University of Science - HCMUS

Object-Oriented Programming Reporting

Topic: Major Assignment #1

Lecturer: Nguyễn Ngọc Long

Members: Trần Lê Minh – 20280066

Mai Thị Thảo Ly – 20280064

20280066 – Trần Lê Minh
20280064 – Mai Thị Thảo Ly

Mục Lục

Phân công công việc:

Họ Tên	MSSV	Phân Công
Trần Lê Minh	20280066	• Quản lý tiến độ làm việc • Lên ý tưởng • Phân tích các rủi ro của giải pháp • Viết cách sườn bài code, viết file header.h và file function.cpp, các hàm trong bài tập dưới dạng friend funtion. • Viết sườn file báo cáo và bổ sung.
Mai Thị Thảo Ly	20280064	• Tìm hiểu về thuật toán Miller Rabin • Lên ý tưởng cách thực hiện phép toán số học • Viết file main (20280066_20280064.cpp) • Viết file báo cáo • Đổi một số hàm từ friend function sang dạng con trỏ.

20280066 – Trần Lê Minh
20280064 – Mai Thị Thảo Ly

I. Giới Thiệu Bài Tập Lớn:

1. Thông tin về bài tập:

Yêu cầu: Xây dựng lớp UintN biểu diễn khái niệm số nguyên không dấu có giá trị rất lớn. Định nghĩa các phép toán số học và so sánh trên số nguyên kể trên. Định nghĩa phương thức xác định một số UintN có nguyên tố hay không. Yêu cầu tối ưu dữ liệu để chương trình không bị chậm với số lớn.

Mục đích: Viết chương trình cho phép nhập vào 2 số UintN, tính và xuất ra kết quả các phép toán số học, so sánh và xác định có phải số nguyên tố không dựa trên 2 số nguyên đó, và xuất ra kết quả của số nguyên tố đầu tiên lớn hơn 2 số nguyên đó.

2. Đề xuất giải pháp:

2.1 Phương hướng giải quyết các vấn đề:

- Định dạng kiểu số nguyên: Dạng chuỗi (string) hoặc dạng mảng (vector<char> hoặc vector<int>).
- Định nghĩa các phép toán số học: Chuyển về dạng nhị phân và thực hiện các phép tính sau đó chuyển về lại dạng số hoặc định nghĩa bằng phương pháp thông thường như ở tiểu học.
- Định nghĩa phương thức nhập: Sử dụng các constructor cho phép nhập vào string, int, unsigned long long, ... và các kiểu nhập vào từ chính nó.
- Định nghĩa phép so sánh: Sử dụng cách so sánh dạng nhị phân hoặc giống như phương pháp so sánh thông thường như ở tiểu học.
- Xác định số nguyên tố: Sử dụng phương pháp brute force hoặc phương pháp kiểm tra tính nguyên tố (primality test) có thể có sai số nhưng không đáng kể đối với phương pháp Miller Rabin.
- Tìm số nguyên tố lớn hơn: So sánh chọn ra số lớn hơn giữa 2 số và cộng từ từ lên rồi xác định từng số sau khi cộng từ từ lên có phải số nguyên tố không rồi xuất ra.

2.2 Lựa chọn công cụ IDE và ngôn ngữ:

- Công cụ lập trình IDE: Microsoft Visual Studio.
- Ngôn ngữ lập trình: C++.

3. Chọn giải pháp:

- Định dạng kiểu số nguyên: Dạng chuỗi (string).
- Định nghĩa các phép toán số học: Sử dụng phương pháp thông thường như chúng ta thường tính ở tiểu học.

- Định nghĩa phép so sánh: Sử dụng phương pháp thông thường như chúng ta thường làm ở tiểu học.

Chúng ta biết rằng string tối ưu memory tốt hơn so với vector<char> , bên cạnh đó vector<int> tốn dữ liệu nhiều hơn so với vector<char>. Tuy nhiên vector dùng để sử lý thuật toán đối với từng phần tử sẽ tối ưu hơn so với string. Ta nhận thấy rằng nếu sử dụng phương pháp chuyển nhị phân thì sẽ phải tốn rất nhiều bộ nhớ nếu lưu bằng vector. Kèm theo ta muốn code được sử dụng chung nhất có thể 1 loại kiểu biến. Do đó chúng ta xác định định dạng kiểu số nguyên là string.

