项目报告

<table>
<tr><td>项目名称</td><td colspan="4">基于 C++的任意精度科学计算器</td></tr>
<tr><td>项目简介</td><td colspan="4">本项目实现了一个基于 C++的任意精度科学计算器，包括对整数和浮点数的运算，所述整数是任意精度的，可表达的值的范围仅取决于可用的内存[1]；所述运算包括四则运算、乘方、阶乘、取余、条件判断等，还包括这些运算的任意复合；同时支持用户自定义变量等，功能强大，效率优良.</td></tr>
<tr><td rowspan="2">小组成员</td><td>姓名</td><td>班级</td><td colspan="2">联系方式（电话及邮箱）</td></tr>
<tr><td>倪可塑</td><td>软 1712</td><td>15641135117</td><td>nks5117@mail.dlut.edu.cn</td></tr>
<tr><td>分享地址</td><td colspan="4">https://pan.baidu.com/s/1dJVg0CHI0a022aa0N99IeA 密码: 69aj</td></tr>
</table>

评分细则及标准

考察项目	总分	评分细则	得分
项目创意	20分	立意新颖，创意独特	
问题规模	20分	程序具有一定规模，能解决实际问题	
技术难度	20分	图形界面友好，功能实用，注重效率	
实现程度	20分	系统设计与实现的吻合程度	
报告质量	10分	文档与图文规范，清晰展示设计思路	
视频展示	10分	生动全面展示系统的实现效果	
总分	100分		

---

[1] 理论上可表示的整数最大值是$(2^{30})^{\mathrm{SIZE_MAX}} - 1$，其中 `SIZE_MAX` 是 `size_t` 类型的最大值. 但由于性能的限制，在可接受的时间内能处理的数据是有限的. 测试时计算过的最大整数为梅森数 M_{44497}，即 $2^{44497} - 1$，这是第一个超过一万位的梅森素数，共有 13,395 位.

目　录

1 开发背景

在平时的计算中，常常有高精度的大数计算的需求，但大部分常用的计算器的精度不高．以 Windows 10 自带的计算器为例，最多仅能保留 32 位有效数字，计算范围最高至 1e10000，而 C 和 C++所提供的数据类型同样只能表示有限范围的数，以整型为例，一个 32 位的无符号整数最大仅能表示 4,294,967,295，即便是 64 位的无符号整数也只能表示到 18,446,744,073,709,551,615，进行大数计算时常常会捉襟见肘．为此，设计了一个基于 C++的任意精度的科学计算器．

2 需求分析

大整数类的功能需求：

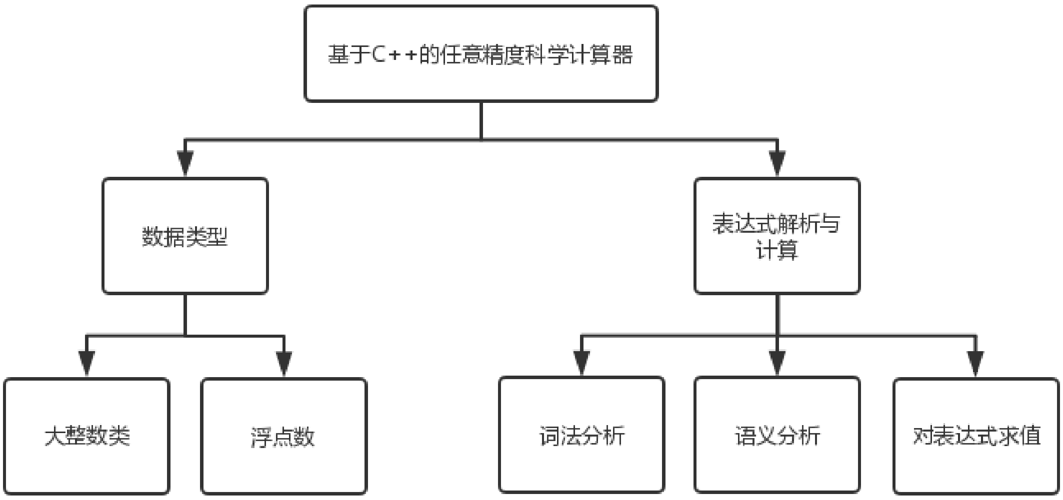
- 能表示任意精度的整数；
- 能进行加减乘除等运算；
- 能与系统内置整数类型相互转化；
- 能与给定进制字符串相互转化；
- 能进行赋值等操作；
- 通过重载各运算符，使其使用方法与内置整数类型无异；
- 能高效地完成上述各操作．

表达式分析和计算模块的功能需求：

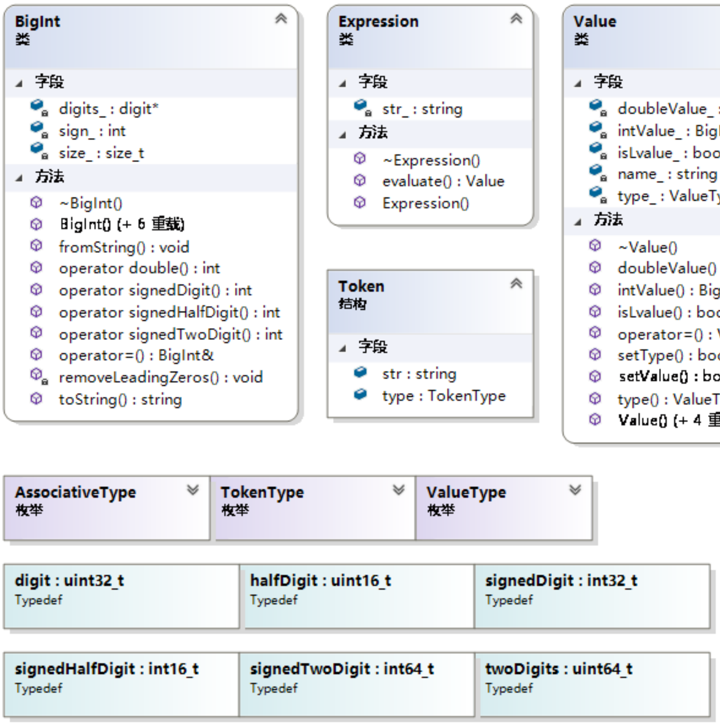
- 能对输入的表达式进行词法分析，以识别各运算符、标识符和字面量；
- 能根据运算符的优先级和结合性，判断表达式的计算顺序；
- 能根据对表达式的分析结果，对表达式求值并输出结果；
- 能识别非法的表达式，并给出相应的错误提示；
- 内置如e、π等多个可直接使用的科学常数；
- 能存储用户自定义的变量；
- 能识别不同类型的值，并根据需要进行类型转换．

