ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Отчёт к лабораторной работе №6

по дисциплине

«Языки программирования»

Работу выполнила

Студент группы СКБ222 _____ М. Х. Халимов
подпись, дата

Работу проверил _____ С. А. Булгаков
подпись, дата

Москва 2022

Содержание

Постановка задачи

Доработать класс `bigint` из лабораторной работы №5.
Класс должен соответствовать следующему интерфейсу:

```
class bigint {
    char   *data;
    size_t  size;
    bool    neg ;

  public:
    bigint();
    bigint(const bigint&);
   ~bigint();

    explicit bigint(long);
    explicit bigint(unsigned long);
    explicit bigint(const char*);

    void print() const;

    static bigint add(const bigint& left, const bigint& right);
    static bigint sub(const bigint& left, const bigint& right);
    static bigint mul(const bigint& left, const bigint& right);
    static bigint div(const bigint& left, const bigint& right);
    static bigint div(const bigint& left, const bigint& right, bigint& rest);
    // Lab-06
    static bigint gcd(const bigint& a, const bigint& b);
    static unsigned long Eratosthenes(bigint *sieve, unsigned long size);
    static bigint sqrt(const bigint& value);
};
```

Метод *`gcd`* реализует алгоритм Евклида. Метод *`Eratosthenes`* строит решето Эратосфена, размера не более *`size`*, и возвращает количество первых ненулевых элементов. Метод *`sqrt`* вычисляет квадратный корень.

Реализацию методов класса вынести в файл исходного кода *`bigint.cpp`*.

В основной функции, размещенной в файле *`main.cpp`*, продемонстрировать
применение разработанного класса и его методов.

1. Описание методов класса *bigint*

1.1. Метод, осуществляющий алгоритм Эвклида

Метод вычисляет наибольший общий делитель двух чисел при помощи деления с остатком, реализованного ранее. На вход принимаются два объекта класса *bigint*. Метод возвращает НОД двух чисел. Блок-схему метода можно рассмотреть в *Приложение А1*.

1.2. Метод, составляющий решето Эратосфена

Метод составляет массив простых чисел, называемый в нашем случае решетом Эратосфена. На вход принимается указатель на массив объектов класса *bigint* и переменная типа *unsigned long*. Метод возвращает количество простых чисел в диапазоне от 1 до *n*, где *n* – переменная типа *unsigned long*, получаемая методом на вход. Блок-схему метода можно рассмотреть в *Приложение А.2*.

1.3. Метод, вычисляющий квадратный корень числа

Метод находит квадратный корень числа путем перебора чисел. Если число при возведении в квадрат равно тому числу, корень которого мы ищем – значит оно и является его корнем. На вход принимается объект класса *bigint*. Метод квадратный корень входного числа. Блок-схему метода можно рассмотреть в *Приложение А.3*.

2. Функция main

Функция *main* включает в себя создание объектов класса *bigint* и вызов методов этого же класса. В результате работы функции в консоль выводится НОД, количество простых чисел до *n* и квадратный корень из числа. Блок-схему метода можно рассмотреть в *Приложение А.4*.

3. Результаты тестирования программы

Листинг

```
-124124
2444
-700
```

```
~~~GCD~~~
52
28
4

~~~ERATOSTHENES
168
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
53
59
61
67
71
73
79
83
89
97
101
103
107
109
113
127
131
137
139
149
151
157
163
167
173
179
181
191
193
```

197
199
211
223
227
229
233
239
241
251
257
263
269
271
277
281
283
293
307
311
313
317
331
337
347
349
353
359
367
373
379
383
389
397
401
409
419
421
431
433
439
443
449
457
461
463
467
479
487
491
499
503

509
521
523
541
547
557
563
569
571
577
587
593
599
601
607
613
617
619
631
641
643
647
653
659
661
673
677
683
691
701
709
719
727
733
739
743
751
757
761
769
773
787
797
809
811
821
823
827
829
839
853
857

