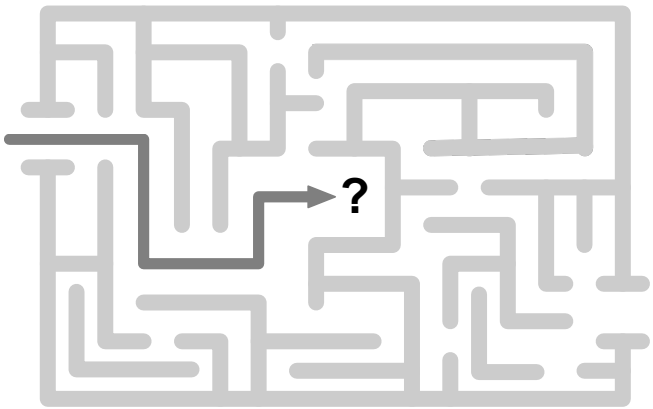
Міністерство освіти і науки України
Львівський національний університет імені Івана Франка

С. А. Ярошко

МЕТОДИ РОЗРОБКИ АЛГОРИТМІВ

Програмування мовою С++



Львів, 2019

ЗМІСТ

Вступ

Сучасні технології програмуваня переживають період бурхливого розвитку. Причинами цього явища є зростання потужностей комп'ютерної техніки, її здешевлення, створення всесвітньої мережі Internet, виникнення все нових і нових сфер застосування комп'ютерів і потреба у різноманітному програмному забезпеченні.

З метою полегшення життя програміста створено середовища візуального програмування. Тепер насправді «одним пальцем» (та за допомогою миші) можна будувати надскладний інтерфейс майбутньої програми: наділяти його різноманітними меню, панелями діалогу, піктограмами та хитромудрими перемикачами. І дуже часто за зовнішнім блиском усіх цих вікон, кнопок та лінійок прокрутки початкуючий програміст втрачає головне.

Що ж головне у програмі? Вікна та меню – це лише засіб обміну інформацією між програмою і зовнішнім світом. Вони прийшли на зміну перфокартам і значно спростили «спілкування» з комп'ютером, зробили його наочнішим і доступнішим широкому загалу користувачів. Вікна та меню – це лише форма, а зміст програми – в алгоритмах, які вона реалізує. Саме алгоритми визначають ефективність (а, отже, корисність) програми, саме алгоритми є найважливішим продуктом і надбанням комп'ютерної науки. Саме вони, вбрані у шати машинних команд чи об'єктно-орієнтованих програм, керують роботою комп'ютерів у всьому світі.

Навчитися складати алгоритми – одне з перших і головних завдань майбутнього програміста. На жаль, не існує простої відповіді на запитання, як це зробити? Майже кожна задача з програмування вимагає індивідуального підходу, недарма програмування порівнюють з мистецтвом. Однак як у живописі чи музиці є основні прийоми і навички, якими зобов'язаний володіти кожен митець, так і в програмуванні є класи задач, для розв'язування яких створено типові алгоритми. Відомі також певні підходи до розробки алгоритмів розв'язування нових задач, чи *методи розробки алгоритмів*.

Кожному випускникові Львівського національного університету імені Івана Франка, що вивчав програмування, необхідно навчитися класифікувати задачі, застосовувати до їхнього розв'язування відомі алгоритми, враховуючи особливості конкретної задачі, необхідно оволодіти основними методами розробки алгоритмів. Саме описові алгоритмів та методів їхньої розробки присвячено цей посібник. У його першій частині (р. 1–8) наведено розв'язки окремих типових задач, докладно описано процес створення та удосконалення кожного алгоритму. Її можна використовувати і як своєрідний «розв'язник» задач з програмування, і для того, щоб набути початкового досвіду проектування алгоритмів. Друга частина (р. 9–14) розрахована на читачів, що зважилися іти далі. Її присвячено описові та ілюстрації основних методів розробки алгоритмів. Вони стануть у пригоді тим, кому доведеться створювати програми для зовсім нових, ще не розв'язаних раніше задач.