Đối với phép toán số học và phép so sánh, chúng ta nhận thấy nếu chuyển về lại dạng nhị phân đúng thì thuật toán sẽ hoạt động nhanh và mượt hơn nhưng nếu số nguyên nhập vào là số rất rất lớn thì chuỗi nhị phân sẽ lên tới hàng ngàn, hàng triệu chữ số, khi đó nó sẽ vượt quá maximum length của string trong C++ nên chúng ta quyết định sử dụng phương pháp tính và so sánh thông thường chúng ta hay làm ở tiểu học .

- Định nghĩa phương thức nhập: Sử dụng các constructor cho phép nhập vào string, int, chính UintN và một cái constructor khởi tạo trống.
- Xác định số nguyên tố: Sử dụng phương pháp Miller Rabin

Yêu cầu bài toán là tối ưu dữ liệu để chương trình không bị chậm với số lớn do đó chúng ta sử dụng phương pháp Miller Rabin để kiểm tra số nguyên tố (có thể có sai số rất nhỏ).

4. Đánh giá tiến độ và bài tập:

Mọi tiến độ công việc đều hoàn thành đúng mới mục tiêu đề ra, trừ phần thuật toán Miller Rabin. Chúng em bị vướng khá nhiều bug ở phần này.

Đánh giá tiến độ làm việc:

- Trần Lê Minh: 9/10
- Mai Thị Thảo Ly: 10/10

Phần bài tập chúng em đã hoàn thành tốt trong việc tối ưu dữ liệu và thuật toán để không bị chạy chậm đối với số lớn ở các phép toán số học và so sánh. Riêng phần kiểm tra số nguyên tố, thuật toán chỉ chạy mượt đối với những số dưới 50 chữ số, trên 50 sẽ delay nhẹ, trên 75 chữ số sẽ bị delay khoảng 5-10s, trên 150 chữ số là khoảng 30s. Và phần tìm số nguyên tố tiếp theo lớn hơn bị ảnh hưởng bởi thuật toán kiểm tra số nguyên tố. Do đó thời gian chạy chương trình sẽ bị ảnh hưởng theo.

Đánh giá bài tập (mức độ hoàn thiện): 8.5/10

II. THỰC HIỆN:

A. Xác định các lớp của bài toán:

Do yêu cầu bài toán khá đơn giản nên chỉ cần 1 lớp là dược

- **Class UintN:**
  - Thuộc tính: string BigInt.
  - Phương thức:
    - Constructor: Constructor có đối số (string a hoặc int n hoặc const UintN& a) và constructor không đối số.
    - Các thuật toán về phép toán số học.
    - Các thuật toán về phép so sánh.
    - Thuật toán nhập và xuất (Cin, Cout) trực tiếp.
    - Thuật toán kiểm tra số nguyên tố.
    - Và một vài thuật toán helper function (những thuật toán phụ cần thiết để giúp chạy những hàm trên).

B. Cài đặt và phương thức hoạt động các hàm:

1. Constructor:

- UintN(string a); UintN(int n); UintN(const UintN& a);

Điểm chung của 2 hàm đầu tiên (string, int) đó chính là số nhập vào sẽ được push_back từng phần từ cuối tới đầu vào trong thuộc tính BigInt. Điểm khác biệt đó chính là ở dạng string, từng phần tử sẽ được chuyển đổi – đi '0' ở dạng string mang số hiệu ascii là 48. Còn constructor cuối thì chúng ta gán thẳng giá trị BigInt bằng với giá trị BigInt của số sau.

Tuy nhiên ở constructor string để tránh tình trạng xuất hiện hàng loạt số 0 đứng trước số (do người dùng nhập sai hoặc cố tình) thì có thêm 1 hàm remove_leading. Hàm này có nhiệm vụ nhận vào 1 chuỗi và 1 số muốn bỏ đi khúc đầu và sẽ trả lại chuỗi mới đã bỏ đi số muốn bỏ. Hàm này sẽ được chạy khúc đầu của constructor string sau đó kiểm tra xem liệu string đó có trống hay không (đối với trường hợp nhập toàn 0 thì sẽ thêm vào lại số 0) rồi mới push_back vào BigInt. Riêng constructor int thì nó sẽ tự động nhận biết và bỏ đi số 0 nên không cần làm gì hết.

- UintN();

Đây là phương thức khởi tạo không đối số, số mặc đinh là 0.