3 总体设计

功能模块图如图：


基于C++的任意精度科学计算器
数据类型
表达式解析与计算
大整数类
浮点数
词法分析
语义分析
对表达式求值

系统类图如图：


BigInt
类
字段
digits_ : digit*
sign_ : int
size_ : size_t
方法
~BigInt()
BigInt() (+ 6 重载)
fromString() : void
operator double() : int
operator signedDigit() : int
operator signedHalfDigit() : int
operator signedTwoDigit() : int
operator=() : BigInt&
removeLeadingZeros() : void
toString() : string
Expression
类
字段
str_ : string
方法
~Expression()
evaluate() : Value
Expression()
Token
结构
字段
str : string
type : TokenType
Value
类
字段
doubleValue_ : double*
intValue_ : BigInt*
isLvalue_ : bool
name_ : string
type_ : ValueType
方法
~Value()
doubleValue() : double
intValue() : BigInt
isLvalue() : bool
operator=() : Value&
setType() : bool
setValue() : bool (+ 1 重载)
type() : ValueType
Value() (+ 4 重载)
Variable
类
字段
doubleValue_ : double
intValue_ : BigInt
name_ : string
type_ : ValueType
方法
doubleValue() : double*
intValue() : BigInt*
name() : string
setType() : bool
type() : ValueType
Variable()
AssociativeType
枚举
TokenType
枚举
ValueType
枚举
digit : uint32_t
Typedef
halfDigit : uint16_t
Typedef
signedDigit : int32_t
Typedef
signedHalfDigit : int16_t
Typedef
signedTwoDigit : int64_t
Typedef
twoDigits : uint64_t
Typedef

4 详细设计与实现

4.1 BigInt 类的设计和实现

BigInt 类为大整数类，为在实现存储任意大的整数的功能的同时兼顾效率，采用了2^{30}进制来存储数据，即每“位”数字的范围是$0\sim 2^{30}-1$. Bigint 类使用一个数组维护各位上的数字，另用两个整数分别表示数字的位数和符号，如下：

```
digit * digits_; // pointer to array of digits, nullptr for zero
size_t size_; // digits in array, 0 for zero
int sign_; // 0 for positive or zero, 1 for negative number
```

其中，digit 为 uint32_t 的类型别名.

经此定义，一个 BigInt 类的对象 n 所表示的数字的绝对值为

$$\sum_{i=0}^{\mathrm{n.size\_}-1} \mathrm{n.digits\_}[i]\times(2^{30})^{i}\,.$$

为 BigInt 类定义了各种方法，包括构造与析构、赋值、类型转换、四则运算、格式化输入和输出、关系运算等. 每个方法都经过精心设计，以达到更高的效率. 下面选取部分方法详细介绍其设计和实现.

4.1.1 适用于 BigInt 的四则运算的设计和实现

BigInt 类重载了运算符+，-，*，/以实现大整数的四则运算：

```
friend BigInt operator-(const BigInt &l);
friend BigInt operator+(const BigInt &l, const BigInt &r);
friend BigInt operator-(const BigInt &l, const BigInt &r);
friend BigInt operator*(const BigInt &l, const BigInt &r);
friend BigInt operator/(const BigInt &l, const BigInt &r);
```

令这些函数为 BigInt 的友元函数主要是为了左操作数可以为内置的数据类型，以实现诸如 int 和 BigInt 类型的变量互相加减的功能.

加、减、乘法的设计与列竖式计算无异，以加法为例，函数自低位起逐步取出两位数字相加，得到和和进位，再将进位加到下一位上，如此循环往复，部分代码如下：

```
BigInt operator+(const BigInt & l, const BigInt & r)
{
// ...
  for (size_t i = 0; i < r.size_; i++) {
    digits[i] = fullAdder(l.digits_[i], r.digits_[i],
                 digits[i], &digits[i + 1]);
  }
  for (size_t i = r.size_; i < l.size_; i++) {
    digits[i] = fullAdder(l.digits_[i], static_cast<digit>(0),
                 digits[i], &digits[i + 1]);
  }
```

```
// ...
}


digit fullAdder(digit n1, digit n2, digit carryIn, digit *carryOut)
{
  digit s = n1 + n2 + carryIn;
  *carryOut = s >> SHIFT;
  s &= MASK;
  return s;
}
```

其中，`fullAdder()`是全加器，返回 `n1`，`n2`，`carryIn` 的和，并将进位存到 `carryOut` 所指的变量.

减法和乘法类似，不再赘述.