Усі приклади програм у посібнику написано мовою C++. Сподіваємося, що їх зможуть легко зрозуміти ті читачі, які вивчають будь-яку іншу мову програмування високого рівня. Побудова алгоритму значною мірою залежить від вибору структур даних для відображення у програмі даних задачі. У кожному прикладі це питання буде обговорено зокрема, проте сподіваємось, що читач володіє знаннями щодо таких структур даних як масив, рядок, файл, список, стек, черга, дерево. Ні один з прикладів не претендує на досконалість. Автори розраховують на те, що читачі спробують запропонувати власні варіанти алгоритмів чи модифікації описаних.

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 здатися, що програми для розв'язування цих задач є дуже складними, 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 будемо старатися 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 необхідно знати їхню суму та кількість. Як обчислити суму цифр, з яких складається запис числа n? Для цього доведеться якимось чином 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.

Які змінні необхідні для побудови алгоритму? Очевидно, такі: n – задане число, c – наймолодша цифра, s – сума цифр, k – лічильник цифр, *average* – шукане середнє арифметичне.

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 n, m; cin >> n; m = n;
	// початкові значення суми та лічильника
	unsigned s = 0; unsigned k = 0;
	// перебираємо цифри введеного числа
	while (m > 0)
	{
		unsigned c = m % 10; // отримали наймолодшу цифру
		s += c; ++k;         // врахували її
		m /= 10;             // вилучили її
	}
```

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

```
        // перетворили ціле на дійсне і обчислили середнє арифметичне
        double average = s; average /= k;
        cout << "n = " << n << "   average = " << average << '\n';
}
```

Виконаємо ручну прокрутку цієї функції, щоб краще 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 *n*. Нехай дано число 1976, тоді

c	*s*	*k*	*n*	*n*>0?		*average*
	0	0	1976	⇒ +		
6	6	1	197	+		
7	13	2	19	+		
9	22	3	1	+		
1	23	4	0	–	⇐	5.75

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

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 превірити, 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 відповідним чином враховувати послідовні значення змінної *c* 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 *n* до змінної *s*. Як зробити, щоб на наступному кроці циклу змінна *s* отримала значення 67, а не 13? Наприклад, за допомогою оператора $s=s\times10+c$ (замість $s=s+c$). Легко переконатися, що виконання цього оператора на кожному кроці циклу дає змогу отримати в *s* обернений запис заданого числа. Проте є ще одна перешкода для отримання розв'язку задачі: під час виконання циклу змінна *n* отримала значення нуль, і ми не маємо з чим порівнювати побудоване нами число. Цю перешкоду можна легко подолати: використаємо в циклі не саме значення *n*, а його копію.

Тепер легко записати весь алгоритм, уточнивши призначення змінних: *n* – задане число; *m* – його копія; *c* – наймолодша цифра; *s* – нове число:

```
void isPolyndrome()
{
        cout << " Введіть натуральне число: ";
        unsigned n; cin >> n;
        // початкове значення нового числа та копія n
        unsigned s = 0; unsigned m = n;
        // перебираємо цифри введеного числа
        while (m > 0)
        {
                unsigned c = m % 10; // отримали наймолодшу цифру
                s = s*10 + c;        // врахували її
                m /= 10;             // і вилучили її
```

```
        }
        if (s == n) cout << n << "- паліндром\n";
        else cout << n << "- не паліндром\n";
        return;
}
```

Отже, ми успішно пристосували попередню функцію до нової задачі. Голівудські 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 і 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, поки 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, l (a ≤ b). Перевірити, чи виконується гіпотеза Безу для кожного натурального числа з проміжку [a, b] не більше, як за l кроків процесу.*

Розв'язання. Щоб побудувати алгоритм розв'язування цієї задачі, спробуємо краще 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 необхідно 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 з метою перевірки на кожному кроці циклу.

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