2. Các thuật toán helper function.

Đây là những thuật toán không trực tiếp dc sử dụng ở hàm main nhưng sẽ dc các hàm dc sử dụng ở main cần để chạy được chương trình.

2.1 friend int size(const UintN& a);

Hàm dùng để trả về số lượng chữ số của 1 số. Bằng phương pháp return size() của string BigInt.

2.2 friend bool Null(const UintN& a);

Hàm kiểm tra xem số đó có phải là số 0 hay không.

2.3 friend UintN ModularExponentiation(UintN, UintN, UintN);

Modular Exponentiation là phép toán modulo của lũy thừa vs 1 số. Trong thuật toán Miller Rabin sử dụng phép toán này để kiểm tra tính nguyên tố của số. Tuy nhiên để đáp ứng yêu cầu bài toán chạy phải mượt do đó chúng ta sẽ sử dụng phương pháp Fast Exponentiation (Binary Exponentiation).

Cách hoạt động: Cho 3 số a, n, m với $n \geq 0$ và $0 \leq a < m$, tính $a^n \pmod m$.

Bước 1: Tạo 2 biến. Với biến đầu tiên là biến factor = a, biến thứ 2 res = 1.

Bước 2: Khi biến n > 0, nếu n lẻ thì res = (res * factor) % m, n = n/2, factor = factor^2 % m.

Bước 3: Trả về biến res.

Đối với phương pháp thông thường thì để tính a^n với n là số rất lớn thì chúng ta sẽ phải tính tới n bước. Còn đối với thuật toán này n sẽ được phân tách ra thành:

- 2^k (đối với trường hợp n chẵn)
  Ví dụ: Với a^{128}, thì tương đương với a^{2^7}
- $2^{k1} * 2^{k2} * ....$
  Ví dụ: $a^{205} = a^{2^7} \times a^{2^6} \times a^{2^3} \times a^{2^2} \times a$

2.4 UintN& operator =(const UintN& a);

Phép gán trực tiếp.

2.5 UintN operator%(const UintN& b); UintN& operator%= (const UintN& b);

Phép toán chia lấy dư.

2.6 string remove_leading(string str, char c);

Hàm loại bỏ chữ số c đứng đầu chuỗi, bằng cách so sánh từng phần tử được nhập vào chuỗi. Sau đó sẽ trả về chuỗi mới bằng constructor của string được build sẵn trong C++

3. Thuật toán nhập và xuất (Cin, Cout) trực tiếp.

Cách nhập vào như cách hoạt động của constructor string.

Cách xuất ra sẽ xuất từng phần tử từ cuối tới đầu, tuy nhiên sẽ được biểu diễn dưới dạng (short) vì từng tử đã được trừ đi “0” nên nếu không có short nó sẽ xuất ra kí tự rác không hiển thị được (do đang ở dạng string).

4. Định nghĩa các phép toán số học và so sánh:

4.1 Phép +=:

```
UintN& UintN:: operator+=(const UintN& b)
{
    int t = 0, s, i;
    int n = size((*this)), m = size(b);
    if (m > n)
        (*this).BigInt.append(m - n, 0);
    n = size((*this));
    for (i = 0; i < n; i++)
    {
        if (i < m)
            s = ((*this).BigInt[i] + b.BigInt[i]) + t;
        else
            s = (*this).BigInt[i] + t;
        t = s / 10;
        (*this).BigInt[i] = (s % 10);
    }
    if (t)
        (*this).BigInt.push_back(t);
    return (*this);
}
```

Bước 1: So sánh độ dài đối tượng gọi hàm và chuỗi số để cộng dồn , nếu kích thước chuỗi số để cộng dồn lớn hơn đối tượng gọi hàm thì ta bổ sung thêm các số 0 vào trước số cộng dồn để đạt độ dài 2 chuỗi số bằng nhau.

Bước 2: Cộng từng kí tự chữ số của hai chuỗi từ trái qua phải, phần nhớ được mang theo sang bên phải ở mỗi lần cộng. Sau mỗi lần cộng ở một hàng, ta thêm kí tự cuối của kết quả cộng hàng đó vào bên phải chuỗi kết quả.

Bước 3: Nếu giá trị biến nhớ còn khác 0, viết thêm giá trị biến nhớ vào bên phải chuỗi kết quả.