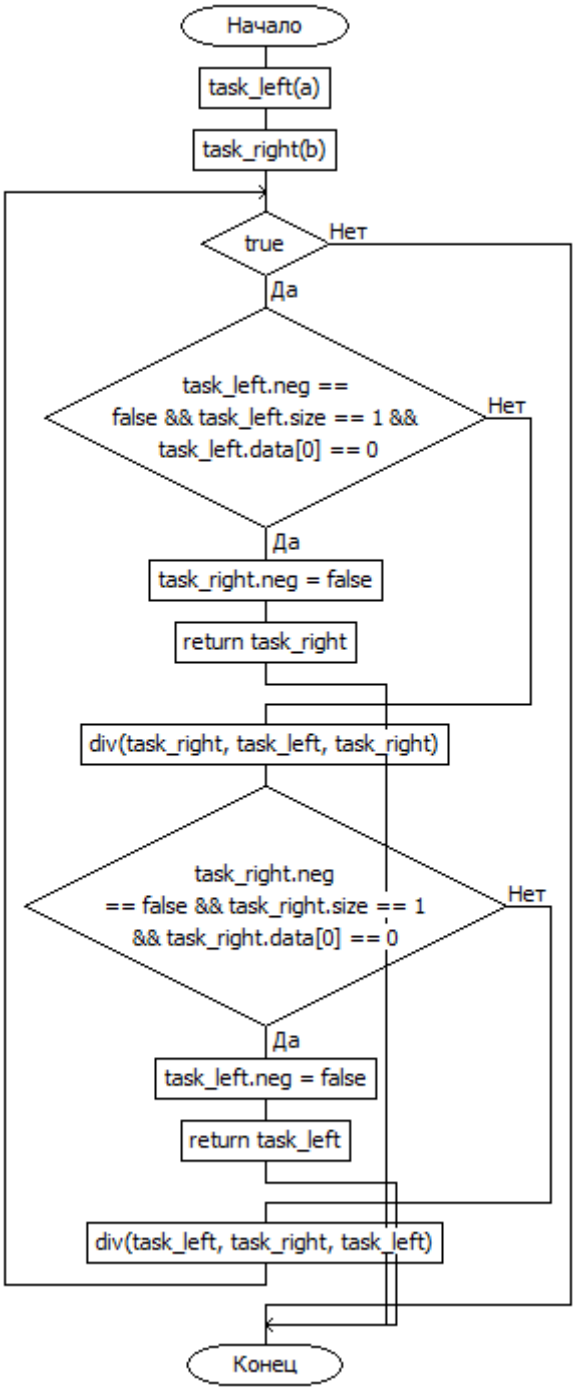
```
859
863
877
881
883
887
907
911
919
929
937
941
947
953
967
971
977
983
991
997