`BigInt` 的除法是程序的难点. 最初考虑使用被除数反复减去除数来求商，但这样做的效率太低. 优化方案之一是逐步放大再减少每次相减的倍数，同时将除以二优化为移位操作：

```
BigInt operator/(const BigInt & l, const BigInt & r)
{
  // some conditional statements
  // ...

  if (r == TWO) {
    BigInt res = l;
    for (size_t i = 0; i < res.size_; i++) {
      res.digits_[i] >>= 1;
      if (i + 1 < res.size_) {
        res.digits_[i] |= res.digits_[i + 1] & static_cast<digit>(1);
      }
    }
    res.removeLeadingZeros();
    return res;
  }

  BigInt res = ZERO, n = abs(l), d = abs(r), step = ONE;
  while (1) {
    if (n - step * d >= ZERO) {
      n = n - step * d;
      res = res + ONE * step;
      step = step * TWO;
    }
    else {
      if (step == ONE) {
        break;
      }
      else {
        step /= TWO;
      }
```

```
    }
  }
  if (l.sign_ != r.sign_) {
    res = -res;
  }
  return res;
}
```

经测试，这种方案相比于每次只减去一倍的除数的方法有了很大的进步，大幅度提高了效率，但事实上，如后文所述，照此算法，程序运行时常常把绝大部分时间都花在了除法运算上，本算法仍需要改进，关于除法算法的改进设计见第 6 节.

4.1.2 BigInt 类的格式化输入输出方法的设计和实现

BigInt 类内部将数表示为2^{30}进制，故需要专门的函数将其转换为十进制字符串才能输出，同样当从用户获取输入时，也需要进行转换才能够存储到 BigInt 类内. 为此设计了 fromString()方法和 toString()方法.

4.1.2.1 fromString()方法的设计和实现

BigInt 类的 fromString()方法能将十进制字符串转化为 BigInt 类的对象.首先将长度为 n 的十进制字符串转换为一个 int 数组 a[n]，其中每一个 int 表示十进制的一位数字，低位在前高位在后，则该字符串所代表的数字即为

$$\sum_{i=0}^{n-1} a_i \times 10^i\ ,$$

计算这个和，并将其存储到 BigInt 对象中即可.

观察这个和式，对其进行如下转化：

$$\begin{aligned}
\sum_{i=0}^{n-1} a_i \times 10^i &= a_0 + a_1 \times 10 + a_2 \times 10^2 + \cdots + a_{n-2} \times 10^{n-2} + a_{n-1} \times 10^{n-1} \\
&= a_{n-1} \times 10^{n-1} + a_{n-2} \times 10^{n-2} + \cdots + a_2 \times 10^2 + a_1 \times 10 + a_0 \\
&= (a_{n-1} \times 10^{n-2} + a_{n-2} \times 10^{n-3} + \cdots + a_2 \times 10 + a_1) \times 10 + a_0 \\
&= ((a_{n-1} \times 10^{n-3} + a_{n-2} \times 10^{n-4} + \cdots + a_2) \times 10 + a_1) \times 10 + a_0 \\
&\vdots \\
&= \big((\ldots(a_{n-1} \times 10 + a_{n-2}) \times 10 + \cdots + a_2) \times 10 + a_1\big) \times 10 + a_0.
\end{aligned}$$

这便是所谓的秦九韶算法（西方称霍纳规则），fromString()方法使用该算法计算这个多项式的值，能大大减少所需要计算的乘法次数. 具体地讲，以一般方法计算一个n次多项式，需要进行$\frac{1}{2} \times (n^2 + n)$次乘法和$n$次加法，而采用秦九韶算法只需要$n$次乘法和$n$次加法，从而大幅提高了效率.

此外，fromString()方法首先使用 strtoll()函数尝试将字符串转换为一个 long long，如果成功，则不需要使用我们编写的方法进行处理，考虑到在实际使用过程中，用户输入的数字很可能大部分都在 long long 可以表示的范围内，这样做也可以提高效率.

fromString()方法和使用秦九韶算法对多项式求值的 polynomialEvaluate()函数代码如下：

```
void BigInt::fromString(const string &s)
{
  errno = 0;
  long long num = strtoll(s.c_str(), nullptr, 10);
  if (errno != ERANGE) {
    delete[] digits_;
    digits_ = nullptr;
    *this = BigInt(num);
    return;
  }

  vector<int> a;
  for (auto iter = s.crbegin(); iter!=s.crend(); ++iter) {
    a.push_back(*iter - '0');
  }
  if (a.back() + '0' == '-' - '0' + '0') {
    a.pop_back();
  }
  delete[] digits_;
  digits_ = nullptr;
  *this = polynomialEvaluate(a, 10);
  return;
}

// use Qin Jiushao's Algorithm
BigInt polynomialEvaluate(const std::vector<int>& a, int n)
{
  BigInt r, x(n);
  for (auto iter = a.crbegin(); iter != a.crend(); ++iter) {
    r = BigInt(*iter) + x * r;
  }
  return r;
}
```

经测试，fromString()方法和 polynomialEvaluate()函数均可良好地工作.

4.1.2.2 toString()方法的设计和实现

BigInt 类的 toString()方法能将 BigInt 内存储的数字转换为对应的十进制字符串，原理为反复对十取余并除以十. 鉴于将进行取余运算时需要先进行除法运算，

一个常数级别的优化是将前述的除法运算的中间结果存储起来，以节省一半的除法计算量，代码如下：

```
std::string BigInt::toString() const
{
  if (*this == ZERO) {
    return "0";
  }
  std::string str;
  auto cpy = abs(*this);
  while (cpy > ZERO) {
    auto tem = cpy / TEN;
    str += static_cast<char>('0' + static_cast<int>(cpy - tem * TEN));
    cpy = tem;
  }
  if (this->sign_ == 1) {
    str += '-';
  }
  return std::string(str.rbegin(), str.rend());
}
```

需要注意的是，`toString()`方法进行了大量的除法运算，尤其是当数很大时，用十除的操作本身就较为耗费时间，这也导致了程序在输出结果时所耗时间很长．举例来讲，我们曾于 2018 年 8 月 14 日在 beta 版的测试系统上计算$2^{44497}-1$，计算这个幂的过程本身仅消耗 0.15s，然而将结果转化为十进制输出却消耗了 12,193s 的时间，这是难以接受的．在 2018 年 9 月 6 日完成的 1.5.0 版程序中，我们对除法算法进行了改进，大幅提升了效率，具体内容见第 6 节．