4.2 Phép -=:

```
UintN& UintN:: operator-=(const UintN& b)
{
    int n = size((*this)), m = size(b);
    int i, t = 0, s;
    for (i = 0; i < n; i++)
    {
        if (i < m)
            s = (*this).BigInt[i] - b.BigInt[i] + t;
        else
            s = (*this).BigInt[i] + t;
        if (s < 0)
            s += 10,
            t = -1;
        else
            t = 0;
        (*this).BigInt[i] = s;
    }
    while (n > 1 && (*this).BigInt[n - 1] == 0)
        (*this).BigInt.pop_back(),
        n--;
    return (*this);
}
```

Bước 1: Lầy từng cặp chữ số trừ đi nhau theo chiều từ trái qua phải, nếu kết quả bị âm thì cộng thêm 10 và nhớ -1 sang hàng phía sau.

Bước 2: Nếu kết quả cuối cùng còn số 0 thì xóa.

4.3 Phép *=:

```
UintN& UintN:: operator*=(const UintN& b)
{
    if (Null((*this)) || Null(b))
    {
        (*this) = UintN();
        return (*this);
    }
    int n = (*this).BigInt.size(), m = b.BigInt.size();
    vector<int> v(n + m, 0);
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            v[i + j] += ((*this).BigInt[i]) *
(b.BigInt[j]);
    n += m;
    (*this).BigInt.resize(v.size());
    for (int s, i = 0, t = 0; i < n; i++)
    {
        s = t + v[i];
        v[i] = s % 10;
        t = s / 10;
        (*this).BigInt[i] = v[i];
    }
    for (int i = n - 1; i >= 1 && !v[i]; i--)
        (*this).BigInt.pop_back();
    return (*this);
}
```

Bước 1: Nếu một trong hai chuỗi số là NULL ta trả về 0.

Bước 2: Tạo 1 vector có size = với size của 2 số và mang toàn bộ số 0. Tiếp theo chạy vòng for từng chữ số để thực hiện phép nhân rồi cộng vào tại vị trí tương ứng

Bước 3: Đẩy từng phần tử trong vector vào sau khi được xử lý vào sau khi được xử lý. Để dễ hình dung thì sau đây có một đoạn code demo.

Chương trình:

```
vector<int> a = { 7,6,5 };
vector<int> b = { 1,9,8 };
vector<int> c(a.size() + b.size(), 0);
for (int i = 0; i < a.size(); i++)
      for (int j = 0; j < b.size(); j++)
            c[i + j] += a[i] * b[j];
vector<int> d;
for (int s, i = 0, t = 0; i < a.size()+b.size(); i++)
{
      s = t + c[i];
      c[i] = s % 10;
      t = s / 10;
      d.push_back(c[i]);
      cout << endl << s << " " << c[i] << " " << t <<
endl;
}
cout << endl;
for (int i = d.size() - 1; i >= 0; i--)
{
      cout << d[i];
}
```

Output:

```
7 7 0

69 9 6

121 1 12

105 5 10

50 0 5

5 5 0

505197
```

4.4 Phép /=:

```
UintN& UintN:: operator/=(const UintN& b)
{
    if ((*this) < b)
    {
        (*this) = UintN();
        return (*this);
    }
    if ((*this) == b)
    {
        (*this) = UintN(1);
        return (*this);
    }
    int i, lgcat = 0, cc;
    int n = size((*this)), m = size(b);
    vector<int> cat(n, 0);
    UintN t;
    for (i = n - 1; t * 10 + (*this).BigInt[i] < b; i--)
    {
        t *= 10;
        t += (*this).BigInt[i];
    }
    for (; i >= 0; i--)
    {
        t = t * 10 + (*this).BigInt[i];
        for (cc = 9; UintN(cc) * b > t; cc--);
        t -= UintN(cc) * b;
        cat[lgcat++] = cc;
    }
    (*this).BigInt.resize(cat.size());
    for (i = 0; i < lgcat; i++)
        (*this).BigInt[i] = cat[lgcat - i - 1];
    (*this).BigInt.resize(lgcat);
    return (*this);
}
```

Bước 1: Nếu đối tượng gọi hàm nhỏ hơn chuỗi số để chia ta trả về 0, nếu bằng ta trả về 1.

Bước 2: Duyệt từ cuối lên đầu ,xây dựng dần số bị chia bằng cách nhân biến trung gian với 10 rồi cộng thêm chữ số hiện tại.