~~~SQRT~~~
49
ERROR!
```

Приложение А

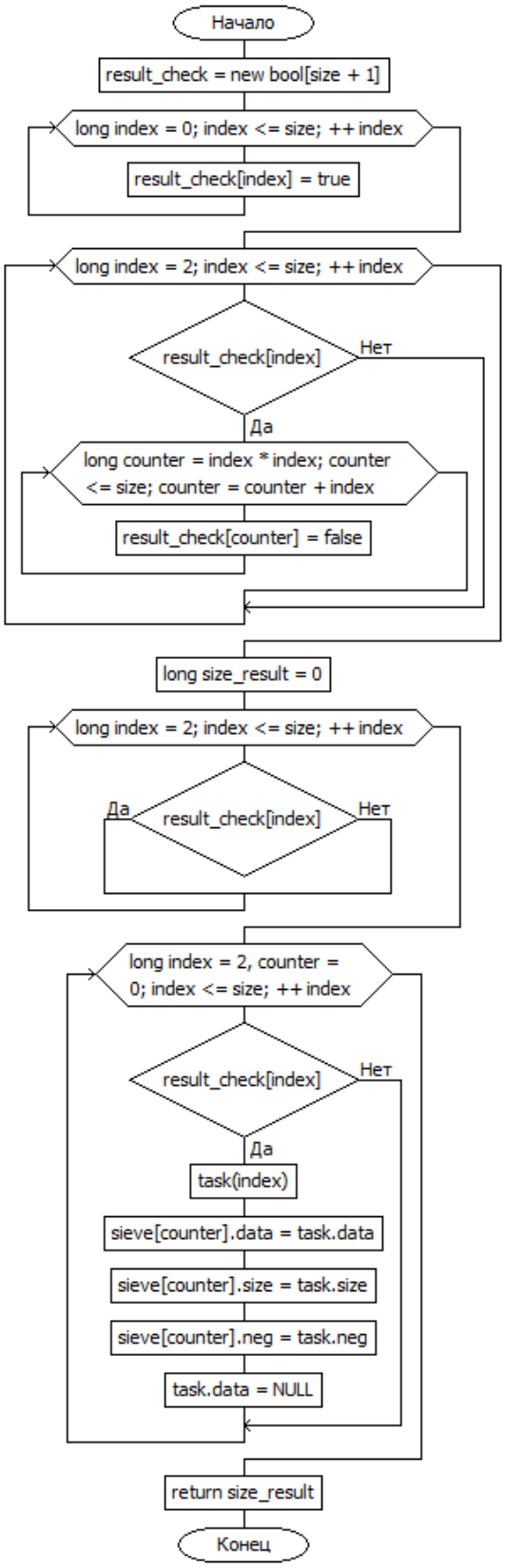
А.1.

Блок-схема метода, реализующего алгоритм Евклида



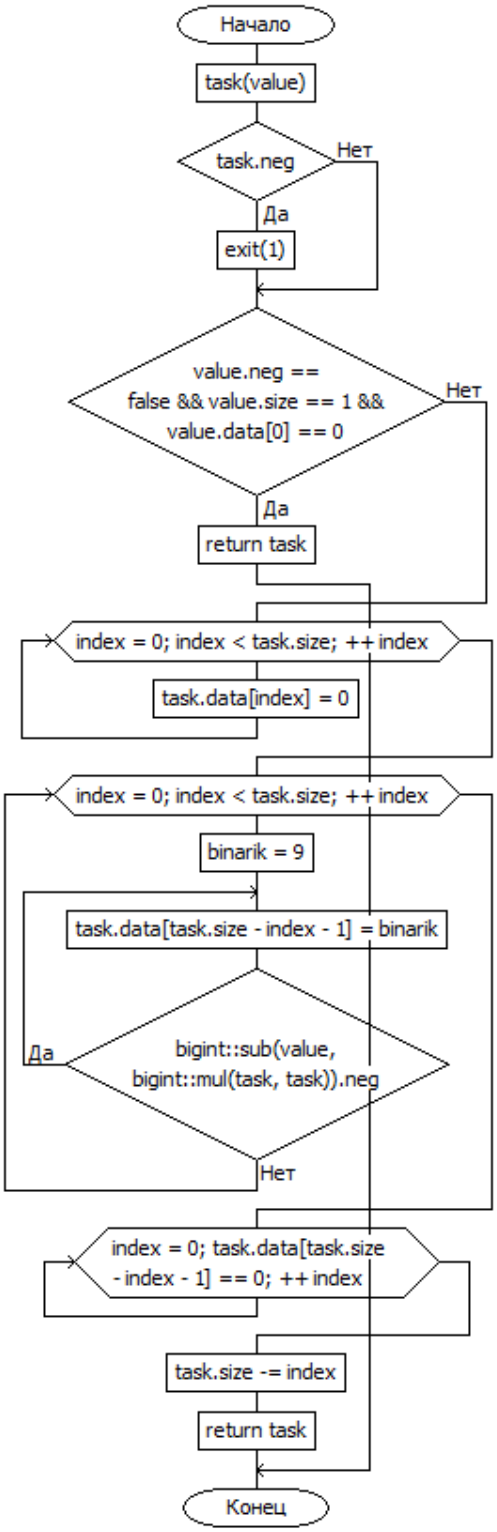
A.2.

Блок-схема метода, реализующего решето Эратосфена



