1. Задачі цілочислової арифметики

Чи доводилось Вам виконувати такі завдання: обчислити середнє арифметичне цифр заданого натурального числа; перевірити, чи задане натуральне число є паліндромом[1]; розкласти задане натуральне число на прості множники? На перший погляд може видатися, що програми для розв'язування цих задач дуже складні, потребують використання масивів чи файлів. Насправді ж для їхнього створення достатньо вміло використати операції цілочислової арифметики. Давайте подивимось, як це зробити.

Надалі під час розв'язування задач ми намагатимемося виконувати такі дії: формулювати допоміжні запитання до умови задачі, щоб краще її зрозуміти, і давати на них відповіді; проектувати прості змінні та структури даних для потреб майбутнього алгоритму, давати його словесний опис.

1.1. Середнє арифметичне цифр числа

Задача 1. *Задано натуральне число. Обчислити і надрукувати середнє арифметичне цифр у записі цього числа.*

Розв'язування. Відомо, що для обчислення середнього арифметичного деяких величин потрібно знати їхню суму та кількість. Як обчислити суму цифр, з яких складається запис числа n? Для цього доведеться якось отримати кожну з них. Найпростіше отримати цифру з наймолодшого розряду – розряду одиниць. Вона дорівнює остачі від ділення n на 10. Наприклад, для n = 1976 легко отримати 1976 % 10 = 6. Як дізнатися кількість десятків у числі n? Спробуємо так: (1976 % 100 – 6) / 10 = 7. Однак, поміркувавши трохи, вдається досягти такого ж результату і простішим способом: 1976 / 10 % 10 = 7. Тут операція ділення вилучає уже враховану цифру з розряду одиниць, а операція взяття остачі повертає наступну – цифру з розряду десятків. Очевидно, що послідовно вилучаючи з запису числа молодші цифри, можна перебрати їх усі.

Як знайти k кількість цифр у записі числа n в десятковій системі числення? Наприклад, за формулою $k = [\lg n] + 1$. Або перелічити їх по одній, адже ми вже придумали, як отримати кожну з них.

Зауважимо також, що вилучення молодших цифр змінюватиме значення заданого числа, а його варто було б зберегти недоторканим, наприклад, щоб надрукувати поруч з обчисленим результатом. Отож використаємо в обчисленнях копію заданого числа.

Які змінні потрібні для побудови алгоритму? Очевидно, такі: *given_number* – задане число, *rightmost_digit* – наймолодша цифра, *rest_of_digits* – ще не опрацьована частина числа (спочатку – копія заданого числа), *sum* – сума цифр, *digits_quantity* – лічильник цифр, *average* – шукане середнє арифметичне. Компіляторові байдуже, як ми називаємо змінні, проте для читабельності програм варто використовувати змістовні імена, які одразу повідомляють читачеві своє призначення.

Тепер запишемо алгоритм у вигляді функції мовою C++. Для того, щоб вона «ожила» і виконалася, достатньо буде викликати її в будь-якому місці головної програми.

```
void DigitsAverage()
{
	cout << "\n Введіть натуральне число: ";
	unsigned given_number; cin >> given_number;
	// неопрацьована частина числа - спочатку ціле число
	unsigned rest_of_digits = given_number;
```

[1] *Паліндром* – число, величина якого не зміниться, якщо порядок цифр у його записі змінити на обернений.

```
        // початкові значення суми та лічильника
        unsigned sum = 0, digits_quantity = 0;

        while (rest_of_digits > 0)     // перебираємо цифри введеного числа
        {
              // отримали наймолодшу цифру
              unsigned rightmost_digit = rest_of_digits % 10;
              sum += rightmost_digit;           // врахували її
              ++digits_quantity;
              rest_of_digits /= 10;             // вилучили її
        }
        // перетворили ціле на дійсне і обчислили середнє арифметичне
        double average = sum; average /= digits_quantity;
        cout <<"задане число = "<< given_number <<"   середнє = "<< average <<'\n';
}
```

Виконаємо ручну прокрутку цієї функції, щоб ліпше зрозуміти, як працює алгоритм. Для цього заповнимо таблицю прокрутки. З її допомогою покажемо, як змінюються значення змінних і умова продовження циклу під час виконання програми для деякого конкретного значення *given_number*. Нехай задано число 1976, тоді

rightmost_digit	*sum*	*digits_quantity*	*rest_of_digits*	*rest_of_digits* >0?	*average*
	0	0	1976	⇒ так	
6	6	1	197	так	
7	13	2	19	так	
9	22	3	1	так	
1	23	4	0	ні ⇐	5.75

Тут символами "⇒" і "⇐" позначають, відповідно, початок і кінець циклу. З таблиці видно, як з *rest_of_digits* послідовно вилучаються цифри, які стають значеннями змінної *rightmost_digit*.

