МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ІВАНА ФРАНКА
ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ ТА ІНФОРМАТИКИ

Кафедра
Інформаційних систем

АЛГОРИТМИ ОБЧИСЛЮВАЛЬНИХ ПРОЦЕСІВ

Звіт виконання лабораторної роботи №2

“Дослідження алгоритмів перевірки на простоту”

Виконав:
Студент групи ПМі-12
Козій Д.Н.
Перевірили:
доц. кафедри ТОП Мельничин А. В.,
доц. кафедри ТОП Чипурко А. І.

Львів – 2026

Метод 1: Ітеративний перебір

Алгоритм:

1. Приймаємо число (n), яке потрібно перевірити.
2. Встановлюємо початковий статус числа як просте (змінна s = true).
3. Запускаємо цикл від 2 до квадратного кореня з числа n.
4. На кожній ітерації перевіряємо, чи ділиться n на лічильник і без остачі.
    a. Якщо ділиться — змінюємо статус на хибний (s = false).
5. Після завершення циклу повертаємо результат s.

Реалізація:

```
bool Method_1(int n) {
    bool s = true;
    for (int i = 2; i < sqrt(n); i++) {
        if (n % i == 0) s = false;
    }
    return(s);
}
```

Приклад роботи:

```
int main() {
    srand(time(NULL));
    int a;
    double ser1, ser2, ser3;
    auto start = chrono::high_resolution_clock::now();
    for (int i = 0; i < 1000000; i++) {
        a = 1 + rand() % 1000000;
        Method_1(a);
    }
    auto end = chrono::high_resolution_clock::now();
    ser1 = chrono::duration_cast<chrono::nanoseconds>(end - start).count();
    ser1 /= 1000000;
    cout << "a = " << a << " Method_1: " << ser1 << endl;
```

```
Microsoft Visual Studio Debug
a = 16728 Method_1: 1266.61
```

Висновок:

Цей метод є найпростішим для розуміння, проте найменш оптимізованим. Основний недолік полягає у відсутності раннього виходу з циклу: навіть якщо програма знаходить дільник, вона продовжує перевіряти всі інші числа аж до sqrt{n}. Через це при роботі з великими складеними числами алгоритм виконує багато зайвої роботи, що негативно впливає на його продуктивність.

Метод 2: Рекурсивний пошук дільника

Алгоритм:

1. Приймаємо число (n) та поточний дільник (i, за замовчуванням дорівнює 2).
2. Перевіряємо базові випадки: якщо $n \leq 2$, повертаємо true, якщо n == 2, інакше — false.
3. Перевіряємо подільність: якщо n ділиться на поточний дільник і без остачі, число не просте — повертаємо false.
4. Перевіряємо умову виходу: якщо квадрат поточного дільника більший за число (i * i > n), можливих дільників більше немає — повертаємо true.
5. Якщо жодна з умов не виконалася, здійснюємо рекурсивний виклик методу для того самого n, але з наступним дільником (i + 1).

Реалізація:

```
bool Method_2(int n, int i) {
    if (n <= 2) return (n == 2);
    if (n % i == 0) return false;
    if (i * i > n) return true;

    return Method_2(n, ++i);
}
```

Приклад роботи:

```
start = chrono::high_resolution_clock::now();
for (int i = 0; i < 1000000; i++) {
    a = 1 + rand() % 1000000;
    Method_2(a);
}
end = chrono::high_resolution_clock::now();
ser2 = chrono::duration_cast<chrono::nanoseconds>(end - start).count();
ser2 /= 1000000;
cout << "a = " << a << " Method_2: " << ser2 << endl;
```

```
Microsoft Visual Studio Debug
a = 29022 Method_2: 222.538
```

Висновок:

Метод використовує рекурсивний підхід і вирішує проблему попереднього алгоритму: він одразу завершує роботу, як тільки знаходить перший дільник. Однак, на практиці рекурсія вимагає виділення додаткової пам'яті в стеку викликів для кожної ітерації. Для дуже великих простих чисел глибока рекурсія може спричинити помилку переповнення стеку та вносить додаткові накладні витрати часу на виклик функцій, що робить цей алгоритм менш надійним для важких обчислень.

Метод 3: Оптимізований перебір

Алгоритм:

1. Перевіряємо очевидні базові випадки: якщо $n \leq 1$, то хибність; якщо $n \leq 3$, то істина.
2. Відкидаємо всі числа, кратні 2 та 3, перевіривши остачу від ділення.
3. Запускаємо цикл, починаючи з i = 5, до sqrt{n}. Крок циклу становить 6.
4. На кожній ітерації перевіряємо, чи ділиться n на i або на i+2 (охоплюючи таким чином числа вигляду 6k-1 та 6k+1).
    a. Якщо знаходиться дільник, повертаємо false.
5. Якщо цикл пройшов повністю і дільників не знайдено — повертаємо true.

Реалізація:

```
bool Method_3(int n) {
    if (n <= 1) return false;
    if (n <= 3) return true;

    if (n % 2 == 0 || n % 3 == 0) return false;

    for (int i = 5; i * i <= n; i += 6) {
        if (n % i == 0 || n % (i + 2) == 0)
            return false;
    }

    return true;
}
```

Приклад роботи:

```
start = chrono::high_resolution_clock::now();
for (int i = 0; i < 1000000; i++) {
    a = 1 + rand() % 1000000;
    Method_3(a);
}
end = chrono::high_resolution_clock::now();
ser3 = chrono::duration_cast<chrono::nanoseconds>(end - start).count();
ser3 /= 1000000;
cout << "a = " << a << " Method_3: " << ser3 << endl;
```

```
Microsoft Visual Studio Debug
a = 1477 Method_3: 89.1354
```

Висновок:

Це найбільш досконалий та математично обґрунтований метод з трьох представлених. Шляхом раннього відсіювання чисел, кратних 2 і 3, та використання кроку циклу в 6, кількість операцій ділення зменшується втричі порівняно з базовим перебором (Метод 1). Він також містить раннє переривання без накладних витрат пам'яті, які властиві рекурсії (Метод 2). Відповідно, цей алгоритм є найшвидшим, найбезпечнішим і оптимально підходить для перевірки великих чисел у реальних програмах.