```
// функція побудови зворотнього запису числа
long long reverse(long long m)
{
        long long s = 0;
        while (m > 0)
        {
                s = s * 10 + m % 10;
                m /= 10;
        }
        return s;
}

void BezuHypothesis()
{
        cout << "\n *Перевірка гіпотези Безу*\n\n";
        unsigned a, b, l;
        cout << "Введіть дані a,b,l: ";
        cin >> a >> b >> l;
        // параметр циклу перебиратиме числа заданого діапазону
```

```
        for (unsigned n = a; n <= b; ++n)
        {
                unsigned k = 0;  // лічильник внутрішнього циклу
                long long m = n; // на першому кроці циклу - копія n,
                                 // на наступних - сума прямого і оберненого запису
                long long s = 0; // обернений запис числа m
                do
                {
                        m += s;
                        s = reverse(m);
                        ++k;
                }
                while (s != m && k <= l);
                // результати перевірки
                std::cout << "Для числа " << n << " за " << k - 1 << " крок(и) ";
                if (k > l) cout << "гіпотеза не виконується\n";
                else cout << "отримано паліндром " << s << '\n';
        }
        return;
}
```

У цій функції початкові значення для змінних *s* і *m* підібрано так, щоб перевірку, чи є паліндромом *n*, здійснити 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 найгроміздкішими 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.

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

Розв'язування. Пригадаємо собі алгоритм переведення цілого числа у нову систему: потрібно обчислити остачу від ділення цього числа на нову основу, отриману частку знову поділити на нову основу і 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 поставленої 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 **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: *n* – задане число; *p* – степінь десяти; *s* – число, що моделює двійковий запис числа *n*.

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

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

n % 2	*n*	*s*	*p*	*n*>0?	
	10	0	1	⇒ +	
0	5	0	10	+	
1	2	10	100	+	
0	1	10	1000	+	
1	0	1010	10000	–	⇐

Бачимо, що величини *s* і *p* 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 необхідно 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()
{
	using std::log;
	cout << "\n *Побудова запису числа у двійковій системі*\n\n";
	unsigned n;
	cout << "Введіть натуральне число: "; cin >> n;
	// готуємо місце для двійкового запису
	int k = log(double(n)) / log(2.0) + 1;
	char * str = new char[k + 1]; str[k] = '\0';
	while (n > 0)
	{
		--k;
		if (n % 2) str[k] = '1'; // остання двійкова цифра непарного числа - 1
		else str[k] = '0';       // наприкінці парного - двійковий 0
		n /= 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 необхідно 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()
{
	using std::log;
	cout << "\n *Побудова запису числа у шістнадцятковій системі*\n\n";
	unsigned n, m;
	cout << "Введіть натуральне число: "; cin >> n; m = n;
	// готуємо місце для шістнадцяткового запису
	int k = log(double(n)) / log(16.0) + 1;
	char * str = new char[k + 1]; str[k] = '\0';
	while (n > 0)
	{
		--k;
		unsigned c = n % 16;		// остання шістадцяткова цифра числа
		if (c < 10) str[k] = '0' + c; // звичайні цифри
		else str[k] = 'A' - 10 + c;   // старші цифри-букви
		n /= 16;
	}
	cout << "Запис " << m << " у шістнадцятковій системі: " << str << '\n';
	delete[] str;
	cout.setf(std::ios_base::hex, std::ios_base::basefield);
	cout << "Те саме засобами компілятора: " << m << '\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: $84 = 2\times2\times3\times7$. 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 та 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? Необхідно 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 *kSimple* набуває значення «істина», якщо k – просте, і значення «хиба» у протилежному випадку. Перевірку того, чи k – просте, можна записати так:

