# СПЕЦІАЛЬНІ РОЗДІЛИ ОБЧИСЛЮВАЛЬНОЇ МАТЕМАТИКИ
## КОМП'ЮТЕРНИЙ ПРАКТИКУМ №1

## Багаторозрядна арифметика

### 1. Мета роботи

Отримання практичних навичок програмної реалізації багаторозрядної арифметики; ознайомлення з прийомами ефективної реалізації критичних по часу ділянок програмного коду та методами 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 $\beta$:

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

де $n = \lfloor \log_{\beta} N \rfloor$ – 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 $\beta = 2^w$:

$$A = a_{n-1}\beta^{n-1} + \ldots + a_1\beta + a_0,$$
$$B = b_{n-1}\beta^{n-1} + \ldots + b_1\beta + b_0,$$

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 $c_n$); 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
```

Навіть такий простий алгоритм містить як «підводні камені», так і шляхи для покращення. По-перше, змінна temp повинна містити $w+1$ значущий біт (чому?), отже, якщо *w* дорівнює розміру регістру процесора, значення temp буде обчислюватись некоректно.

По-друге, здається, що алгоритм додавання містить багато важких операцій ділення та остачі від ділення, тобто він має бути повільним. Однак насправді це не так: ми будемо широко використовувати той факт, що ми працюємо у двійковій архітектурі, в якій ділення на степінь двійки є лише бітовим зсувом вправо (операція `lsr`, або `>>`), а остача від ділення на степінь двійки – це останні *w* біт числа, які можна знайти за допомогою операції логічного ТА (`and`, або `&`). Остаточно процедура додавання матиме такий вигляд:

**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 & (2^w - 1);          // чому саме так?
      carry := temp >> 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:             // всі цифри однакові
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:

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

Бачимо, що для обчислення добутку необхідно навчитись множити багаторозрядні числа на одну цифру та на степінь $\beta$. Однак в нашій моделі чисел множення на степені $\beta$ 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 & (2^w - 1);
      carry := temp >> w;          // скільки значущих біт містить carry?
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;          // багаторозрядне додавання!
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.

Аналіз показує, що в ході роботи процедура LongMul виконує $n^2$ множень окремих цифр (а також ще деяку кількість додавань та зсувів). Довгий час вважалось, що більш швидкого алгоритму множення не існує: дійсно, нам потрібно перемножити кожну цифру числа $A$ на кожну цифру числа $B$, що й дає загальну кількість операцій $n^2$. І лише в середині XX століття радянські математики Карацуба та Офман показали, що існують асимптотично більш швидкі алгоритми множення.

Розглянемо $2n$-розрядні числа $A$ та $B$, які розділимо на дві половини по $n$ розрядів кожна:

$$A = A_1\beta^n + A_0,\ B = B_1\beta^n + B_0.$$

Тоді їх добуток можна записати як

$$A \cdot B = (A_1\beta^n + A_0)(B_1\beta^n + B_0) = A_1B_1\beta^{2n} + (A_0B_1 + A_1B_0)\beta^n + A_0B_0.$$

Спочатку ми мали $(2n)^2$ операцій множення, після перетворень – чотири добутки по $n^2$ операцій кожен; здається, що ніякого виграшу не має. Однак розглянемо такий вираз:

$$(A_1 + A_0)(B_1 + B_0) = A_1B_1 + A_0B_0 + A_1B_0 + A_0B_1,$$
$$A_0B_0 + A_1B_0 = (A_1 + A_0)(B_1 + B_0) - A_1B_1 - A_0B_1;$$

підставляючи одержаний вираз у формулу добутку, маємо:

$$A \cdot B = A_1B_1\beta^{2n} + ((A_1 + A_0)(B_1 + B_0) - A_1B_1 - A_0B_1)\beta^n + A_0B_0,$$

і, таким чином, замість чотирьох множень *n*-розрядних чисел ми одержали лише три! Більш того, для кожного такого множення можна застосувати той самий принцип, зменшуючи кількість операцій на кожному рівні рекурсії, аж поки ми не дістанемо добуток одноцифрових чисел.

Доведено, що алгоритм множення Карацуби має асимптотичну складність $O(n^{\log_2 3}) \approx O(n^{1,58})$. На практиці ефективність алгоритму Карацуби починає проявлятись для чисел із довжиною більше ніж 512 біт, що викликано додатковими накладними витратами на реалізацію. Тому часто множення чисел в промислових бібліотеках реалізовано ітеративним «подрібленням» чисел за алгоритмом Карацуби до деякої порогової довжини (наприклад, 256 біт), на якій числа перемножуються у стовпчик.

Робота Карацуби та Офмана надала значного поштовху цілому напрямку обчислювальної математики. Пізніше Тоом та Кук розробили ще більш асимптотично ефективний алгоритм множення, що використовує поділ числа на три частини, а також довели, що можна побудувати алгоритм множення із асимптотичною складністю $O(n^{1+\varepsilon})$ для довільного $\varepsilon > 0$. Втім, такі алгоритми стають ефективними на практиці для дуже великих довжин чисел. З іншого боку, Шенгаге та Штрассен розробили інший швидкий алгоритм множення, що базується на швидкому перетворенні Фур'є та має асимптотичну складність $O(n \log n \log\log n)$. Цей алгоритм є доволі складним, однак його використання оправдане у випадках, коли саме швидкість множення критична для обчислювальної системи.

### 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.

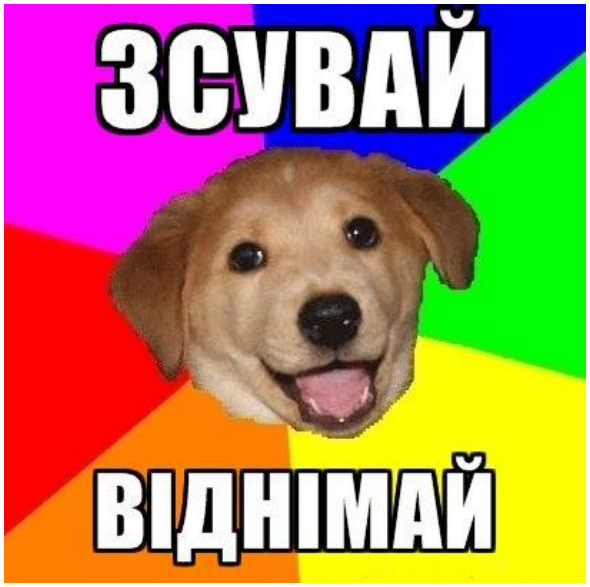


**Рисунок 2.1** – Advice Dog радить, яким чином реалізувати ділення у вашому практикумі.

**Процедура LongDivMod (A, B, Q, R)**
Вхід: багаторозрядні числа A, B.
Вихід: багаторозрядні частка Q та остача від ділення R: $A = B \cdot Q + R,\ 0 \le R < B$.