Задачу розв'язано.

1.2. Чи є число паліндромом?

Задача 2. *Задано натуральне число. Перевірити, чи воно є паліндромом.*

Розв'язування. Запис паліндрома симетричний: перша цифра дорівнює останній, друга – передостанній і т. д. Проте перевірити, чи так воно є у заданого числа, не так уже й просто: отримати першу цифру набагато складніше, ніж останню. А чи не можна пристосувати якось алгоритм з попередньої задачі для того, щоб отримати обернений запис числа? Це легко зробити, якщо відповідно враховувати послідовні значення змінної *rightmost_digit* і будувати з них нове число. Адже кожне число можна записати у вигляді полінома за степенями 10, коефіцієнтами якого є цифри числа.

Розглянемо другий рядок таблиці прокрутки попередньої задачі. Бачимо, що цифра 6 уже перемістилась зі змінної *rest_of_digits* до змінної *sum*. Як зробити, щоб на наступному кроці циклу змінна *sum* отримала значення 67, а не 13? Наприклад, за допомогою інструкції *sum*=*sum*×10+*rightmost_digit* (замість *sum*=*sum*+*rightmost_digit*). Легко переконатися, що виконання цієї інструкції на кожному кроці циклу дає змогу отримати в *sum* обернений запис заданого числа. Для отримання розв'язку задачі потрібно порівняти побудоване нами число з заданим, тому, як і в попередній програмі, використаємо в обчисленнях копію заданого числа.

Тепер легко записати весь алгоритм, використавши замість *sum* змінну *reversed_number* – нове число:

```
void IsPalindrome()
{
	cout << " Введіть натуральне число: ";
	unsigned given_number; cin >> given_number;
	// початкове значення нового числа та копія заданого
	unsigned reversed_number = 0;
	unsigned rest_of_digits = given_number;
	while (rest_of_digits > 0) // перебираємо цифри введеного числа
	{
		// отримали наймолодшу цифру
		unsigned rightmost_digit = rest_of_digits % 10;
		reversed_number = reversed_number*10 + rightmost_digit; // врахували
		rest_of_digits /= 10;                                   // і вилучили її
	}
	if (reversed_number == given_number)
		cout << given_number << " - паліндром\n";
	else cout << given_number << " - не паліндром\n";
	return;
}
```

Отже, ми успішно пристосували попередню функцію до нової задачі. Голлівудські продюсери давно зрозуміли, що хороший сценарій можна використати і для зйомок фільму, і для серіалу, і для публікації роману чи хоча б збірника коміксів. Це розуміння вони перейняли від програмістів: адже один з методів розробки алгоритмів саме полягає у повторному використанні старого перевіреного алгоритму (чи його частини) у нових умовах.

1.3. Гіпотеза Безу

Розглянемо деяке натуральне число *n*. Якщо воно не паліндром, то побудуємо нове число, змінивши порядок цифр у записі *n* на обернений, і додамо його до *n*. Якщо отримана сума не паліндром, то повторимо з нею описані дії, доки не отримаємо паліндром. Гіпотеза Безу стверджує, що описаний процес скінченний для будь-якого натурального *n*. Цю гіпотезу досі не доведено.

