Modular Arithmetic Homework

Identifying information:

Yacine ELHAMER -

Primality Tests:

A primality testing algorithm is an algorithm that is used to determine whether a natural number is a prime number or not. There are many primality testing algorithms in existance with varying properties, and in this report we will attempt to examine and implement some of them, as well as classifying them by their temporal complexity.

Prime numbers:

Prime numbers are natural numbers that have exactly two divisors; For example, the number 1 is not prime since it has only one divisor, while the number 2 is prime since it has exactly two divisors.

Prime numbers have many applications in applied mathematics, one of its most prominant use cases is in cryptographic algorithms. One of the most widely-spread cryptographic algorithms is the RSA algorithm which is used to secure many impotant communicatoin protocoles (such as SSL in HTTPS), and for this algorithm to function it must be able to generate large prime numbers in a short period of time; Which brings us to the necessity of primality testing algorithms

1. Trial division:

This primality test is one of the most basic primality tests, it has a complexity of O(sqrt(N)), and its algorithm is as follows

```c
_Bool isPrime(int number) {
    for(int i = 2; i * i <= x; i++) {
        if(x % i == 0)
            return 0;
    }

    return 1;
}
```

The way this algorithm works is that it find the number of divisors a number has, and that is by checking all the possible divisors for that number; If the number of divisors is exactly 2, then the algorithm returns True, therby signifying that the number is prime, otherwise is returns False.

This algorithm is optimized by only checking the first sqrt(n) values, and that is because for any natural number n, for every divisor that is bigger than sqrt(n), there exists another associated divisor that is smaller than sqrt(n).

Considering any divisor d of a number n, the following equation is true: *d * n/d <= sqrt(n) * sqrt(n)*, which means that either *n/d* or *d* are smaller than *sqrt(n)*. And since this is valid for all decompositions of *n*, then the trial division algorithm is true.

2. Fermat's primality test:

This algorithm is a probabilistic algorithm, in that it does not assert whether a number is prime or not with absolute confidence, rather, it gives a probability of whether it is or not. The output probability can be increased by increasing the numbers tested.

According to fermat's little theorem, the following equation holds true for all prime numbers and generally does not for composite numbers:

$$a^p \equiv a \pmod{p}.$$

Figure 1: flt_general

Where is any prime number, and a is any natural number.

A corollary that follows is the following theorem:

$$a^{p-1} \equiv 1 \pmod{p}.$$

Figure 2: flt

Where a is not divisible by p.

The second theorem can be utilized to construct an efficient primality test, and that is by picking the number *a* to be in the range [2:*p*-2], thereby guaranteeing that it is not divisible by *p*. After this, we test a number of possible values for *a*, and if fermat's property holds true for all tested values of *a*, then we have a probability that *p* is prime which corresponds to the number of values tested for *a*.

One thing should be kept in mind when using this theorem, and it is that this property also holds true for some composite numbers; However, the density of these numbers is very low, specifically, only 646 numbers in the first 10^9 natural numberes, which make this primality test remain valuable.

An implementation of this primality test is as follows:

```c
_Bool fltPrime(int number, unsigned int accuracy) {
    int a;

    // Because a has to be in the range [2; number-2]
    if (n < 4) {
        return n == 2 || n == 3;

    // Check values
    for(int i = 0; i < accuracy; i++) {
        // Set the number a
        a = 2 + rand() % (number-3);

        // Check if a^p-1 = 1 mod p
        if(power(a, number-1) % number == 1)

            // The number is not prime
            return 0;
    }

    // The number is prime given the provided accuracy
    return 1;
}
```

As for the time complexity of this algorithm, assuming that the *power()* function being used is the binary exponentiation function which is O(log(n)), then this primality test is O(N*log(N)).

Miller-Rabin primality test:

This theorem extends the ideas from the previous algorithm by decomposing the problem into a set of smaller problems, specifically, by testing the divisibility against smaller integers which is going to improve the runtime of this algorithm.

