Міністерство освіти і науки України

Національний технічний університет України

"Київський політехнічний інститут імені Ігоря Сікорського"

Навчально-науковий Фізико-технічний інститут

Спеціальні розділи обчислювальної математики

Комп'ютерний практикум №2

Багаторозрядна модулярна арифметика

Виконав:
студент групи ФІ-12
Юрчук Олексій

Київ 2024

Тема: Багаторозрядна модулярна арифметика

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

Завдання:

А) Доопрацювати бібліотеку для роботи з m-бітними цілими числами, створену на комп'ютерному практикумі №1, додавши до неї такі операції:

1) обчислення НСД та НСК двох чисел;
2) додавання чисел за модулем;
3) віднімання чисел за модулем;
4) множення чисел та піднесення чисел до квадрату за модулем;
5) піднесення числа до багаторозрядного степеня d по модулю n

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

1. $(a+b)\cdot c \equiv c\cdot(a+b) \equiv a\cdot c + b\cdot c \pmod n$;
2. $n\cdot a \equiv \underbrace{a+a+\ldots+a}_{n} \pmod m$, де n повинно бути не менш за 100;
3. $a^{\varphi(n)} = 1 \bmod n$ (за умови $\gcd(a,n)=1$).

Перевірити останню тотожність для простого n; перевірити для $n = 3^k$ та довільного $a \,\bar{\vdots}\, 3$: $\varphi(3^k) = 3^k - 3^{k-1} = 2\cdot 3^{k-1}$, отже, має виконуватись $a^{\varphi(3^k)} = a^{2\cdot 3^{k-1}} = 1 \bmod 3^k$.

Продумати та реалізувати свої тести на коректність.

Для перевірки роботи операцій із простими числами можна використовувати заздалегідь відомі прості числа (наприклад, числа Мерсенна).

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.

Теоретичні відомості:

- Алгоритм Евкліда

2.1. Алгоритм Евкліда

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

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

$$\begin{aligned} r_0 &= r_1 q_1 + r_2, \\ r_1 &= r_2 q_2 + r_3, \\ &\ldots \\ r_{s-2} &= r_{s-1} q_{s-1} + r_s; \\ r_{s-1} &= r_s q_s. \end{aligned}$$

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 $\left|\log_\phi a\right| = O(\log a) = O(n)$ операцій.

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:   // виокремлення загальної парної частини
      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) := (min{a, b}, abs(a - b))

d := d * a;
return d;
```

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

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

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

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.

• Редукція за Барреттом

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: $x = qn + r$, $0 \le r < n$.

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

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

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

причому у виразі для $\hat{q}$ 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 $x \le n^2$, то знайдена таким чином частка $\hat{q}$ відрізнятиметься від справжнього значення q не більш ніж на 2, причому в 90% випадків $\hat{q}$ та q взагалі збігаються.

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 $r = x \bmod n$.

```
q := KillLastDigits(x, k-1);        // відкидання останніх k-1 цифри
q := q * μ;
q := KillLastDigits(q, k+1);
r := x - q * n;
while (r >= n) do:                   // Барретт гарантує, що цикл виконується
      r := r - n;                    //    не більше двох разів
return r;
```

Процедура LongModPowerBarrett (*A*, *B*, *N*, *C*)

Вхід: багаторозрядні числа A, B, N; B задане двійковим записом $B = b_{m-1}2^{m-1} + ... + b_1 2 + b_0$.

Вихід: багаторозрядне число $C = A^B \bmod N$.

```
C := 1;
μ := LongShiftDigitsToHigh(1, 2*k) / n;         // єдине ділення!
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
```

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 $B = b_{m-1}2^{m-1} + ... + b_1 2 + b_0$.

Вихід: багаторозрядне число $C = A^B \bmod N$.

```
C := 1;
μ := LongShiftDigitsToHigh(1, 2*k) / n;         // єдине ділення!
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
```


- ## Редукція за Монтгомері

2.3. 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 $mont(x) = xR \bmod n$. *Функцією редукції Монтгомері* називають функцію $redc(x) = x \cdot R^{-1} \bmod n$, 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 $redc(mont(x)) = x \bmod n$. 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 xy замінюються на $redc(xy)$;

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 $B = b_{m-1}2^{m-1} + ... + b_1 2 + b_0$.

Вихід: багаторозрядне число $C = A^B \bmod N$.

```
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 $u = x + (x \cdot n' \bmod R) \cdot n$. 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 $\bmod R$ – 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 $redc(x)$.