4.2　表达式的解析和处理模块的设计和实现

为处理输入的表达式，设计了一个表达式类 `Expression`，`Expression` 类的设计较为简单，仅使用一个 `string` 存储字符串形式的表达式，使用 `evaluate()`方法对表达式求值：

```
class Expression {
private:
  std::string str_;
public:
  Expression(const std::string &str) :str_(str) {};
  ~Expression() {};
  Value evaluate();
};
```

`evaluate()`方法是解析并处理表达式的核心，对输入的表达式，`evaluate()`方法首先进行词法分析，识别各标识符、字面量和运算符，将字符串转化为记号（token）串．此后对记号串进行语义分析，根据各运算符的优先级和结合性确定计算顺序，接着对表达式求值，并根据需要输出结果．

4.2.1 词法分析过程

最初的表达式是一个字符串，这个字符串中的每个字符，有的是字面量或标识符，有的是运算符，还有的是上述三种的一部分．字面量、标识符和运算符统称为记号（token），词法分析过程便是将字符串转换为记号串的过程．

为此，首先设计了一个结构 `Token`，这样每个记号就都可用一个 `Token` 类型的变量表示，方便统一处理：

```
struct Token {
  std::string str;
  TokenType type;
};
```

其中，`str` 存储记号的字符串形式（如`"+"`，`"123"`，`"abc"`等），`type` 存储记号的类型．`TokenType` 是枚举类型，为每种记号指定了一个整数．

将字符串转换为记号串的过程是通过 `getTokenStream()` 函数实现的：

```
vector<Token> getTokenStream(const string & s)
{
  vector<Token> r;
  string t = removeBlank(s);
  for (size_t i = 0; i < t.size();) {
    r.push_back(getToken(t, i));
    i += r.back().str.size();
  }
  return r;
}
```

`getTokenStream()` 函数首先移除字符串 `s` 的空格，接着反复从字符串中读取并存储下一个记号，以完成词法分析过程．

`getToken()` 函数原型如下：

```
Token getToken(const std::string & s, size_t i);
```

其中，`s` 是字符串，`getToken()` 识别并返回以 `s[i]` 为起始的一个记号．

4.2.2 语义分析过程

由于程序一次只处理用户输入的一行表达式，故不需要语法分析过程，直接对表达式进行语义分析即可．

语义分析主要解析表达式的含义，并检查用户输入的表达式是否有语法上的错误，比如括号不匹配、操作数类型不匹配、运算符过多等．部分错误直到真正对表达式开始求值时才能发现，我们的语义分析过程主要检查括号不匹配的错误，同时根据运算符的优先级和结合性确定表达式的求值顺序．

为此，语义分析过程使用 `infixToPostfix()`函数将记号串由中缀记法转换为后缀记法，即操作数在前，运算符在后的形式，如"3 + 4 × (2 - 1)"将被转换成"3 4 2 1 - × +"．这种记法不需要括号来注明运算顺序，便于编写求值算法：

```
vector<Token> infixToPostfix(const vector<Token>& a);
```

`infixToPostfix()`函数使用调度场算法（Shunting Yard Algorithm）完成转换，该算法的流程图见附录 B.

4.2.3　表达式的求值过程

一个表达式的值，可能是一个整数（`BigInt`），也可能是一个浮点数（`double`）；可能是左值（如用户自定义的变量等），也可能是右值（如字面量等）．故设计了一个类 `Value` 来作为值的统一表示：

```
enum ValueType {
  V_BIGINT, V_DOUBLE, V_NA
};

class Value {
private:
  ValueType type_;
  BigInt *intValue_;
  double *doubleValue_;
  bool isLvalue_;
public:
  Value(ValueType type = V_NA, BigInt *intValue = nullptr,
        double *doubleValue = nullptr, bool isLvalue = false);
  Value(const BigInt &n);
  Value(const double &n);
  Value(const bool &n);
  Value(const Value &n);
  ~Value();
  Value &operator=(const Value &n);
  ValueType type() const;
  BigInt intValue() const;
  double doubleValue() const;
  bool isLvalue() const;
  bool setValue(const BigInt &n);
  bool setValue(const double &n);
};

Value toValue(const Token &n);
```

此外，对于各种运算符，都应有相应的函数，这些函数的参数和返回值的类型都是 `Value`：

```
Value factorial(const Value &n);
Value power(const Value &lhs, const Value &rhs);
Value assign(Value &lhs, const Value &rhs);
Value plus(const Value &lhs, const Value &rhs);
```

```
Value comma(const Value &lhs, const Value &rhs);
// ...
```

