Федеральное государственное образовательное бюджетное учреждение высшего образования

«ФИНАНСОВЫЙ УНИВЕРСИТЕТ ПРИ ПРАВИТЕЛЬСТВЕ

РОССИЙСКОЙ ФЕДЕРАЦИИ»

(Финансовый университет)

**Колледж информатики и программирования**

**Отчёт о проделанной работе  
по практическому занятию № 2 по дисциплине**

**МДК.02.02 «Криптографические средства и методы защиты информации»**

На тему: «Проверка чисел на простоту».

Студенты группы 3ОИБАС-1221

|  |  |
| --- | --- |
| Верстунин П. В.,  Кован Н. В.,  Осадчий И. А. | «6» ноября .2023 г. |

Основная профессиональная образовательная программа по специальности

10.02.05 Обеспечение информационной безопасности автоматизированных систем

Форма обучения очная

Проверил: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Рой А.В.

Москва

2023

**Введение:**

Простые числа, являющиеся уникальными в мире числовой теории, играют важнейшую роль в различных областях математики и информационных технологий. Они служат основой для множества алгоритмов и криптографических методов, используемых в современном мире. Поэтому их идентификация и проверка на простоту представляют собой важный аспект вычислительной математики.

В данной работе рассмотрим методы и алгоритмы для проверки чисел на простоту, а также их практическое применение. Мы изучим классические методы, такие как тесты деления и тест Ферма, и сосредоточимся на более современных и эффективных алгоритмах, используемых в современных вычислительных задачах.

**Цель работы:**

Изучение и практическое применение методов проверки чисел на простоту с целью освоения алгоритмов, развития навыков программирования и понимания математических концепций, связанных с простыми числами.

**Задачи:**

* Изучение теоретических основ: Изучить основные математические концепции, связанные с простыми числами, включая тест Ферма, тест Миллера-Рабина, решето Эратосфена и другие методы проверки чисел на простоту.
* Программирование алгоритмов: Написать программы на выбранном языке программирования для реализации методов проверки чисел на простоту, включая решето Эратосфена, тест Ферма и тест Миллера-Рабина.
* Документация и выводы: Составить документацию к коду, включая описание каждого алгоритма, принципы их работы и результаты тестирования

**Изучение алгоритмом проверки чисел на простоту**

**1. Проверка чисел на простоту с использованием корня:**

Простое число - натуральное число больше 1, которое имеет только два делителя: 1 и само число. Проверка числа на простоту означает определение, является ли данное число простым или составным (имеющим более двух делителей). Существует множество методов для проверки чисел на простоту, и одним из них является метод, использующий корень числа.

Идея этого метода заключается в том, что если число n имеет делитель больше √n, то он также должен иметь делитель меньше √n. Таким образом, для проверки числа √n на простоту, достаточно проверить делители от 2 до √n. Если ни одно из этих чисел не является делителем n, то n является простым числом.

**2. Решето Эратосфена:**

Решето Эратосфена — это один из самых древних и эффективных алгоритмов для нахождения всех простых чисел до заданного числа n. Алгоритм получил свое название в честь древнегреческого математика Эратосфена Киренского, который описал его метод в III веке до н. э.

Принцип работы решета Эратосфена очень прост: начиная с числа 2, отмечаем все его кратные числа как составные (не простые). Затем переходим к следующему непомеченному числу и повторяем процесс, пока неисчерпаем все числа до n. После завершения алгоритма все непомеченные числа останутся простыми.

Вот как это делается шаг за шагом:

* Инициализация:

Создаем список чисел от 2 до n.

* Шаг 1:

Начинаем с первого числа в списке (которое является 2). Помечаем все его кратные числа (4, 6, 8, и так далее) как составные.

* Шаг 2:

Переходим к следующему непомеченному числу в списке (в данном случае, 3) и помечаем все его кратные числа (6, 9, 12, и так далее) как составные.

* Шаг 3:

Повторяем Шаг 2 с каждым непомеченным числом в списке, пока не достигнем конца списка.

* Завершение:

Все непомеченные числа в списке теперь являются простыми числами.

**3. Тест Ферма (или тест Ферма-Эйлера):**

Тест Ферма (или тест Ферма-Эйлера) - это вероятностный алгоритм для проверки числа на простоту. Алгоритм назван в честь математика Пьера де Ферма, который впервые сформулировал его в XVII веке на основе малой теоремы Ферма.