Задача 3. *Задано натуральні числа a, b, k (a ≤ b). Перевірити, чи виконується гіпотеза Безу для кожного натурального числа з проміжку [a, b] не більше, як за k кроків процесу.*

Розв'язування. Щоб побудувати алгоритм розв'язування цієї задачі, спробуємо ліпше зрозуміти її умову та сформулювати головні етапи отримання розв'язку. Отже, у задачі йдеться про перевірку гіпотези Безу для послідовності чисел *a*, *a*+1, ..., *b*. Якщо б ми вміли виконувати таку перевірку для одного числа *n*, то перевірити усю послідовність теж змогли б, послідовно надаючи змінній *n* значення *a*, *a*+1, ..., *b* (наприклад, за допомогою інструкції циклу **for**). Як виконати перевірку для *n*? Очевидно, що описаний процес перетворення є циклічним. Цикл треба виконувати до отримання паліндрома або до вичерпання заданої кількості повторень. Розв'язуючи попередню задачу, ми описали алгоритм отримання оберненого запису заданого числа. Оформимо тепер цей алгоритм у вигляді окремої функції, яку використаємо для перевірки на кожному кроці циклу.

Тепер уже можна записати алгоритм. Призначення змінних пояснено у коментарях:

```
// функція побудови зворотного запису числа
long long Reverse(long long m)
{     // параметр-значення m містить копію заданого числа
      long long s = 0;  // "перевернуте" число
      while (m > 0)
      {
            s = s * 10 + m % 10; // враховуємо молодшу цифру
            m /= 10;             // відкидаємо її
      }
      return s;
}

void BezuHypothesis()
{
      cout << "\n *Перевірка гіпотези Безу для x є [a;b]*\n\n";
      unsigned a, b, k;
      cout << "Введіть дані a, b, k: ";
      cin >> a >> b >> k;
      // параметр циклу перебиратиме числа заданого діапазону
      for (unsigned number = a; number <= b; ++number)
      {
            unsigned i = 0;            // лічильник внутрішнього циклу
            long long direct = number; // на першому кроці циклу - копія number,
                                // на наступних - сума прямого і оберненого запису
            long long reversed = 0;    // обернений запис числа direct
            do
            {
                  direct += reversed;
                  reversed = Reverse(direct);
                  ++i;
            }
            while (reversed != direct && i <= k);
            // результати перевірки
            std::cout << "Для числа " << number << " за " << i << " крок(и) ";
            if (reversed != direct) cout << "гіпотеза не виконується\n";
            else cout << "отримано паліндром " << reversed << '\n';
      }
      return;
}
```

У цій функції початкові значення для змінних *direct* і *reversed* підібрано так, щоб можна було перевірити, чи є паліндромом *number*, на першому кроці циклу. Результати перевірки гіпотези друкують для кожного числа з проміжку [*a*, *b*].

Розв'язок цієї задачі демонструє, як у складній програмі можна використати алгоритм – розв'язок простішої задачі. Таблицю прокрутки цієї програми пропонуємо читачеві побудувати самостійно.

1.4. Запис числа у шістнадцятковій системі

Безперечно, більшість із нас розпочинала вивчати програмування з опанування двійкової, вісімкової та шістнадцяткової систем числення. Пригадуєте, чи не найгромізд-кішими були обчислення під час переведення чисел з однієї системи в іншу. От би мати таку програму, яка б швидко і безпомилково виконувала такі переведення! Звичайно, кожен компілятор вміє робити переведення з десяткової системи у двійкову, і навпаки. Проте

виконує він це для внутрішніх потреб програми і майже ніколи не демонструє користувачеві результати переведення. (А про калькулятор у смартфоні ми тут згадувати не будемо. ☺)

Задача 4. *Задано натуральне число. Отримати його запис у двійковій та шістнадцятковій системах числення.*

Розв'язування. Пригадаємо алгоритм переведення цілого числа у нову систему: потрібно обчислити остачу від ділення цього числа на нову основу, отриману частку знову поділити на нову основу й обчислити остачу і так далі, аж доки чергова частка дорівнюватиме нулю. Знайдені остачі, записані в оберненому до черговості отримання порядку, утворюють запис числа в новій системі.