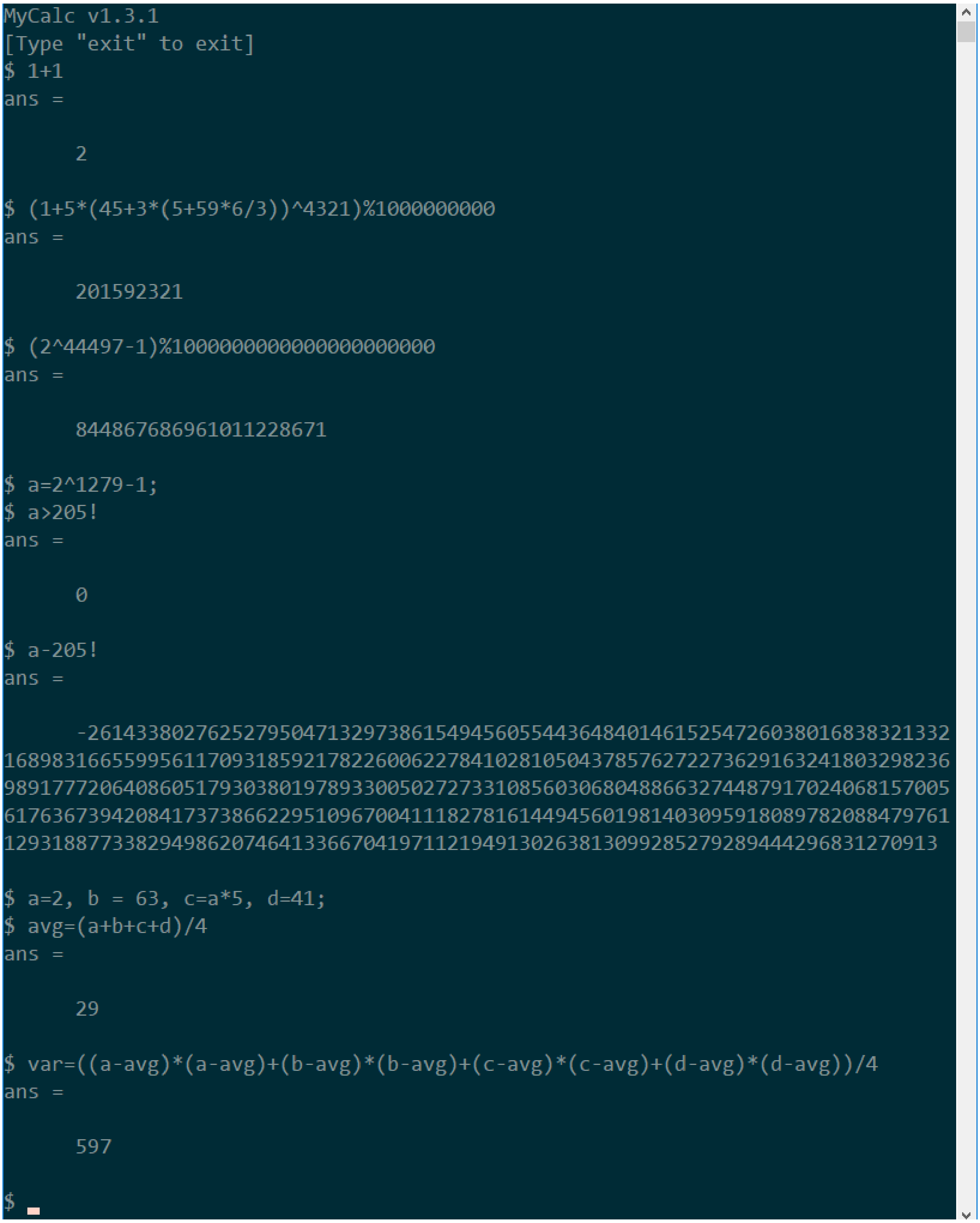
做好这些准备后，就可以开始编写对表达式求值的代码了．对已经转换为后缀记法的表达式求值较为简单，伪代码如下：

```
while 记号串非空
  读入下一个记号，记为X
  if X是一个操作数
    X入栈
  else // X是一个运算符
    确定X需要n个操作数
    if 栈中操作数不足n个
      （出错）用户没有输入足够的操作数
    else
      n个操作数出栈
      计算运算符
      将结果入栈
if 栈内只有一个值
  将该值作为结果返回
else // 栈内多于一个值
  （出错）用户输入了过多的操作数
```

以上就是表达式求值的全过程．

5 部分程序界面展示

此处展示的是于 2018 年 8 月 24 日编译的 1.3.1 版程序，运行于 Windows 系统.



```
MyCalc v1.3.1
[Type "exit" to exit]
$ 1+1
ans =

      2

$ (1+5*(45+3*(5+59*6/3))^4321)%1000000000
ans =

      201592321

$ (2^44497-1)%1000000000000000000000
ans =

      844867686961011228671

$ a=2^1279-1;
$ a>205!
ans =

      0

$ a-205!
ans =

      -261433802762527950471329738615494560554436484014615254726038016838321332
1689831665599561170931859217822600622784102810504378576272273629163241803298236
9891777206408605179303801978933005027273310856030680488663274487917024068157005
6176367394208417373866229510967004111827816144945601981403095918089782088479761
129318877338294986207464133667041971121949130263813099285279289444296831270913

$ a=2, b = 63, c=a*5, d=41;
$ avg=(a+b+c+d)/4
ans =

      29

$ var=((a-avg)*(a-avg)+(b-avg)*(b-avg)+(c-avg)*(c-avg)+(d-avg)*(d-avg))/4
ans =

      597

$
```

6 在 1.5.0 版中对除法算法的改进

首先考虑我们常见的手算除法，即列竖式除法．在我们列竖式计算多位数之间的除法的时候，我们通过把被除数“截短”，一次给出商的一位数，称之为“试商”．但试商的过程往往是直觉猜测，需要将其严格地算法化．鉴于试商过程只得到一位数的商，因此只需考虑商为一位数的除法．

设$a=(a_n a_{n-1}\dots a_1 a_0)_B$，$b=(b_{n-1}\dots b_1 b_0)_B$分别为$n+1$位和$n$位的$B$进制正整数，且$a$除以$b$的商为一位数，即$0\le\left\lfloor\frac{a}{b}\right\rfloor\le B-1$，则有

$$\left\lfloor\frac{a}{b}\right\rfloor\le\min\left\{\left\lfloor\frac{a_nB+a_{n-1}}{b_{n-1}}\right\rfloor,B-1\right\}.$$

证明：由于

$$\begin{aligned}\left\lfloor\frac{a_nB+a_{n-1}}{b_{n-1}}\right\rfloor+1&\ge\frac{a_nB+a_{n-1}+1}{b_{n-1}}\\&=\frac{a_nB^n+(a_{n-1}+1)B^{n-1}}{b_{n-1}B^{n-1}}\\&>\frac{a}{b},\end{aligned}$$

且$\frac{a}{b}\ge\left\lfloor\frac{a}{b}\right\rfloor$，故

$$\left\lfloor\frac{a}{b}\right\rfloor<\left\lfloor\frac{a_nB+a_{n-1}}{b_{n-1}}\right\rfloor+1.$$

考虑到不等式左右两边都是整数，故

$$\left\lfloor\frac{a}{b}\right\rfloor\le\left\lfloor\frac{a_nB+a_{n-1}}{b_{n-1}}\right\rfloor.$$