Основная идея теста Ферма заключается в том, что если n - простое число, то для любого целого числа a в диапазоне от 1 до n−1 выполняется следующее условие, известное как малая теорема Ферма:

Это означает, что если выбрать случайное целое число a и проверить это условие, и оно не выполняется, то число n точно составное. Если условие выполняется, то это еще не гарантирует, что n простое (такие числа называются псевдопростыми по основанию a), но вероятность ошибки с каждой итерацией уменьшается.

Однако важно отметить, что тест Ферма не является идеальным и не гарантирует 100% точность для всех значений n. Существуют составные числа, которые могут обмануть тест Ферма для определенных баз a, такие числа называются Кармайкловыми числами.

В практических приложениях тест Ферма обычно используется в сочетании с другими тестами простоты, такими как тест Миллера-Рабина, для обеспечения более надежного определения простоты числа.

**4. Тест Миллера-Рабина:**

Тест Миллера-Рабина (Miller-Rabin primality test) - это вероятностный алгоритм проверки числа на простоту. Этот тест был разработан в 1976 году Дональдом Миллером (Donald Knuth) и Гари Рабином (Gary Miller) и представляет собой один из наиболее эффективных методов определения простоты больших чисел в компьютерной математике и криптографии.

Основная идея теста Миллера-Рабина основана на малой теореме Ферма и исследует свойства возведения в степень в конечных полях (modular arithmetic). Если n - простое число, то для любого целого числа a, где 1<a<n−1, выполняется следующее условие:

Тест Миллера-Рабина использует этот факт для проверки числа на простоту. Однако вместо проверки условия для всех возможных a в диапазоне от 2 до n−2, алгоритм выбирает случайное число a и проверяет это условие. Если условие не выполняется, то число n точно составное. Если условие выполняется, число n может быть простым, но не обязательно (такие числа называются псевдопростыми по основанию a). Повторяя тест с разными случайными значениями a, можно увеличить вероятность правильного определения простоты числа.

Важно отметить, что количество итераций теста Миллера-Рабина влияет на его точность. Чем больше итераций, тем меньше вероятность ошибки. В большинстве случаев небольшое количество итераций (например, 5-10) обеспечивает достаточную точность для определения простоты больших чисел.

Комбинирование теста Миллера-Рабина с другими методами, таким как тест Ферма, улучшает надежность проверки числа на простоту в приложениях криптографии и алгоритмике.

**Практические задания:**

**Практическая часть.**

**Решение алгоритмов проверки чисел на простоту  
Программы для проверки чисел на простоту**

**Задание 1.**

Проверьте следующие пять чисел на простоту (n - номер по списку в журнале): (33-n)^2+1, 2^(n+1)-1, (2^2^(n mod 4))+1, дата занятия в формате ГГГГ-ММ-ДД, дата своего рождения ГГГГ-ММ-ДД. Проверку надо будет сделать следующими методами: с помощью корня, решето Эратосфена, тест Ферма, теста Миллера-Рабина.

**Задание 2.** Напишите программы для проверки любых чисел на простоту теми же методами, что и в пункте 1. Сравните результаты проверки пяти числ пункта 1. Проверьте работоспособность программ ещё на нескольких произвольных примерах. Поместите в отчёт исходный текст работающей версии программ и необходимые копии экрана или консоли.

#include <iostream>  
#include <random>  
#include <math.h>  
#include <chrono>  
  
#define **cout**(text) std::cout << text << std::endl  
#define **coutd**(text, data) std::cout << text << " = " << data << std::endl  
#define **coutt**(test\_name, data) std::cout << "Executing " << test\_name << "(" << data << ")" << std::endl  
#define **couttt**(test\_name, data, time) std::cout << "Runtime of " << test\_name << "(" << data << "): " << time << "ms (" << time/1000 << "s)" << std::endl  
#define **coutr**(test\_name, data, res, time) std::cout << "Result of " << test\_name << "(" << data <<"): " << res << std::endl; **couttt**(test\_name, data, time)  
#define **coutrnt**(test\_name, data, res) std::cout << "Result of " << test\_name << "(" << data <<"): " << res << std::endl  
#define **td** std::cout << "----------------------------------------------------------------------------------------------" << std::endl  
#define **rd** std::cout << std::endl << "\033[32m" << "----------------------------------------------------------------------------------------------" << "\033[0m" << std::endl << std::endl  
#define **brd** std::cout << std::endl << "\033[31m" << "----------------------------------------------------------------------------------------------" << "\033[0m" << std::endl << std::endl  
  