Розв'яжемо спочатку простішу від сформульованої задачу: переведемо задане число у двійкову систему. Щоб послідовно обчислити згадані частки й остачі, нам стане в пригоді досвід розв'язування попередніх задач (використаємо в циклі оператори % і /). Як побудувати двійковий запис числа з обчислених остач? Щоб відповісти на це запитання, доведеться спочатку вирішити, змінну якого типу ми використаємо для зберігання цього запису. Наприклад, його можна змоделювати змінною цілого типу. Двійковий запис числа буває досить довгим, тому доцільно використати тип **unsigned long long int**, що має найширший серед цілих типів діапазон можливих значень. Щоб чергова остача під час побудови двійкового запису займала відповідний розряд, будемо домножувати її на відповідний степінь десяти. Цей степінь можна зберігати у додатковій робочій змінній. Отже, використаємо такі змінні: *given_number* – задане число; *power_of_10* – степінь десяти; *binary_number* – число, що моделює двійковий запис числа *n*.

```
void BinaryForm()
{
	cout << "\n *Переведення числа у двійкову систему*\n\n";
	unsigned given_number;
	cout << "Введіть натуральне число: "; cin >> given_number;
	unsigned long long binary_number = 0ULL; // двійковий запис спочатку порожній
	unsigned long long power_of_10 = 1ULL;   // power_of_10 = 10^0
	while (given_number > 0)
	{	// остачу помістили в двійковий запис
		binary_number += (given_number % 2) * power_of_10;
		given_number /= 2;		// частку треба ще перевести
		power_of_10 *= 10;		// підготували степінь 10 для наступної цифри
	}
	cout << "Запис у двійковій системі: " << binary_number << '\n';
	return;
}
```

Перевіримо за допомогою ручної прокрутки, чи буде ця функція працювати правильно, наприклад, для *given_number* = 10:

given_number%2	*given_number*	*binary_number*	*power_of_10*	*given_number*>0?	
	10	0	1	⇒ так	
0	5	0	10	так	
1	2	10	100	так	
0	1	10	1000	так	
1	0	1010	10000	ні	⇐

Бачимо, що величини *binary_number* і *power_of_10* зростають дуже швидко. Їхнього розміру вистачить для утворення двійкового запису довжиною до 19 цифр, тобто для $n \in [0; 524287]$. Щоб мати змогу переводити у двійкову систему і більші числа, використаємо

для зображення їхнього запису рядок. Перед початком циклу він порожній. У циклі на початку цього рядка треба дописувати ’0’ або ’1’, залежно від парності частки (за такого підходу нам навіть не потрібно буде обчислювати остачу).

Порівняйте таку функцію з попередньою:

```
void BinaryFormStr()
{
	cout << "\n *Побудова запису числа у двійковій системі*\n\n";
	unsigned given_number;
	cout << "Введіть натуральне число: "; cin >> given_number;
	// готуємо місце для двійкового запису
	int k = log2(double(given_number)) + 1; // кількість двійкових цифр
	char * str = new char[k + 1];
	str[k] = '\0';		// кожен рядок закінчується літерою з кодом 0
	while (given_number > 0)
	{
		--k;
		if (given_number % 2 == 1)
			str[k] = '1'; // остання двійкова цифра непарного числа - 1
		else str[k] = '0';  // наприкінці парного - двійковий 0
		given_number /= 2;
	}
	cout << "Запис у двійковій системі: " << str << '\n';
	delete[] str;
	return;
}
```

Програма *BinaryFormStr* виглядає привабливіше за *BinaryForm*: вона потребує менше змінних і обчислень, працює для ширшого діапазону вхідних даних. Пристосуємо її щодо отримання шістнадцяткового запису числа. З цією метою нам треба вирішити, як перетворювати остачі c від ділення на 16 у літери – шістнадцяткові цифри. Для значень $c = 0, 1, ..., 9$ таке перетворення можна виконати за допомогою інструкції *aChar*='0'+*c*. А для значень $c = 10, ..., 15$ – за допомогою інструкції *aChar*='A'–10+*c*. Тут використано ту особливість C++, що літерний тип – один з цілих, і такі властивості: '0'–'0'==0, '5'–'0'==5, 'B' – 'A'==1.

```
void HexaFormStr()
{
	cout << "\n *Побудова запису числа у шістнадцятковій системі*\n\n";
	unsigned given_number;
	cout << "Введіть натуральне число: "; cin >> given_number;
	unsigned number = given_number;   // копія заданого числа для перетворень
	// готуємо місце для шістнадцяткового запису
	int k = log2(double(number)) / 4.0 + 1; // кількість шістнадцяткових цифр
	char * str = new char[k + 1];
	str[k] = '\0';			// кожен рядок закінчується літерою з кодом 0
	while (n > 0)
	{
		--k;
		unsigned digit = number % 16;  // остання шістнадцяткова цифра числа
		if (digit < 10) str[k] = '0' + digit;	// звичайні цифри
		else str[k] = 'A' - 10 + digit;		// старші цифри-букви
		number /= 16;
	}
	cout << "Запис " << given_number
		<< " у шістнадцятковій системі: " << str << '\n';
```

```
    delete[] str;
    cout.setf(std::ios_base::hex, std::ios_base::basefield);
    cout << "Те саме засобами стандартної бібліотеки: " << given_number << '\n';
    return;
}
```