```
bool kSimple = true;
unsigned high = sqrt(double(k)) + 0.5;
for (unsigned j = 2; j <= high; ++j)
{     if (k % j == 0)
      {     kSimple = 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 для ми додали до нього 0.5, щоб взяти значення з надлишком (змоделювали заокруглення до цілого). Тепер можна записати всю функцію:

```
void decomposition1()
{
      using std::sqrt;
      cout << "\n *Розклад числа на прості множники*\n\n";
      unsigned n;
      cout << "Введіть натуральне число: "; cin >> n;
      // Знайдемо просте число, перевіримо, чи є воно дільником
      // починаємо друкувати розклад
      cout << n << " = ";
      // переберемо можливі дільники
      for (unsigned k = 2; k <= n; ++k)
      {
            // кожного з кандидатів перевіримо на простоту
            bool kSimple = true;
            unsigned high = sqrt(double(k)) + 0.5;
            for (unsigned j = 2; j <= high; ++j)
```

```
            {
                  if (k % j == 0)
                  {
                        kSimple = false;
                        break;
                  }
            }
            if (kSimple)
            {
                  unsigned m = k;      // степені простого k
                  while (n % m == 0)  // скільки разів k ділить n?
                  {
                        cout << k << " x ";  // стільки разів його друкуємо
                        m *= k;
                  }
            }
      }
      cout << "1\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 decomposition1 можна дійти висновку, що набагато вигідніше перевіряти на простоту виключно дільники числа *n*: обчислити остачу легше, ніж виконати згаданий цикл. Після нескладної переробки отримаємо:

```
void decomposition2()
{
      cout << "\n *Розклад числа на прості множники*\n\n";
      unsigned n;
      cout << "Введіть натуральне число: "; cin >> n;
      // Знайдемо дільник, перевіримо, чи він просте число
      // починаємо друкувати розклад
      cout << n << " = ";
      // переберемо можливі дільники
      for (unsigned k = 2; k <= n; ++k)
      {
            if (n % k == 0)
            {
                  // знайдений дільник перевіримо на простоту
                  bool kSimple = true;
                  unsigned high = sqrt(double(k)) + 0.5;
                  for (unsigned j = 2; j <= high; ++j)
                  {
                        if (k % j == 0)
                        {
                              kSimple = false;
                              break;
                        }
                  }
                  if (kSimple)
                  {
                        unsigned m = k;      // степені простого k
                        while (n % m == 0)  // скільки разів k ділить n?
                        {
                              cout << k << " x ";  // стільки разів його друкуємо
```

```
                        m *= k;
                    }
                }
            }
        }
        cout << "1\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 є зайвою: 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 decomposition2 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*\n\n";
        unsigned n;
        cout << "Введіть натуральне число: "; cin >> n;
        // Вилучатимемо дільники з числа, що гарантуватиме їхню простоту
        // починаємо друкувати розклад
        cout << n << " = ";
        unsigned k = 2; // перше просте число
        while (n > 1)
        {
                if (n % k == 0)
                {
                        // надрукуємо черговий дільник і вилучимо його з n
                        cout << k << " x ";
                        n/= k;
                }
                else ++k; // перевіримо наступного кандидата
        }
        cout << "1\n";  // друк розкладу завершено
        return;
}
```

Ми розглянули декілька 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 краща за неї.

Проте, нема межі досконалості, й алгоритм Decomposition теж можна покращити: цикл можна завершити ще до того, як частка *n* стане рівною 1. Достатньо переконатися, що *n* містить просте число, яке і буде останнім множником розкладу. Для цього замінимо умову

циклу ($n > 1$) на ($k*k <= n$), адже, як вже було сказано, достатньо перевірити на подільність числа $k = 2, 3, ..., \sqrt{n}$. Остаточне виправлення алгоритму пропонуємо зробити читачам.

1.6. Головна програма

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

```
#include <Windows.h>
#include "intProcedures.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: isPolyndrome(); break;
		case 3: BezuHypothesis(); break;
		case 4: binaryForm(); binaryFormStr();
			hexaFormStr(); break;
		case 5: decomposition1(); decomposition2();
			Decomposition(); break;
		default: cout << "Допобачення!\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.