#define **cexit**(text) {**cout**(text); *return* 0;}  
#define **ll** *long long  
  
bool* REQUIRE\_BENCHMARK = *false*;  
  
*int* BaseCheck(**ll** n);  
*int* PowerMod(**ll** base, **ll** exponent, **ll** modulus);  
*bool* IsPrimeByRoot(**ll** n);  
*bool* IsPrimeByMillerRabinTest(**ll** n, **ll** k);  
*bool* IsPrimeByFerma(**ll** n, **ll** k);  
*bool* IsPrimeBySieveEratosthenes(**ll** n);  
*void* ExecuteTests(**ll** n1, **ll** n2, **ll** n3, **ll** n4, **ll** n5, **ll** k);  
*void* ExecuteTest(**ll** n, **ll** k);  
  
*int* main(*int* argc) {  
 *if* (argc >= 2) REQUIRE\_BENCHMARK = *true*;  
 **ll** n1\_1, n1\_2, n1\_3, *//5* n2\_1, n2\_2, n2\_3, *//13* n3\_1, n3\_2, n3\_3, *//22* n5\_1, n5\_2, n5\_3, n4, k;  
  
 n1\_1 = std::pow(33-5, 2) + 1 ;  
 n1\_2 = std::pow(5+1, 2) - 1;  
 n1\_3 = std::pow(2, std::pow(2, 5 % 4)) + 1;  
  
 n2\_1 = std::pow(33-13, 2) + 1 ;  
 n2\_2 = std::pow(13+1, 2) - 1;  
 n2\_3 = std::pow(2, std::pow(2, 13 % 4)) + 1;  
  
 n3\_1 = std::pow(33-22, 2) + 1 ;  
 n3\_2 = std::pow(22+1, 2) - 1;  
 n3\_3 = std::pow(2, std::pow(2, 22 % 4)) + 1;  
  
 n5\_1 = 20050906;  
 n5\_2 = 20050719;  
 n5\_3 = 20050606;  
  
 n4 = 20231107;  
  
 **cout**("Insert number of iterations for Ferma and Miller-Rabin test");  
 std::cin >> k;  
  
 ExecuteTests(n1\_1, n2\_1, n3\_1, n4, n5\_1, k);  
 **brd**;  
 ExecuteTests(n1\_2, n2\_2, n3\_2, 0, n5\_2, k);  
 **brd**;  
 ExecuteTests(n1\_3, n2\_3, n3\_3, 0, n5\_3, k);  
  
 *int* t1, t2, t3, t4, t5, kk;  
  
 **cout**("Insert 5 number for tests");  
 std::cin >> t1 >> t2 >> t3 >> t4 >> t5;  
  
 **cout**("Insert number of iterations for Ferma and Miller-Rabin test");  
 std::cin >> kk;  
  
 **brd**;  
 ExecuteTests(t1, t2, t3, t4, t5, kk);  
 *return* 0;  
}  
  
*int* BaseCheck(**ll** n) {  
 *if* (n <= 1) *return* 0;  
 *if* (n <= 3) *return* 1;  
 *return* 2;  
}  
  
*bool* IsPrimeByRoot(**ll** n) {  
 *if* (*auto* check = BaseCheck(n); check != 2) *return* (*bool*)check;  
 *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*;  
}  
  
*bool* IsPrimeByMillerRabinTest(**ll** n, **ll** k) {  
 *if* (*auto* check = BaseCheck(n); check != 2) *return* (*bool*)check;  
  
 **ll** r = 0;  
 **ll** d = n - 1;  
  
 *while* (d % 2 == 0) {  
 r++;  
 d /= 2;  
 }  
  
 std::random\_device dev;  
 std::mt19937 rng(dev());  
  
 *for* (*int* i = 0; i < k; i++) {  
 *int* a = 2 + std::uniform\_int\_distribution<std::mt19937::result\_type>(2,n - 2)(rng);  
 *int* x = (**ll**) std::pow(a, d) % n;  
  
 *if* (x == 1 || x == n - 1) *continue*;  
  