```
k := BitLength(B);
R := A;
Q := 0;
while R >= B do:              // багаторозрядне порівняння!
t := BitLength(R);
C := LongShiftBitsToHigh(B, t - k);
if R < C then:           // багаторозрядне порівняння! вийшло забагато?
      t := t - 1;
      C := LongShiftBitsToHigh(B, t - k);
R := R - C;
Q := Q + 2^(t - k);         // встановити в Q біт із номером (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 $B = b_{m-1}2^{m-1} + ... + b_1 2 + b_0$, де $b_i \in \{0, 1\}$ – окремі біти числа $B$. Тоді можемо записати:

$$A^B = A^{b_{m-1}2^{m-1}+...+b_1 2+b_0} = \prod_{i=0}^{m-1} \left(A^{2^i}\right)^{b_i}.$$

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

Вихід: багаторозрядне число $C = A^B$.

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

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

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

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

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

Вихід: багаторозрядне число $C = A^B$.

```
C := 1;
for i := m-1 to 0 do:
if b[i] = 1 then:
      C := C * A;       // багаторозрядне множення!
if i <> 0 then:         // на останньому кроці до квадрату не підносимо
            C := C * C;       // багаторозрядне множення!
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

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

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 $\beta' = 2^t$: $B = b_{m-1}2^{t(m-1)} + ... + b_1 2^t + b_0$.

Вихід: багаторозрядне число $C = A^B$.

```
C := 1;
D[0] := 1; D[1] := A;          // D[] – таблиця степенів A
for i := 2 to (2^t – 1) do:    // передобчислення
D[i] := D[i – 1] * A;

for i := m-1 to 0 do:
C := C * D[b[i]];       // багаторозрядне множення!
if i <> 0 then:         // на останньому кроці до квадратів не підносимо
            for k := 1 to t do:     // підносимо до квадрату t разів
                  C := C * C;       // багаторозрядне множення!
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.

## 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.

**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.

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. $n\cdot a = \underbrace{a+a+...+a}_{n}$, 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.