The algorithm work by decomposing the previous problem of checking the divisibility of *a^(n-1) - 1* by *n*, into checking the divisibility of *a^d - 1* by *n*, or $a^{(d*2}r) + 1)$ by n. Where *a* is a base, n is our prime number, *d* is a divisor of *n-1*, and *r* is the number of *2*-divisors of n-1.

This works because given that all prime numbers are odd, *n-1* must be even, which means that it can be rewritten as: *2^r * d* where *r* is a strictly positive number. Given this result, fermat's little theorem could be extended to the following:

As a result, the problem's runtime gets improved drastically since to prove that a number is prime, only one component of left side of the congruency has to be divisible by *n*, which gives us a much better best-case scenario considering that if for all bases *a*, if only the first-tested component is divisible by *n*, then the other components need not be tested.

$$\begin{aligned}
a^{n-1} \equiv 1 \bmod n \quad &\iff a^{2^s d} - 1 \equiv 0 \bmod n \\
&\iff (a^{2^{s-1}d} + 1)(a^{2^{s-1}d} - 1) \equiv 0 \bmod n \\
&\iff (a^{2^{s-1}d} + 1)(a^{2^{s-2}d} + 1)(a^{2^{s-2}d} - 1) \equiv 0 \bmod n \\
&\vdots \\
&\iff (a^{2^{s-1}d} + 1)(a^{2^{s-2}d} + 1) \cdots (a^{d} + 1)(a^{d} - 1) \equiv 0 \bmod n
\end{aligned}$$

Figure 3: miller-rabin

An implementation for this algorithm is as follows:

```
typedef unsigned long u64;
typedef unsigned long long u128;

u64 binpower(u64 base, u64 e, u64 mod) {
    u64 result = 1;
    base %= mod;
    while (e) {
        if (e & 1)
            result = (u128)result * base % mod;
        base = (u128)base * base % mod;
        e >>= 1;
    }

    return result;
}

bool check_composite(u64 n, u64 a, u64 d, int s) {
    u64 x = binpower(a, d, n);
    if (x == 1 || x == n - 1)
        return false;
    for (int r = 1; r < s; r++) {
        x = (u128)x * x % n;
        if (x == n - 1)
            return false;
    }

    return true;
};
```

```
bool MillerRabin(u64 n, int iter=5) {
    if (n < 4)
        return n == 2 || n == 3;

    int s = 0;
    u64 d = n - 1;
    while ((d & 1) == 0) {
        d >>= 1;
        s++;
    }

    for (int i = 0; i < iter; i++) {
        int a = 2 + rand() % (n - 3);
        if (check_composite(n, a, d, s))
            return false;
    }

    return true;
}
```

Bach's deterministic version of the Miller-Rabin primality test:

Bach improved on the previous algorithm by proving that it could be made deterministic by testing only the bases a such that $a <= 2ln(n)^2$. However, while this variant of the algorithm being deterministic is certainly a positive, the number of bases that need to be tested is still too large to be deployed in production, which prompted people to work on improving this algorithm, and indeed a solution was found.

Research has showed that for 32-bit numbers (up to 4294967295), only the first 4 prime bases need to be checked, i.e. 2, 3, 5, and 7. Additionally, for 64-bit numbers (up to 18446744073709551616), only the first 12 prime numbers need to be checked, i.e. 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, and 37.

The previous result makes the use of this primality test much more efficient than its counterparts.

An implementation for it is as follows:

```
typedef unsigned long u64;

bool MillerRabin(u64 n) {
    if (n < 2)
        return false;

    int r = 0;
    u64 d = n - 1;
    while ((d & 1) == 0) {
        d >>= 1;
        r++;
    }

    for (int a : {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37}) {
        if (n == a)
            return true;
        if (check_composite(n, a, d, r))
            return false;
    }

    return true;
}
```