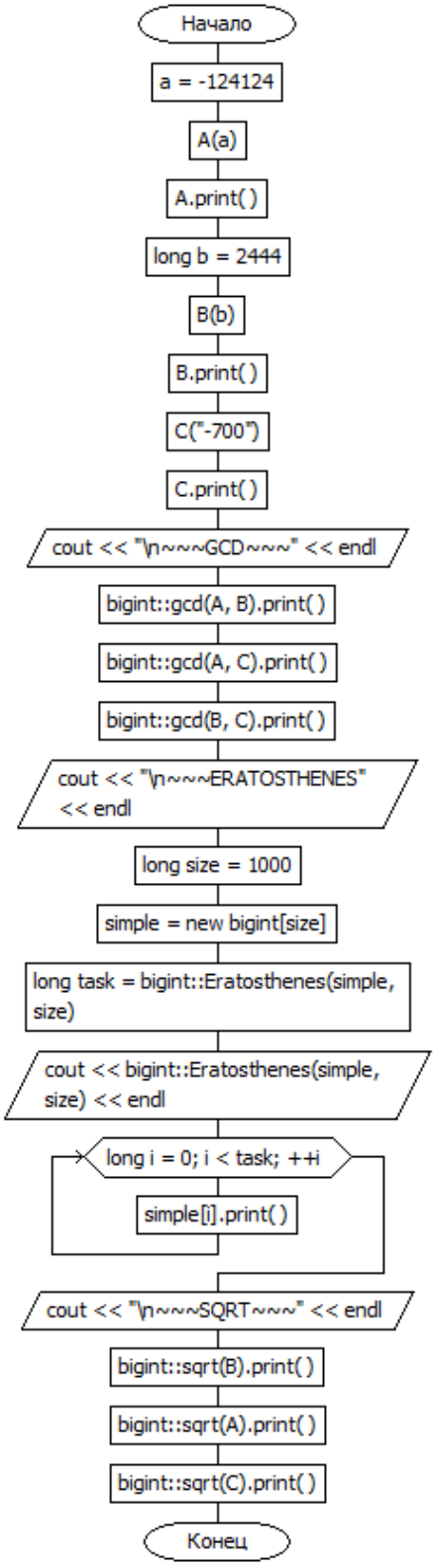
А.3.

Блок-схема метода, реализующего нахождение квадратного корня некого числа



A.4.

Блок-схема функции *main*



Приложение Б

Исходный код