У програмуванні часто буває так, що доводиться спочатку формулювати і розв'язувати деякі допоміжні задачі, вибирати кращий варіант з кількох можливих, далі удосконалювати його і доводити до завершення. Такий процес побудови алгоритму ми і хотіли проілюструвати цим прикладом.

Два рядки інструкцій наприкінці програми вставлено для перевірки правильності побудованого запису. Перший з них налаштовує потік виведення на режим відображення цілих у шістнадцятковій системі. Ми могли одразу скористатися саме ним, але тоді так би і не дізналися, «як воно працює». Розроблений нами алгоритм можна пристосувати до довільної основи числення, а можливості *cout* обмежено лише трьома: звичайна десяткова, шістнадцяткова та вісімкова (прапорець *std::ios_base::oct*).

1.5. Розкладання числа на прості множники

На завершення цього параграфа опишемо розв'язування ще однієї цікавої задачі.

Задача 5. *Задано натуральне число n (n>1). Розкласти його на прості множники.*

Розкладом є послідовність простих чисел – дільників n – з урахуванням їхньої кратності. Наприклад, 84 = 2×2×3×7. Як отримати такий розклад? Перше, що спадає на думку, – це перебрати всі числа k з проміжку від 2 до n і для кожного з них визначити, чи воно просте, чи є воно дільником n. Якщо так, то надрукувати знайдене значення k потрібну кількість разів. Як перевірити, чи ділиться n на k? Скільки разів? Просто перевірити остачу від ділення n на k, k^2, k^3, ... Наприклад, використавши змінну m для зберігання степеня k це можна зробити так:

```
unsigned m = k;
while (n % m == 0)
{     cout << k << " x ";
      m *= k;
}
```

Як перевірити, чи число k є простим? Треба визначити, чи має k інші дільники, крім 1 і k. З цією метою перебираємо усі числа 2, 3, ..., k–1, перевіряючи, чи k ділиться на ці числа. Проте неважко здогадатися, що кожному дільникові j числа k, більшому за $\sqrt{k}$, відповідає дільник k / j, менший за $\sqrt{k}$, тому достатньо буде перебрати числа 2, 3, ..., $[\sqrt{k}]$ (для великих чисел k це суттєво скорочує перебір). Нехай змінна *k_prime* набуває значення «істина», якщо k – просте, і значення «хиба» у протилежному випадку. Перевірку того, чи k – просте, можна записати так:

```
bool k_prime = true;
unsigned high = sqrt(double(k)) + 0.5;
for (unsigned j = 2; j <= high; ++j)
{     if (k % j == 0)
      {     k_prime = false; break;
      }
}
```

Відомо, що функція обчислення кореня визначена для всіх дійсних типів і ні для одного цілого. Компілятор не зможе автоматично вирішити, яку з них застосовувати до цілого *k*, тому застосуємо примусове перетворення типу до *double*. Отриманий результат матиме дійсний тип і може бути обчислений з недостачею. Тому у цьому фрагменті ми додали до нього 0.5, щоб взяти значення з надлишком (змоделювали заокруглення до цілого). Тепер можна записати всю функцію розкладання числа. Як і раніше, використаємо змістовні імена змінних: *given_number* – число, яке потрібно розкласти; *divider* – можливий дільник заданого числа (у попередніх міркуваннях ми називали його *k*); *div_times* – степінь дільника (аналог *m*); *is_prime* – ознака того, чи *divider* є простим числом.

```
void Decomp_1()
{
	cout << "\n *Розклад числа на прості множники (1)*\n\n";
	unsigned given_number;
	cout << "Введіть натуральне число: ";
	cin >> given_number;
	// Знайдемо просте число, перевіримо, чи є воно дільником

	// починаємо друкувати розклад
	cout << given_number << " = 1";

	// переберемо можливі дільники
	for (unsigned divider = 2; divider <= given_number; ++divider)
	{
		// кожного з кандидатів перевіримо на простоту
		bool is_prime = true;
		unsigned high = sqrt(double(divider)) + 0.5;
		for (unsigned i = 2; i <= high; ++i)
		{
			if (divider % i == 0)
			{
				is_prime = false;
				break;
			}
		}
		if (is_prime)
		{
			unsigned div_times = divider;	// степені простого divider
			while (given_number % div_times == 0)
			{	// скільки разів divider ділить given_number,
				cout << " x " << divider;  // стільки разів його друкуємо
				div_times *= divider;
			}
		}
	}
	cout << '\n';  // друк розкладу завершено
	return;
}
```