□

上述定理意味着$\left\lfloor\frac{a_nB+a_{n-1}}{b_{n-1}}\right\rfloor$是商$\left\lfloor\frac{a}{b}\right\rfloor$的一个上界．事实上，接下来将看到，$\left\lfloor\frac{a_nB+a_{n-1}}{b_{n-1}}\right\rfloor$还是一个很好的上界．

为讨论方便我们记

$$q=\left\lfloor\frac{a}{b}\right\rfloor,\quad\hat{q}=\min\left\{\left\lfloor\frac{a_nB+a_{n-1}}{b_{n-1}}\right\rfloor,B-1\right\},$$

则当$b_{n-1}\ge\left\lfloor\frac{B}{2}\right\rfloor$时，可以证明

$$\hat{q}-2\le q\le\hat{q},$$

即$\hat{q}$比q至多大 2．因此，$\hat{q}$可以作为对q的一个很好的估计．

而对于$b_{n-1} \leq \left[\frac{B}{2}\right]$的情况，只需对除数和被除数同时乘以$\left[\frac{B}{b_{n-1}+1}\right]$即可.

至此，商为一位数的除法问题，已经得到了解决，除法算法中其余部分比较简单，在此从略. 我们于 2018 年 9 月 6 日在采用了改进后的除法算法的 1.5.0 版程序进行了测试，测试项目与使用改进前的算法的测试项目一样，均测试计算了梅森素数$2^{44497}-1$，在当时计算并输出这个幂的过程耗时 12,193s，而现在计算和输出的总时间可以保持在 60s 以内，这意味着程序的效率有了革命性的提升，本系统的实用性有了质的飞跃.

7 设计总结

7.1 项目总体完成情况

本项目创意产生自小学期，立意新颖，创意独特；问题规模大，自立项至本报告的完成共历时约一个月，以 1.5.0 版为例，共有超过 1,800 行代码，实现了强大的任意精度科学计算的功能，程序具有一定规模，能解决实际问题；技术难度高，界面经精心设计，格式美观，程序功能强大，采取多种方法对算法进行优化，能在短时间内处理上万位的大数，界面友好，功能实用，注重效率；实现程度好，基本实现了系统设计时的各个需求，系统设计与实现的吻合程度高；本报告逻辑清晰，图表规范，排版美观，语言简练，文档与图文规范，清晰展示设计思路；随附视频经精心剪辑，生动全面展示了系统的实现效果.

7.2 技术难点及其解决

本项目的技术难点之一是大整数类的实现. 为此查阅了各种资料，阅读了包括 CPython 等在内的众多开源项目对任意精度整数的实现代码，学习了包括秦九韶算法在内的各种算法以提升效率.

此外，对内存的管理也是难点，首先是要小心处理分配的内存，确保其在合适的地方释放，否则极易造成内存泄漏；另一方面更要时刻注意指针是否有效，避免访问或释放已被释放的内存. 在项目的编写过程中曾在内存管理的问题上导致很多 bug，举例来讲，在测试刚编写完毕的 `fromString()`方法时，发现其只能正常工作一次，当第二次调用该方法就会出现问题. 通过漫长的单步调试和查错的过程后，发现问题出在当 `digits_`在 `fromString()`内被释放后，其会在下一行的赋值操作时再一次被释放. 由于测试代码中第一次调用 `fromString()`时该 `BigInt` 对象是使用默认构造函数初始化为零的，其 `digits_`为 `nullptr`，就导致释放两次没有问题，而第二次调用

`fromString()`时 `digits_`已经不是空指针了，于是当赋值操作尝试释放一个已经释放的指针就会出错.

7.3 不足及下阶段目标

对于错误处理，现阶段的程序表现不好，当遇到诸如表达式括号不匹配等的错误时，程序能给出正确的提示，但不能继续运行，计划下一步尝试使用异常机制完善错误提示和处理功能.

此外，虽然经过力求完善的设计，但由于时间紧，程序仍存在一些待修复的 bug.

参考文献

[1] 王东明, 夏壁灿, 李子明. 计算机代数[M]. 清华大学出版社, 2007.
[2] 李超. 计算机代数系统的数学原理[M]. 清华大学出版社, 2010.
[3] 陈玉福. 计算机代数讲义[M]. 高等教育出版社, 2009.
[4] AlfredV.Aho. 编译原理[M]. 机械工业出版社, 2009.
[5] Calc: http://www.isthe.com/chongo/tech/comp/calc/
[6] Cpython: https://www.python.org/downloads/source/

附录 A：梅森数 M_{44497}的值

形如2^n-1的数被称为梅森数，记为M_n．如果一个梅森数为素数，那么称它为梅森素数．当n为合数时，M_n一定为合数；但当n为素数时，M_n不一定均为素数．对于梅森素数的探寻难度极大，人类目前为止仅发现 50 个梅森素数，第五十个梅森素数$M_{77232917}$是于 2017 年 12 月 26 日被发现的，这个超大素数共有 23,249,425 位数，如果全部打印出来需要一本 700 多页的书．

在编写本项目的过程中，我们也使用本系统计算了一些梅森素数的值，以测试系统能否给出正确的结果，以及测试系统的计算效率．在测试时计算过的最大的数为M_{44497}，这是第 27 个梅森素数，也是第一个位数超过一万的梅森素数，共有 13,395 位，最初是在 1979 年 4 月由美国克雷公司计算机专家史洛温斯使用 Cray-1 型超级计算机花费一个半月的时间发现的，在 3 年内为当时已知的最大素数．

除了M_{44497}，我们还使用了较小的梅森素数如M_{1279}（共 386 位）、M_{4423}（共 1,332 位）、M_{11213}（共 3,376 位）和M_{21701}（共 6,533 位）等进行了测试，1.3.1 版的测试程序计算并输出这些结果分别耗时约 3s、40s、360s 和 1830s．在 1.5.0 版的程序上，这些数都能在 15 秒内计算并输出完毕．这些测试的成功意味着我们的程序能够在可接受的时间内处理大至上万位的大整数，在现实中具有很好的实用性．