Bước 3: Lấy biến trung gian chia cho chuỗi số chia, viết chữ số cuối của thương thu được vào bên trái kết quả cuối.

4.5 Phép + - * /:

```
UintN UintN:: operator+(const UintN& b)
{
    UintN temp;
    temp = *this;
    temp += b;
    return temp;
}
```

```
UintN UintN:: operator-(const UintN& b)
{
    UintN temp;
    temp = *this;
    temp -= b;
    return temp;
}
```

```
UintN UintN:: operator*(const UintN& b)
{
    UintN temp;
    temp = *this;
    temp *= b;
    return temp;
}
```

```
UintN UintN:: operator/(const UintN& b)
{
    UintN temp;
    temp = *this;
    temp /= b;
    return temp;
}
```

Trong cả 4 phép toán trên đều sử dung chung 1 phương thức:

- Tạo biến temp bằng với biến gọi hàm.
- Temp += | -= | *= | /= với biến còn lại
- Rồi trả về kết quả temp.

4.6 Các phép so sánh:

Các phép toán so sánh chỉ như so sánh thông thường, tuy nhiên trong chương trình chúng ta chỉ cần 2 phép so sánh quan trọng phép so sánh < và phép so sánh == vì các phép còn lại sẽ được lập trình dựa trên 2 phép so sánh đó.

```
bool operator<(const UintN& a, const UintN& b)
{
    int n = size(a), m = size(b);
    if (n != m)
        return n < m;
    while (n--)
        if (a.BigInt[n] != b.BigInt[n])
            return a.BigInt[n] < b.BigInt[n];
    return false;
}
```

Phép so sánh <

Nếu size của a và b khác nhau, kết quả sẽ được trả về dựa trên phép so sánh số học bình thường của biến n và m.

Nếu bằng nhau thì nó sẽ so sánh trực tiếp dựa từng phần tử chạy từ đầu tới cuối, nếu từng phần tử tại các vị trí tương ứng nó khác nhau thì sẽ tiến hành trả về kết quả so sánh dựa trên phép so sánh bình thường.

Nếu duyệt hết nhưng không có vị trí nào khác nhau thì sẽ trả về false.

```
bool operator==(const UintN& a, const UintN& b)
{
    return a.BigInt == b.BigInt;
}
```

Phép so sánh ==

Trả về kết quả so sánh trực tiếp của 2 string.

```
bool operator>(const UintN& a, const UintN& b)
{
    return b < a;
}

bool operator>=(const UintN& a, const UintN& b)
{
    return !(a < b);
}
bool operator<=(const UintN& a, const UintN& b)
{
    return !(a > b);
}
```

Phép so sánh >

Trả về kết quả so sánh của b < a

Phép so sánh >=

Trả về kết quả so sánh ngược của a < b

Vì nếu a = b thì nó cũng sẽ thỏa điều kiện của a < b là false và ngược lại là true.

Phép so sánh <=

Tương tự nhưng ngược lại của >=

```
bool operator!=(const UintN& a, const UintN& b)
{
    return !(a == b);
}
```

Phép so sánh !=

Trả về kết quả ngược của phép so sánh a == b.

5. Thuật toán kiểm tra số nguyên tố:

Sử dụng thuật toán kiểm tra tính nguyên tố Miller Rabin.

```
bool PrimeNumber(UintN& v)
{
  if (v == 1)
    return false;
  else if (v == 0)
    return false;
  else if (v <= 3)
    return true;
  else if (v % 2 == 0)
    return false;

  UintN s = 0;
  UintN t = v - 1;
  UintN b = 2;
  UintN nmin1 = v - 1;
  UintN r, j, smin1;

  while (t % 2 == 0)
  {
    t /= 2;
    s++;
  }
  smin1 = s - 1;
```

```
for (int i = 0; i < 3; i++)
  {
    r = ModularExponentiation(b, t, v);

    if ((r != 1) && (r != nmin1))
    {
      j = 1;
      while ((j <= smin1) && (r != nmin1))
      {
        r = (r * r) % v;
        if (r == 1)
          return false;
        j++;
      }
      if (r != nmin1)
        return false;
    }

    if (b == 2)
      b++;
    else
      b += 2;
  }

  return true;
}
```

Thuật toán sẽ kiểm tra và trả về false với những số bé <= 1 và số chẵn. Sau đó sẽ trả về true với số 2, 3.