 *for* (*int* j = 0; j < r - 1; j++)  
 *if* (x = (**ll**) std::pow(x, 2) % n; x == n - 1) *break*;  
  
 *if* (x != n - 1) *return false*;  
 }  
  
 *return true*;  
}  
  
*bool* IsPrimeByFerma(**ll** n, **ll** k) {  
 *if* (n <= 1 || n % 2 == 0) {  
 *return* n == 2;  
 }  
  
 std::random\_device dev;  
 std::mt19937 rng(dev());  
  
 *for* (*int* i = 0; i < k; i++)  
 *if* (PowerMod(  
 std::uniform\_int\_distribution<std::mt19937::result\_type>(2,n - 2)(rng) + 2,  
 n - 1,n) != 1)  
 *return false*;  
  
 *return true*;  
}  
  
*bool* IsPrimeBySieveEratosthenes(**ll** n) {  
 *bool*\* isPrime = *new bool*[n + 1];  
  
 *for* (*int* i = 2; i <= n; i++)  
 isPrime[i] = *true*;  
  
 *for* (*int* p = 2; p \* p <= n; p++)  
 *if* (isPrime[p])  
 *for* (*int* i = p \* p; i <= n; i += p)  
 isPrime[i] = *false*;  
  
 *return* isPrime[n];  
}  
  
*int* PowerMod(**ll** base, **ll** exponent, **ll** modulus) {  
 *int* result = 1;  
 *while* (exponent > 0) {  
 *if* (exponent % 2 == 1)  
 result = (result \* base) % modulus;  
 base = (base \* base) % modulus;  
 exponent /= 2;  
 }  
 *return* result;  
}  
  
*void* ExecuteTests(**ll** n1 ,**ll** n2, **ll** n3, **ll** n4, **ll** n5, **ll** k) {  
 ExecuteTest(n1, k);  
 **rd**;  
 ExecuteTest(n2, k);  
 **rd**;  
 ExecuteTest(n3, k);  
 **rd**;  
 *if* (n4 != 0) {ExecuteTest(n4, k); **rd**;}  
 ExecuteTest(n5, k);  
}  
  
*void* ExecuteTest(**ll** n, **ll** k) {  
 **coutt**("IsPrimeByRoot", n);  
 *auto* start = std::chrono::system\_clock::now();  
 *auto* result = IsPrimeByRoot(n);  
 *auto* end = std::chrono::system\_clock::now();  
 *if* (REQUIRE\_BENCHMARK) { **coutr**("IsPrimeByRoot", n, result, std::chrono::duration\_cast<std::chrono::milliseconds>(start - end).count()); }  
 *else* **coutrnt**("IsPrimeByRoot", n, result);  
  
 **td**;  
 **coutt**("IsPrimeByMillerRabinTest", n);  
 start = std::chrono::system\_clock::now();  
 result = IsPrimeByMillerRabinTest(n, k);  
 end = std::chrono::system\_clock::now();  
 *if* (REQUIRE\_BENCHMARK) { **coutr**("IsPrimeByMillerRabinTest", n, result, std::chrono::duration\_cast<std::chrono::milliseconds>(start - end).count()); }  
 *else* **coutrnt**("IsPrimeByMillerRabinTest", n, result);  
  
 **td**;  
 **coutt**("IsPrimeByFerma", n);  
 start = std::chrono::system\_clock::now();  
 result = IsPrimeByFerma(n, k);  
 end = std::chrono::system\_clock::now();  
 *if* (REQUIRE\_BENCHMARK) { **coutr**("IsPrimeByFerma", n, result, std::chrono::duration\_cast<std::chrono::milliseconds>(start - end).count()); }  
 *else* **coutrnt**("IsPrimeByFerma", n, result);  
  
 **td**;  
 **coutt**("IsPrimeBySieveEratosthenes", n);  
 start = std::chrono::system\_clock::now();  
 result = IsPrimeBySieveEratosthenes(n);  
 end = std::chrono::system\_clock::now();  
 *if* (REQUIRE\_BENCHMARK) { **coutr**("IsPrimeBySieveEratosthenes", n, result,  
 std::chrono::duration\_cast<std::chrono::milliseconds>(start - end).count()); }  
 *else* **coutrnt**("IsPrimeBySieveEratosthenes", n, result);  
}

**Заключение:**

Мы научились определять простые числа 4 способами, а так же написали программу, основываясь на полученных знаниях