作为证明，下面给出使用本项目计算得到的M_{44497}的结果：

```
85450982430363380319330070531840303650990159130402105834326925828229006478216763585620050001445764586148131529525
32236749383405022256414367942948362866139336719228387223492861850544537994849197028140662986824128530225945827025
32253637046393573819102339382603546705057592743425373988510067594258489091882652816984230481339231089370597522429
65796222102362538978683847622256270093757284949365677830970984890261155981748182164856299141430118685263110991962
80145068918799382699198763750414760545704803938178020676425738029343208040046892120612631349371911469392179678639
13998340450302066353316060446899821960016965149455247451256866120416455837785237889442736716622863694799063147972
42191513728118522363948553093908070070396563052428480369632952259456017856523100642835591426445580953590482718456
78819707528150686464125352129464062934464045775197479006845366099020725849926179813110483653182493320032776957273
34239248756022919323029881117042744212106822448167090688816268058419617823050362806871644195094989164027842731200
31353652875289877862567447354927840677318519712072487344098119928604499876105419523137395694918998919187655622934
47301150450914622261031686188340973446099971566101332288314059541186983973608678369384512670266266590918439881807
73279436086490179650284300557692234183476442984646309183072757232974100222299116389491938691664026821610953870032
13374949182998363445625100201876791107734199898240654211719244296115963918783759416790689681330645143318895098600
06022559477014454710514711241248484336829214745749720575068214150684678501124818809203316307200136273429717962200
98146625253125901337980386234817629898710708176158298693133149214166399431656277231164600237527367266776679870865
78579783023611727781835105986580158837678651090390043464286182430736866084352550914618011348921597025307313453693
23345932633434286573386273784921786777977849042729418318134604348292381228571121858661612351733798961027307677614
23558393178166224715620952758653591232586065130931612235961024495955052576313785415587572218003004463096123010675
46042958623358277478437574661406234308742888737665542006323788806086319894800406043577072710325401655167198097967
09244397686970277891575974508206254271409670848762533304329070823626162080583431773451380188128330299336602004181
78113067998719426282674176029537056946676010743381949863248322018120347951203883971813116798180480041145250934319
94019521237699707002040347324061842751239556599401763926764238759035227062340384140939877819796505281888049412267
98796374509829298725598806238845146621966945677440279401300381799201825952482200299605458360684378365372204211518
65495276144874163510998224619086027469138960543563366461310280362835876324827709909151758836738659026346983354907
15853374117200855734728919183411887434450721170639972848828016471994191854859699641006407411283551674645401444064
01476999667920016366490784920494972999920370107225027671236146745702083250857862039789132460130885006400527804037
83197363796495378941014322353383867757116808444859236033884177503488694746360263081077518495278672673007687714633
81467507408140392770844905258236224194369720628723658720856229364135609456618525515788779451753217328311183300030
20256130052134054822528781490702652054403735454269101616113411855590509889060201376141094819658477367963921125213
75419675323000779270489340923806543050956827559032136728828425649916323160041509893239953929650589544864646169009
56812597012933138966840997303390729744599907618665109576531031106104368289656901620969453699613626647019156940803
33235416940573867387352314653914177158575314885778972508517929099101694707248532614278499405493343058964755928239
92656458347993884214575074296814676874464971197215649974506724058916041508916049169707455165438683627351824021232
17384765017500956924684458533976433144403987971559593528051564089748125704788471526363562183131225073309524474147
33794129949289372465044718733600934229459016696886367330528722151883926683438705183322173579806860706093205544809
10523795553942258900948600643831553785127382196992543338144141784198002454900271443091682756574616685753240566083
38450591198664472083873879654075449787482408931435442776856039012582190697433137791558987483309683222036482621463
75324674432572193253735425956321688681968456623145352577457405406900887212589323757269267807660429605225702623919
44742291827500524719618152204706641289353527039250445951674387140964202979771525691667923635515333128240868122085
66645140560583495647240658229819681596199632316156585051694167984580980370638874320600508249828790882844714246861
```

```
26989425137754171119488405175107174441584153114258722355049122929009315302945400125292606252239475841867870910652
70997026119386422407099057687522692243445528559804365149519299366360636524787812055597948447738733157386435933019
18169152712330014243916513263922065833816688260071700594605706810950819028916907780058985503093984941430560993832
44210656853427873351226289438932247375588009552912301623544737036682694182468764500514590987299496864276169913103
46258467807714200493294634954267280195055460982624792493147249023081291160699372476860084646890874325382990275414
74948297683037741993725346735763886591350818028660540976824064090487860868728469057809708418994396699120357632716
29818344737569466799027996015564032270359977334084630518828780406486059034899435009900904288172821365622445431382
27751934870681977529470093118307377797864388134866337375506717466865840299561391687595163814871073257516237138953
0847397413210465125198987830559851375629099868628228214186979092186183758247762435961296355767993599936934408592
34807176986995734919472669172383161365946975157488841328443968146718472737197448793578528795477262243957780187918
17246412295326838150979939296919653496539270225341826037714755592301500206279366160321101433476538168211877778289
23638249283950781630905410199680788196599025114406053158326966733250363984727128933120543136606406953088382089986
87067094368364818487896907646730238626957405970453586069579283244568157661300058463854019000943953536847808957763
36253796523782075679613024968339290175813865387793255361742847784029992110588949340556741643663846869775018478118
39444336525079329981750624460353640851581050445608565668118348476690963250411915910185778621308437037577825337199
39505682706083687202559200961074842596957850027992441852518700947757293339959334990762573796010750259397593822826
14484710286255549932745375952696109293041118929794123755168354275719367365141154223682577274756986939801666353878
98271241967204789582795142156360130421935293920161728194513071144459349891050007835720277557869870841775865047673
24443216153895863371831780523631045770366679769817943797751372549746991255216638059996525455419360357233898260568