```
#include <iostream>
#include "bigint.h"
#include <cstdlib>

using std::cerr;
using std::cout;
using std::endl;
using std::exit;

bigint::bigint()
{
     data = NULL;
     size = 0;
     neg = false;
}

bigint::bigint(const bigint& object)
{
     data = object.data;
     size = object.size;
     neg = object.neg;

     try
     {
          data = new char [size];
     }
     catch(...)
     {
          cerr << "Error AM" << endl;
          exit(1);
     }

     for(size_t i = 0; i < size; i++)
          data[i] = object.data[i];
}

bigint::~bigint()
{
     delete[] data;
     data = NULL;
}

bigint::bigint(long data_long)
{
     data = NULL;
     size = 0;
     neg = false;
```

```
	long data_long_clone = data_long;


	if( data_long_clone < 0)
	{
		data_long = -data_long;
		neg = true;
		size = 1;
		while ( data_long_clone < 0 )
		{
			data_long_clone = data_long_clone / 10;
			size++;
		}
		size--;
	}
	else if (data_long_clone > 0)
	{
		while (data_long_clone > 0)
		{
			data_long_clone = data_long_clone / 10;
			size++;
		}
	}
	else
	{
		size = 1;
	}

	try
	{
		data = new char [size];
	}
	catch(...)
	{
		cerr << "Error AM" << endl;
		exit(1);
	}

	for(size_t i = 0; i < size; i++)
	{
		data[i] = data_long % 10;
		data_long = data_long / 10;
	}
}

bigint::bigint(unsigned long data_unsigned_long)
{
	data = NULL;
	size = 0;
	neg = false;
```

```
	unsigned long data_unsigned_long_clone = data_unsigned_long;

	while (data_unsigned_long_clone > 0)
	{
		data_unsigned_long_clone = data_unsigned_long_clone / 10;
		size++;
	}

	try
	{
		data = new char [size];
	}
	catch(...)
	{
		cerr << "Error AM" << endl;
		exit(1);
	}

	for(size_t i = 0; i < size; i++)
	{
		data[i] = data_unsigned_long % 10;
		data_unsigned_long = data_unsigned_long / 10;
	}
}

bigint::bigint(const char* arr)
{
	data = NULL;
	size = 0;
	neg = false;

	if(arr[0] == '-')
	{
		neg = true;
	}

	size_t len_str = 0;

	while (arr[len_str] != '\0')
	{
		++len_str;
	}

	size = len_str;

	if(neg == true)
	{
		--size;
	}

	try
	{
```

```
            data = new char[size];
        }
        catch (...)
        {
            cerr << "Error AM" << endl;
            exit(1);
        }



        for(size_t i = 0; i < size; ++i)
        {
            data[i] = arr[len_str - 1 - i] - '0';
        }

        int flag = 0;

        for(size_t i = 0; i < size; ++i)
        {
            if(data[i] != 0)
            {
                flag = 1;
            }
        }

        if (flag == 0)
        {
            neg = false;
        }
}

void bigint::print() const
{
        if (neg)
            cout << "-";
        for(size_t i = 0; i < size; ++i)
        {
            cout << static_cast<char>(data[size-1-i]+48);
        }
        cout << endl;
}

bigint bigint::add(const bigint& left, const bigint& right)
{
        char* data_result;
        //bool neg_result;
        size_t size_result, left_size_clone = left.size, right_size_clone
= right.size;

        bigint task;

        if (left.size > right.size)
```

```
    {
        size_result = left.size;
    }
    else
    {
        size_result = right.size;
    }

    size_result = size_result + 1; //Потому что может быть увелечение
разряда (это я для себя, потому что читая продолжив писать через 2
дня, я забыл что писал)

    try
    {
        data_result = new char[size_result];
    }
    catch (...)
    {
        cerr << "Error AM" << endl;
        exit(1);
    }

    for(size_t i = 0; i < size_result; ++i)
    {
        data_result[i] = 0;
    }

    size_t size_min = 0;
    left_size_clone = left.size;
    right_size_clone = right.size;

    while(right_size_clone != 0 && left_size_clone != 0)
    {
        right_size_clone--;
        left_size_clone--;
        size_min++;
    }

    if(left.neg == right.neg)
    {
        size_t counter;

        for(counter = 0; counter < size_min; ++counter)
        {
            data_result[counter] = left.data[counter] +
right.data[counter];
        }

        if(right.size > left.size)
        {
            for(; counter < right.size; ++counter)
            {
```