У цій функції кожне з чисел 2, 3, ..., *n* необхідно перевірити, чи воно просте. Зауважимо, що така перевірка потребує виконання циклу. Після уважного аналізу *Decomp_1* можна дійти висновку, що набагато вигідніше перевіряти на простоту тільки дільники заданого числа: обчислити остачу легше, ніж виконати згаданий цикл. Після нескладної переробки отримаємо:

```
void Decomp_2()
{
	cout << "\n *Розклад числа на прості множники (2)*\n\n";
	unsigned given_number;
	cout << "Введіть натуральне число: ";
	cin >> given_number;
	// Знайдемо дільник, перевіримо, чи він просте число

	// починаємо друкувати розклад
	cout << given_number << " = 1";

	// переберемо можливі дільники
	for (unsigned divider = 2; divider <= given_number; ++divider)
	{
		if (given_number % divider == 0)
		{
			// знайдений дільник перевіримо на простоту
			bool is_prime = true;
			unsigned high = sqrt(double(divider)) + 0.5;
			for (unsigned i = 2; i <= high; ++i)
			{
				if (divider % i == 0)
				{
					is_prime = false;
					break;
				}
			}
			if (is_prime)
			{
				unsigned div_times = divider;  // степені простого divider
				while (given_number % div_times == 0)
				{	// скільки разів divider ділить given_number,
					// стільки разів його друкуємо
					cout << " x " << divider;
					div_times *= divider;
				}
			}
		}
	}
	cout << '\n';  // друк розкладу завершено
	return;
}
```

Ця функція відрізняється від попередньої лише одним умовним оператором, однак працює набагато швидше. Проаналізуймо її. Вона розпочинає роботу з того, що знаходить серед чисел 2, 3, ..., *n* перше, яке ділить *n*. Далі відбувається перевірка, чи це число є простим, хоча ця перевірка зайва: якщо *n* не ділиться на 2, 3, ..., *k*–1 і ділиться на *k*, то *k* – просте число. Справді, якщо б *k* було складним, то воно ділилося б на котресь з чисел 2, 3, ..., *k*–1, але тоді б і *n* ділилось на це число, що суперечить способові побудови *k*.

Наведені міркування засвідчують, що і *Decomp_2* не є досконалою. Очевидно, можна суттєво зменшити кількість дорогих перевірок чисел на простоту. Адже перший вибраний з чисел 2, 3, ..., *n* дільник *k* буде простим. Якщо тепер надрукувати його і поділити на нього *n*, то часткою буде число, набір простих дільників якого збігається з набором ще неврахованих дільників *n*. Тобто описані дії можна повторити для частки *n* / *k*. Процес завершиться, коли чергова частка дорівнюватиме одиниці.

Враховуючи все сказане, складемо функцію, яка не міститиме явних перевірок на простоту і не виконуватиме перебору зайвих значень *k*:

```
void Decomposition()
{
	cout << "\n *Розклад числа на прості множники (3)*\n\n";
	unsigned given_number;
	cout << "Введіть натуральне число: ";
	cin >> given_number;
	// Вилучатимемо дільники з числа, що гарантуватиме їхню простоту

	// починаємо друкувати розклад
	cout << given_number << " = 1";

	unsigned divider = 2; // перше просте число
	while (given_number > 1)
	{
		if (given_number % divider == 0)
		{
			// надрукуємо черговий дільник і вилучимо його з given_number
			cout << " x " << divider;
			given_number /= divider;
		}
		else ++divider; // перевіримо наступного кандидата
	}

	cout << '\n';  // друк розкладу завершено
	return;
}
```

Ми розглянули кілька варіантів алгоритму розв'язування однієї задачі. Процес їх побудови демонструє, як, відштовхуючись від програми, яка спочатку видалася досить доброю, ми, використовуючи очевидні міркування, отримали програму, що мало схожа на початкову і, що важливо, набагато ліпша за неї.