Nhưng vì tính chất số lớn của thuật toán, nên chúng ta sẽ chỉ chọn số base (b) = 2, 3, 5 là đã khá đầy đủ và bao quát, tỉ lệ sai cũng thấp hơn khá nhiều so với chọn base (b) = 2, 3 thôi, và tốc độ cũng không quá chậm. Thuật toán ModularExponentiation cũng đã được giải thích ở trên.

Các bước thực hiện:

Bước 1: Tạo các biến như ở trên, sau đó chia biến t tới khi t trở thành số lẻ, và biến s sẽ là số bước đã chia.

Bước 2: Bắt đầu chạy vòng lập để test lần lượt các test base (b) = 2, 3, 5. Với mỗi test base chúng ta sẽ có một biến r tương ứng. Vì nếu r = 1 hoặc = -1 (ở đây chúng ta không có dấu nên sẽ so sánh r với số cần xét – 1) thì thuật toán sẽ đúng nhưng còn các test base khác nên có 1 phép so sánh ở biến r vs 2 biến trên.

Bước 3: Liên tục gán r = dư của phép chia bình phương r với số cần xét. Nếu r = 1 thì sẽ trả về false và lập lại các bước cho đến khi biến j > smin1. Sau khi xong vòng lập nếu r khác với số cần xét – 1 (nmin1) sẽ trả về false.

Bước 4: Sau khi xong vòng lập các test base (b) sẽ trả về true.

Thuật toán còn nhiều hạn chế, vì với số quá lớn sẽ chạy khá chậm, tốc độ chạy sẽ tỉ lệ thuận với số lượng chữ số. Và làm ảnh hưởng trực tiếp đến bài toán tìm số nguyên tố tiếp theo (tùy thuộc vào máy sẽ có tốc độ khác nhau, CPU càng mạnh chạy càng nhanh).

```
UintN NextPrime(UintN a, UintN b)
{
    UintN temp;
    if (a > b)
        temp = a + 1;
    else
        temp = b + 1;
    if (temp % 2 == 0)
        temp++;
    while (!PrimeNumber(temp))
        temp+=2;
    return temp;
}
```

Thuật toán tìm số nguyên tố tiếp theo:

Tạo một biến temp bằng với số lớn hơn trong 2 số nhập vào cộng thêm 1.

Nếu temp là số chẵn thì sẽ cộng tiếp thêm 1. Để số temp luôn luôn là số lẻ, vì bên dưới vòng lập sẽ chạy + 2 (mọi số chẵn sẽ không là số nguyên tố trừ số 2)

Và bắt đầu chạy vòng lập cho tới khi nhận được số nguyên tố sẽ trả về.

C. Kết quả:

```
Please input first number: 868331761881188649551819440127999999
Please input second number: 13232435567890987654334
Summation: 868331761881189972795376229226765433
Subtraction: 86833176188118732630826265102923456665
Multiplication: 114901440906460883942684238791756949945943043159256532345666
Division: 656214615537769
Comparision: First number bigger than Second number
First number could be a prime number
Second number is not a prime number
Next prime number: 868331761881188649551819440128000037
```

- **Input:**
  - **Số đầu:** 868331761881188649551819440127999999
  - **Số thứ hai:** 13232435567890987654334
- **Output:**

- **Tổng:** 868331761881189972795376229226765433 3
- **Hiệu:** 86833176188118732630826265102923456 65
- **Tích:** 1149014409064608839426842387917569499459430431592565323456 66
- **Thương:** 656214615537769
- **So sánh hai số: Số thứ nhất lớn hơn số thứ hai**
- **Kiểm tra số nguyên tố:**
  - Số thứ nhất là số nguyên tố
  - Số thứ hai không là số nguyên tố
- **Số nguyên tố lớn hơn hai số trên là:** 868331761881188649551819440128000003 7

D. Tài liệu tham khảo:

Miller Rabin test:

- en.wikipedia.org/wiki/Miller-Rabin_primality_test
- www.geeksforgeeks.org/primality-test-set-3-miller-rabin/
- www.codeproject.com/Articles/60108/BigInteger-Library-2

Modular Exponentiation:

- homepages.math.uic.edu/~leon/cs-mcs401-s08/handouts/fastexp.pdf
- vi.wikipedia.org/wiki/Modular_exponentiation

BigInt:

- www.geeksforgeeks.org/bigint-big-integers-in-c-with-example/
- github.com/WHU-Cryptography/BigInteger

__End__