Хід роботи

Введені числа:

number_one =
c2d1e0f1efed847dccb876543210fedcba9876543210fedcba9876543210fedcba9876d2c3b4a5e1d2a32edb0
98fa7f5e4d3c2b1a0f9e8d7c6b5a4f3e2d1c0b9a8f7e6d5c4b3a2f1e0d9c8b7

number_two =
6b5a4f3d7c6b5a97786a5b4c3d2e1f0e1d28586977c3b4a858697786a5b4c3d2e1f0e1d2c3b8e1f4a59687786
95a4b3c2d1e0f86a5b4c3d2e1f0e1d2c3b4a86a5b4c3d24f3977c3b4a0e1d2586

number_three =
f4a5b6c7d8e9f0a1b2c3d4e5f7af890bde23a2d852d1e0a4b3c7793f42d3c4b5a68a4b3c2d1e0f1e2d3c4b5a6
8f0e1d2c3b4a5968778695a4f1e8b3c2d1e0bafecabfefcabcabcabcacab7d8e9f0a1b2c3d4e5f7af890bde23
a2d1e5a4b3c2d6789abcdef0123456789abcdef01cabca01

number_four =
fed847da5e1d2a32edb098fa7f5e4d3c2b1a0f1e0f1e2d3c4b5a68a5b6c7d8e9f0aa5e1d2a32edb098fa7f5e4
d3c2b1a0f1e0f1e2d3c4b5a68a5b6c7d8e9f0a46bec3

Перевірка властивостей:

```
Check the properties of numbers:
First property:         (a+b)*c = c*(a+b) = a*c + b*c (mod n)
6d45dd951912cc00345097eab86744f0619c3de1a64fb100520a5125158b53d5749bbb9d7bf7e6d39db4efe9d949800b0e607e09cd9b6c4a7c071db9d19afeb7
ba7e

6d45dd951912cc00345097eab86744f0619c3de1a64fb100520a5125158b53d5749bbb9d7bf7e6d39db4efe9d949800b0e607e09cd9b6c4a7c071db9d19afeb7
ba7e

6d45dd951912cc00345097eab86744f0619c3de1a64fb100520a5125158b53d5749bbb9d7bf7e6d39db4efe9d949800b0e607e09cd9b6c4a7c071db9d19afeb7
ba7e

Second property:        a + ... + a = a*m (mod n)
6cc867c163b2baedea419569056b6523a431810c7456d484430b5f24a593e5b4617ee96f51fdcf7c2540efd8273b10f39d2095cda15c079f843c8966829f6423
191ba

6cc867c163b2baedea419569056b6523a431810c7456d484430b5f24a593e5b4617ee96f51fdcf7c2540efd8273b10f39d2095cda15c079f843c8966829f6423
191ba

Third property: a^phi(3^k) = a^(2*3^(k-1)) = 1 mod 3^k
GCD a and FOUR is:      1
Yes, property is correct
```

Результат НСД, НСК багаторозрядних чисел:

```
GCD is: 3
LCM is: 1b3b7bb3ebd114baf6b59dee004b95098772ee5e8fe9fb4cbb040580118a1d21cb7b88689b52340fa2a05d9784cf8cd382ef3d59e76d7ae2
a2d6590a8cbe54b16ba09e2c6e7f70f641c294a9feec4bc732c1755184ce48107e18a0f803e19da154a380a6b30d63d11f727bb3d77426ec7e555d40
4a2f0265de02040359346072b4a67c3c4da8697cf455ed39c4ba4e6f4da760468cf8e980ee
```

Результат додавання, віднімання, множення, піднесення до степеню за модулем багаторозрядного числа:

```
SUM is: 759eef1c6eada97b35a57c7c4b23266ddf29c0549fa23fd374a05aa4acece23ed4ddb41bb98e57d6f1b5660bafe87ad39962cbfdf7f57b97
76ca12e4347c0767adef8
Difference is:  653e087b79f45d666a2998367414bcf195d0a717d0c36911134655f916100dd045737e0c0fcdde8b1b5b7ab9a0a4476e51c9a0a0
806e6283a4f2b9c02629d6af6216d
Multiply is:    489ef9d689bf01c91e5cbc1be3dd1706576f683489a57cd2cd9148639706613f1abaffac1f94cc38d0bccc6c62a2d1f991c9ab30
a1293e4870fd1c36b46b350d4b466
Power is:       5a6a7afb0427c46f9f05b4ac1447691240bbdef11a55a1f74666a4fbbf87ee0e87fe71a08e713114e0021e78caecbbb06d917f1d
e9a3b116b64e12b4515806480b9f1
```

Усі операції працюють коректно, перевірено за допомогою https://srom-check.herokuapp.com

Також було виміряно середній час виконання кожної зазначеної операції, на 10 запусках, та відповідна кількість тактів процесора, результати були згруповані у таблицю:

Операція	Середній час виконання (секунди)	Кількість тактів процесора
НСД	0.00109537	1423981
НСК	0.00789117	10258521
Додавання	0.00280677	3648801
Віднімання	0.00288052	3744676
Множення	0.02768571	35991423
Піднесення в квадрат	0.00103588	1346644
Піднесення до степеня великого числа	2.77519	3607747000

Кількість тактів процесора обрахована за формулою:
кількість тактів = час виконання (секунди) * тактова частота (Гц)
Тактова частота комп'ютера: 1,30ГГц, тобто 1300000000Гц

Додатки:

Увесь код можна знайти за посиланням, на GitHub:
https://github.com/MansteinOrGuderian/SpRzOM-1