12617646779754277945047644116609828329736790436393517413288251558898045514749445906423194323706306414119366629747
90748128737394083284291396945123318691486960674329191640959101959263202529962819761138425326492474854071795500614
80112131963309733381733583507313303588011766870558206670073162064851742949807196815806110075763663155668179886766
76629818069714322226851713501056676703693691882424689948234282990975689672226518343470550974174195976295776513379
48434267244002876575107127736128123401182445186498711851190026802362066092553781959901712322324019522203702716155
97853719682023122892223797160232801997625032033458423644758731654437119961216372186560140771099382876344626419112
37643661498730924997478477798307306107972975765021697764315038032227248049476011713075846328424181343680728329020
29953272441846878501459877358391275097364790233445605310334723695667960648217214526430715669855992628604717850374
19495135404480386049800581348033189618678473179241064113054288220604554570346239934780818731120408799492843186991
06416865521492614531231783253320962518186939715705143181582332451403715983077334128673665133436729743874451580776
24883327628700339090121726103636453494594859757826706919728961614225489103509524591648412068603529031592421400424
62228517157800262305022000123049056545998080443832936399306107127684527621882721355268489374825603097257342393590
41881081604932493295818851065897194460578664420343244668944218859930843009169151416439747854996132480384642823239
37510426569082960331376397887763359244368115648722603400704786495505539617149452362493152601086524782868042224415
38275178749282986227426052757857191867998595059793083524915936269699881214132693407669538222029456925323422740776
93139016760771380343305198289191630235123888130456500445772972946518534337647490350165170146543325102202345905951
72093806334152213286582560015904004321317422594037838516233510525493306002477377116720950900933852100523276958876
65925181656971753521046745176706965691054651279856277848683984180707708875566485063766798745865102011160387319121
12614381982989015173251887069224596014819312276729672573465458064549365015500115328448677712179357286925172763809
12171040501275955833675779498789506191997771297574643200496889810068611837684823691781892805404521993501864661455
68888916737825212261095398766038535182493969641415734744117861788641521872310728726818324266174080415618260832770
82996504979905121576729761882674166062981961840205928992149503122068791348059470110958176511423269913276037734681
75374573342815663708943893273993369643235668266873246413963203509481806219532095500158056785145343484080545312563
11072058582058900263530915733676717201250162439835733969563931157528056463253080117072765336420863845049611806746
98139170920517060518064920905572492922057196081196237682745164205846001032313420319686537276432540345071367632274
05490746629579607283457523422765298076577299603938223525888633875154018944223661405148737736819187125465224154052
83338036709315811863986807318354486657595424761671804533753551507284143823957942127454371661155353670482639577684
41505047319734903176176353583999693271099471070016855442488574546932825424729408666367431733711263035317370459792
31192698257936230458903181858714897519253070336679857478420570097668897513717070270409212256556106753539856104777
02881428913336435405860941919428810988858611706766144952162176632249696256044947497528415268095623327051305952276
94664136670233457880177927254357892280283501771674153340181295757185220624956488625960241342854349445359997526435
56196039239258953414843632413322104808556421283803041683766333574094938902376763041066775619592300028960103170050
74065957512847787146179418174079951439967447862874204315616543146015363164197737220867778939023005578145446907261
53524427504853234241083524047092443029212029099057510938541927161209092151527496660113715920637523480328430989742
74481011827517880227324994437967194183466104981042482448433555485430033245475238238257538778536250396934904820722
80223786003210772424272735595775962167059549783855774768324817676986910199693637641249080541044312581355623094032
29995669410507869061560001668799471259677428825587266421236272771356548966731920925935827605895525944958390557775
18818259943077397632943736062726286946453727139541112763529510255310608863064434889524537804698808228199130012543
57069805777583827486963206517106298893780072696016043481500959224644667237180899440693022334072471000920301416349
79668048628892544474704539980677959253045377690089917491920942135794067331069174835878047355763069916910505856744
86722847881955207651022032122114422511531155690262132015158094063153145482356620752891978753896071073113439725144
6863301951561400004159243550508846994979798702200216540143630612639070552788234480155374181893452622208321010548621
93900982756780966682798488261862603073406590467181148676985277920649376717780278320966958417359163686441603606338
18829211450682109163209140660219178347576786394979819192039130303782363544036685232979253030440128207221759093730
65139207026094680171262628608854940456945670067491970563095806433729851779889390295662390144586316599244433816941
25529837649434600565981192424844144940596207132164490577515769239206357734633567452508425758085518073722500745788
91032927608366217430799701982166239636878400048541020349474650141837942096607723708232129047047861588703418569357
99967714862154048046954811224815715137042808526270121862674699916359816856408894733734758940978719075353512731556
61295260715372756782780366420854197224596223931832477283860850556978410751616679570401202979195793508679229270695
81350133256315316607015409576382208205168439974873272955396595853298598791179790211071331815536855826621458991831
60733704894572221864142486412665825779350197782684268805844425849723735061349824699601537590837582433332202658893
17394698666742717430467634753497645506911473017508272050122325088825503782634226189081418110360232044483094917122
07334883294519655893004545060618663006646426492368239601914612355782049441001104110535202924830894135675969109298
88137080832814722692866272761520718175205260236429307900516029601710966982661458879524467149727096231213360888002
87469715461344195144343476677832930394086124235564902368800329544881012516861962147429738421449344065429019424751
59424773215247811845340810724337752936427435250190429656583689900348934536801305292077567762663855888745782630624
89301716082492902736211921475784584202757540419343642808732775763483408838308777899523518517353684258377044939793
37701560295930387002196518418354534911347404051951004651581481820521681180801097862520366245152535253445173913441
50342409268322065640168935051565846935539540801721854719107442963978359909489932041003986357594647255805987710580
8942471773922977396345497637789562340536844867686961011228671
```

附录 B：4.2.2 节调度场算法的流程图