```
                    data_result[counter] = right.data[counter];
               }
          }
          else
          {
               for(; counter < left.size; ++counter)
               {
                    data_result[counter] = left.data[counter];
               }
          }

          for(size_t i = 0; i < size_result - 1; ++i)
          {
               if(data_result[i] >= 10)
               {
                    data_result[i] = data_result[i] - 10;
                    data_result[i + 1] = data_result[i + 1] + 1;
               }
          }

          if(data_result[size_result - 1] == 0)
          {
               task.size = size_result - 1;
          }
          else
          {
               task.size = size_result;
          }
          task.neg = right.neg;
          task.data = data_result;
     }
     else
     {
          size_t counter;

          for(counter = 0; counter < size_min; ++counter)
          {
               data_result[counter] = left.data[counter] -
right.data[counter];
          }

          if(right.size > left.size)
          {
               for(; counter < right.size; ++counter)
               {
                    data_result[counter] = -right.data[counter];
               }
          }
          else
          {
               for(; counter < left.size; ++counter)
               {
```

```
                data_result[counter] = left.data[counter];
            }
        }

        bool absolute_sign = true;

        for(size_t i = 0; i < size_result; ++i)
        {
            if(data_result[size_result - 1 - i] != 0)
            {
                absolute_sign = data_result[size_result - 1 - i] >
0;
                break;
            }
            if (i == size_result - 1 && left.neg)
            {
                absolute_sign = false;
            }
        }

        if(absolute_sign)
        {
            for (size_t i = 0; i < size_result; ++i)
            {
                if(data_result[i] < 0)
                {
                    data_result[i] = data_result[i] + 10;
                    data_result[i + 1] = data_result[i + 1] - 1;
                }
            }
        }
        else
        {
            for(size_t i = 0; i < size_result; ++i)
            {
                data_result[i] = -data_result[i];
            }
            for(size_t i = 0; i < size_result; ++i)
            {
                if(data_result[i] < 0)
                {
                    data_result[i] = data_result[i] + 10;
                    data_result[i + 1] = data_result[i + 1] - 1;
                }
            }
        }

        counter = 0;

        while(size_result - 1 - counter > 0 &&
data_result[size_result - counter - 1] == 0)
        {
```

```
            ++counter;
        }

        task.size = size_result - counter;
        task.neg = absolute_sign == left.neg;
        task.data = data_result;

    }

    return task;
}

bigint bigint::sub(const bigint& left, const bigint& right)
{
    bigint task(right);
    task.neg = !task.neg;

    return add(left, task);
}

bigint bigint::mul(const bigint& left, const bigint& right)
{
    char* data_result = NULL;
    bool neg_result = true;
    size_t size_result = left.size + right.size;
    int* data_int = NULL;
    bigint task;

    if (left.neg == right.neg)
    {
        neg_result = false;
    }
    else
    {
        neg_result = true;
    }

    try
    {
        data_int = new int[size_result];
        data_result = new char[size_result];
    }
    catch(...)
    {
        cerr << "Error AM" << endl;
        exit(1);
    }

    for(size_t i = 0; i < size_result; ++i)
    {
        data_int[i] = 0;
    }
```

```
    for(size_t index_left = 0; index_left < left.size; index_left++)
    {
        for(size_t index_right = 0; index_right < right.size;
index_right++)
        {
            data_int[index_right + index_left] +=
left.data[index_left] * right.data[index_right];
        }
    }

    for(size_t index_result = 0; index_result < size_result;
index_result++)
    {
        data_int[index_result + 1] += data_int[index_result] / 10;
        data_int[index_result] = data_int[index_result] % 10;
    }

    /*while(data_result[size_result] == 0)
    {
        size_result--;
    }*/

    for(size_t i = 0; i < size_result; ++i )
    {
        data_result[i] = static_cast<char>(data_int[i]);
    }

    delete[] data_int;

    task.data = data_result;

    bool zero = true;

    for(size_t in = 0; in < size_result; ++in)
    {
        zero = zero && (data_result[in] == 0);
    }

    if(zero == true)
    {
        task.size = 1;
        task.neg = false;
        return task;
    }

    size_t i = 0;

    for(; data_result[size_result - 1 - i] == 0; i++);


    task.size = size_result - i;
```

```
     task.neg = neg_result;

     return task;
}

bigint bigint::div(const bigint& left, const bigint& right)
{

     bigint left_clone(left);
     bigint right_clone(right);