**Продемонструвати працюючу програму викладачеві!**
**(бажано компілювати на місці, щоб була можливість змінювати програму)**

## 4. Технічні зауваження

Наступні зауваження та рекомендації можуть виявитись корисними під час виконання комп'ютерного практикуму.

– Числа рекомендується реалізовувати беззнаковими;

– довжина числа повинна легко параметризуватися, тобто програма повинна легко модифікуватися для роботи з числами іншої довжини;

– незалежно від внутрішньої реалізації числа (наприклад, масив 32-х бітних слів фіксованої чи змінної довжини, або масив байтів, що містять двійкові цифри) зовнішній користувач повинен бачити *m*-бітне число у природному (символьному) вигляді; обов'язково реалізувати представлення чисел у шістнадцятковому вигляді, за бажанням – у десятковому або іншій системі числення.

– рекомендована (але не обов'язкова) внутрішня реалізація – масив 32-х бітних слів (64-бітних для відповідних архітектур) фіксованої довжини; це досить просто та ефективно по швидкодії та пам'яті (взагалі, чим більший розмір машинного слова, тим краще);

– реалізація чисел зі змінною внутрішньою довжиною не рекомендується (потрібно обґрунтувати, що це дасть суттєвий виграш у швидкодії);

– реалізації типу «одна комірка масиву = один біт числа» є дуже неефективними по швидкодії, однак дуже простими для розробки;

– якщо при виконанні немодулярного додавання (множення) результат перевищує допустиму довжину числа, можна обрубати результат до потрібної довжини; також можна реалізувати підтримку чисел подвійної довжини або якийсь інший варіант – вибір за вами;

– час можна вимірювати як за допомогою власних тестів із заміром часу, так і за допомогою profiler'a – спеціального інструменту, який призначений, зокрема, і для визначення часової ефективності програми; профайлери вбудовані в більшість середовищ розробки;

– для підвищення точності вимірювання треба викликати піддослідну функцію у циклі декілька (сто, тисяча, мільйон) разів на випадкових даних; якщо викликати функцію постійно на одних й тих самих даних, то єдине, що ви обчислите – це час зчитування з кешу процесора (а це – доволі мала величина);

– оптимізувати слід лише ті функції, які забирають суттєвий відсоток часу;

– критичні за часом ділянки можна реалізувати на асемблері (за допомогою асемблерних вставок або як окремі асемблерні функції), якщо середовище розробки дозволяє; також слід проконтролювати, щоб в опціях компілятора була встановлена оптимізація по швидкості (без неї

час роботи може збільшитися в декілька разів, це особливо важливо, коли не використовується асемблер).

Використання вбудованих типів даних (наприклад, BigInteger у Java) та готових бібліотек, що реалізують багаторозрядну арифметику, дозволяється лише для таких цілей:

1) перевірити коректність роботи свого класу;
2) щоб показати, в скільки разів Ваша реалізація швидша за стандартну.

PHP, Javascript та інші інтерпретаторні мови п'ятого покоління, функціональні мови програмування, мова Brainfuck тощо не допускаються через вкрай низьку швидкодію.

## 5. Оформлення звіту

Звіт оформлюється відповідно до стандартних правил оформлення наукових робіт. Звіт має містити:

1) мету, теоретичну частину та завдання, наведені вище, згідно варіанту;
2) бітові/символьні зображення тестових чисел та результатів операцій над ними;
3) бітові/символьні зображення результатів контролю за пунктом Б);
4) текст програми (у додатку)

## 6. Оцінювання комп'ютерного практикуму

За виконання лабораторної роботи студент може одержати до 8 рейтингових балів; зокрема, оцінюються такі позиції:

- реалізація основного завдання – до 6-ти балів;
- оформлення звіту – 1 бали;
- своєчасне виконання основного завдання – 1 бал;
- несвоєчасне виконання роботи – (-1) бал за кожні два тижні пропуску.

## 7. Рекомендована література


1. *Д. Кнут*. Искусство программирования на ЭВМ. Т.2. – М.:Мир, 1976. –724 с.
2. *О.Н. Василенко.* Теоретико-числове алгоритмы в криптографии
3. *А.А. Болотов.* Алгоритмические основы современной криптологии
4. *А.В. Анісімов.* Алгоритмічна теорія великих чисел. Модулярна арифметика великих чисел. – К.: Академкнига, 2001.
5. *Çetin Kaya Koç*. High-Speed RSA Implementation.