Проте немає межі досконалості, й алгоритм Decomposition теж можна поліпшити. Поміркуймо, як він працюватиме, якщо заданим числом є досить велике просте, наприклад, 9973 (або останній множник у розкладі – велике просте число). Алгоритм сумлінно перебиратиме значення *divider* від 2 до *given_number*. Але, як вже було сказано, достатньо перевірити на подільність числа $k = 2, 3, ..., \sqrt{n}$, щоб переконатися, що *n* – просте. Замінимо умову циклу (*given_number* > 1) на (*divider***divider* <= *given_number*). Так ми завершимо цикл ще до того, як нерозкладена частина числа дорівнюватиме 1, переконавшися, що *given_number* містить просте число, яке і буде останнім множником розкладу. Остаточне виправлення алгоритму пропонуємо зробити читачам.

1.6. Головна програма

На завершення параграфа продемонструємо, як може виглядати головна програма. Вона організовує просте текстове меню, щоб користувач міг легко викликати описані раніше функції.

```
#include <Windows.h>
#include "int-procedures.h"
```

```
int main()
{     // --- Задачі цілочислової арифметики
      SetConsoleOutputCP(1251); // налаштуємо виведення кирилицею
      int answer;
      do
      {
            system("cls");      // очистити екран
                                // і надрукувати меню
            cout << "Виберіть програму для запуску:\n\n"
                 << " 1 - Середнє арифметичне цифр числа\n"
                 << " 2 - Чи є число паліндромом?\n"
                 << " 3 - Перевірка гіпотези Безу\n"
                 << " 4 - Переведення до (2) і (16) систем числення\n"
                 << " 5 - Розклад числа на прості множники\n"
                 << " 6 - Завершення роботи\n >>>> ";
            cin >> answer;      // вибір користувача
            system("cls");
            switch (answer)     // виклик відповідної функції
            {
            case 1: DigitsAverage(); break;
            case 2: IsPalindrome(); break;
            case 3: BezuHypothesis(); break;
            case 4: BinaryForm(); BinaryFormStr(); HexaFormStr(); break;
            case 5: Decomp_1(); Decomp_2(); Decomposition(); break;
            default: cout << "До побачення!\n";
            }
            system("pause");    // затримка завершення програми
      }
      while (answer > 0 && answer < 6);
      return 0;
}
```

Заголовковий файл *Windows.h* приєднуємо, щоб налаштувати виведення кирилиці, а файл *intProcedures.h* містить прототипи усіх розроблених нами функцій.

```
#ifndef _INT_PROCEDURES_GUARD_
#define _INT_PROCEDURES_GUARD_

#include <iostream>
using std::cout;
using std::cin;

void DigitsAverage();
void IsPalindrome();
void BezuHypothesis();
void BinaryForm();
void BinaryFormStr();
void HexaFormStr();
void Decomposition();
void Factorization();

void Decomp_1();
void Decomp_2();

#endif
```

1.7. Запитання та завдання для самоперевірки

1. Як обчислити молодшу цифру десяткового запису натурального числа?
2. Як знайти кількість цифр у записі числа в десятковій системі числення?
3. Як заповнюють таблицю ручної прокрутки алгоритму?
4. Як отримати обернений запис натурального числа?
5. Як у складній програмі, наприклад, для перевірки гіпотези Безу можна використати алгоритм – розв'язок простішої задачі?
6. Сформулюйте алгоритм переведення натурального числа у нову систему числення.
7. Які літери використовують для запису числа в шістнадцятковій системі числення?
8. Як перетворити на літеру ціле число з діапазону [0; 9]?
9. Як програмно перевірити, чи задане число є простим?
10. Як вдалося уникнути перевірок на простоту у функції *Decomposition*, що будує розклад числа на прості множники?
11. Завантажте програми за наведеним посиланням, запустіть їх на виконання.
12. Запропонуйте та випробуйте власні зміни та доповнення до програм.