     left_clone.neg = false;
     right_clone.neg = false;

     bigint task(left_clone);

     for (size_t i = 0; i < task.size; ++i)
     {
          task.data[i] = 0;
     }
     for(size_t i = 0; i < task.size; ++i)
     {
          size_t ind = task.size - 1 - i;
          char l = 0, r = 9, m;

          while(l < r)
          {
               m = (l + r + 1) / 2;
               task.data[ind] = m;
               if (bigint::sub(left_clone, bigint::mul(task,
right_clone)).neg)
               {
                    r = m - 1;
               }
               else
               {
                    l = m;
               }
          }

          task.data[ind] = l;
     }

     bool zero = true;

     for(size_t i = 0; i < task.size; ++i)
     {
          zero = zero && (task.data[i] == 0);
     }

     if(zero)
     {
```

```
          task.size = 1;
          task.neg = false;
          return task;
     }

     size_t index = 0;

     for(index = 0; task.data[task.size - 1 - index] == 0; ++index);

     task.size -= index;

     task.neg = left.neg != right.neg;

     return task;
}

bigint bigint::div(const bigint& left, const bigint& right, bigint&
rest)
{
     bigint task = bigint::div(left, right);
     bigint* rem = new bigint(bigint::sub(left, bigint::mul(task,
right)));

     rest.data = rem -> data;
     rest.size = rem -> size;
     rest.neg = rem -> neg;

     rem -> data = NULL;

     delete rem;

     return task;
}

bigint bigint::gcd(const bigint& a, const bigint& b)
{
     bigint task_left(a);
     bigint task_right(b);

     while(true)
     {
          if (task_left.neg == false && task_left.size == 1  &&
task_left.data[0] == 0)
          {
               task_right.neg = false;
               return task_right;
          }

          div(task_right, task_left, task_right);
          if (task_right.neg == false && task_right.size == 1 &&
task_right.data[0] == 0)
          {
```

```
            task_left.neg = false;
            return task_left;
        }
        div(task_left, task_right, task_left);
    }
}

unsigned long bigint::Eratosthenes (bigint *sieve, unsigned long size)
{
    bool* result_check = new bool[size + 1];

    for(unsigned long index = 0; index <= size; ++index)
    {
        result_check[index] = true;
    }

    for(unsigned long index = 2; index <= size; ++index)
    {
        if(result_check[index])
        {
            for(unsigned long counter = index * index; counter <=
size; counter = counter + index)
            {
                result_check[counter] = false;
            }
        }
    }

    unsigned long size_result = 0;

    for(unsigned long index = 2; index <= size; ++index)
    {
        if(result_check[index])
        {
            ++size_result;
        }
    }

    for(unsigned long index = 2, counter = 0; index <= size; ++index)
    {
        if(result_check[index])
        {
            bigint task(index);

            sieve[counter].data = task.data;
            sieve[counter].size = task.size;
            sieve[counter].neg = task.neg;

            task.data = NULL;

            ++counter;
        }
```

```
    }

    return  size_result;
}

bigint bigint::sqrt(const bigint& value)
{
    bigint task(value);

    if(task.neg)
    {
        cerr << "ERROR!" << endl;
        exit(1);
    }

    if(value.neg == false && value.size == 1 && value.data[0] == 0)
    {
        return task;
    }

    for(size_t index = 0; index < task.size; ++index)
    {
        task.data[index] = 0;
    }

    for(size_t index = 0; index < task.size; ++index)
    {
        char binarik = 9;
        do
        {
            task.data[task.size - index - 1] = binarik;
            --binarik;
        } while(bigint::sub(value, bigint::mul(task, task)).neg);
    }

    size_t index;
    for(index = 0; task.data[task.size - index - 1] == 0; ++index);

    task.size -= index;

    return task;
}
```

Приложение В

UML-схема класса *bigint*