512$	33693133	19984433	65824037	414154767	3126152166
$n = 1024$	62430333	40243467	308310200	869322700	5650153266

5.2. Додаткове завдання

Теоретично:

Час роботи схеми Горнера(у кількості операцій множення):

$T_1(n) = 2(\log_2 n + 1)$, де $\log_2 n$ - довжина степіня у двійковій системі числення.

Час роботи віконного методу(у кількості операцій множення):

$T_2(n) = 2^\omega + \omega(\log_{2^\omega} n + 1) + (\log_{2^\omega} n + 1) = 2^\omega + (\omega + 1)(\log_{2^\omega} n + 1)$

$\log_{2^\omega} n$ - довжина степіня у системі числення з основою $\beta = 2^\omega$

Для $\omega = 4$ (число прадставлено у шістнадцятковій системі числення):

$T_2(n) = 16 + 5(\log_{16} n + 1)$

Наприклад для $\log_2 n = 1024$:

$T_1(n) = 2050$ операцій множення

$T_2(n) = 16 + 5(256 + 1) = 1301$ операцій множення

Тож для великих значень $\log_2 n$ віконний метод дає значний вийграш у часі.

За це ми платимо пам'яттю (для схеми Горнера додатковой пам'яті не потрібно, а для віконного методу: $S(n) = 2^\omega$)

кількість біт	схема Горнера	віконний метод	за модулем
$n = 8$	54146033	53286333	3640433
$n = 12$	8941811950	6860509300	3775750

Як ми бачимо навіть для невеликих значень $\log_2 n$ віконний метод все ж таки дає вийграш у часі, але не настільки суттєвий.

7. Висновок

У цих комп'ютерних практикумах (1-2) ми розглянули багаторозрядну арифметику та деякі операції з великими числами, такі як: додавання, різниця, множення, ділення, піднесення до багаторозрядного степіня. А також ці операції за модулем великого числа. Також було розглянуто НСД та НСК.

Порівняли алгоритми піднесення до степіня (схема Горнера, віконний метод) за часом роботи та пам'яттю. В результаті отримали, що віконний метод дає переваги у часі, але використовує більшу кількість пам'яті.