

Министерство образования и науки Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ  
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ  
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ  
«САРАТОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ  
ИМЕНИ Н.Г. ЧЕРНЫШЕВСКОГО»

Кафедра математической  
кибернетики и компьютерных наук

**РАЗРАБОТКА ВЫСОКОНАГРУЖЕННЫХ ПРИЛОЖЕНИЙ НА ЯЗЫКЕ  
JAVASCRIPT.**

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА**

Студента 4 курса 411 группы  
направления 010300 — Фундаментальная информатика и информационные  
технологии  
факультета КНиИТ  
Низамутдинова Артура Салаватовича

Научный руководитель

доцент, к. ф.-м. н.

\_\_\_\_\_

С. В. Миронов

Заведующий кафедрой

к.ф.-м.н.

\_\_\_\_\_

А. С. Иванов

Саратов 2016

## Содержание

ВВЕДЕНИЕ .....	3
1 Архитектура NodeJS .....	4
2 Наибольший общий делитель и наименьшее общее кратное .....	5
3 Алгоритмы нахождения наибольшего общего делителя .....	6
3.1 Античный алгоритм Евклида .....	6
3.2 Алгоритм Евклида .....	6
3.3 Алгоритм нахождения НОД методом перебора .....	6
3.4 Бинарный алгоритм Стейна .....	6
4 Реализация алгоритмов .....	8
4.1 Подробное описание алгоритмов .....	8
4.1.1 Античный алгоритм Евклида .....	8
4.1.2 Алгоритм Евклида .....	8
4.1.3 Алгоритм нахождения НОД методом перебора .....	9
4.1.4 Бинарный алгоритм Стейна .....	10
4.2 Экспериментальные данные .....	11
ЗАКЛЮЧЕНИЕ .....	15
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ .....	16
Приложение А Листинг античного алгоритма Евклида, написанного на Microsoft Visual Studio C++ 2010 .....	17
Приложение Б Листинг алгоритма Евклида, написанного на Microsoft Visual Studio C++ 2010 .....	19
Приложение В Листинг алгоритма для нахождения НОД методом пере- бора, написанного на Microsoft Visual Studio C++ 2010 .....	21
Приложение Г Листинг бинарного алгоритма Стейна, написанного на Microsoft Visual Studio C++ 2010 .....	23
Приложение Д Листинг программы для нахождения чисел Фибоначчи, написанного на Delphi 7.0 .....	26

## ВВЕДЕНИЕ

В 2009 году Райан Дал после двух лет экспериментирования над созданием серверных веб-компонентов разработал NodeJS.

NodeJS является программной платформой, основанной на разработанной компанией Google JavaScript—движке V8 (транслирующем JavaScript в машинный код), преобразующий JavaScript из узкоспециализированного в язык общего назначения. NodeJS используется преимущественно на сервере, выполняя роль веб-сервера, но кроме этого есть возможность разрабатывать на NodeJS и десктопные приложения (с использованием NW.js, Electron или AppJS для Windows, Linux и Mac OS) и даже программировать микроконтроллеры (например, espruino и tessell).

В основе Node.js лежит событийно-ориентированное и асинхронное (или реактивное) программирование с неблокирующим вводом/выводом.

Поставленные задачи:

- изучить архитектуру NodeJS;
- рассмотреть способы асинхронного обмена данными с сервером с использованием websocket, comet, iframe и jsonp;
- сравнить websocket и comet;
- реализовать веб-сервер на websocket и comet;
- реализовать утилиту для нагрузочного тестирования веб-сервера на websocket и comet;
- провести нагрузочное тестирование и сделать выводы.

## 1 Архитектура NodeJS

В основе NodeJS лежит выполнение приложения в одном программном потоке, а также асинхронная обработка всех событий. При запуске NodeJS-приложения создается единственный программный поток. NodeJS-приложение выполняется в этом потоке в ожидании, что некое приложение сделает запрос. Когда NodeJS-приложение получает запрос, то никакие другие запросы не обрабатываются до тех пор, пока не завершится обработка текущего запроса.

На первый взгляд, все это кажется не очень эффективным, если бы не то обстоятельство, что NodeJS работает в асинхронном режиме, используя цикл обработки событий и функции обратного вызова. Цикл обработки событий просто опрашивает конкретные события и в нужное время вызывает обработчики событий. В NodeJS таким обработчиком событий является функция обратного вызова.

В отличие от других однопоточных приложений, когда к NodeJS - приложению делается запрос, оно должно, в свою очередь, запросить какие-то ресурсы (например, получить доступ к файлу или обратиться к базе данных). В этом случае NodeJS инициирует запрос но не ожидает ответа на этот запрос. Вместо этого запросу назначается некая функция обратного вызова. Когда запрошенное значение будет готово (или завершено) генерируется событие, активизирующее соответствующую функцию обратного вызова, призванную что-то сделать либо с результатами запрошенного действия, либо с запрошенными ресурсами.

Если несколько человек обращаются к NodeJS-приложению в одно и то же время и приложению нужно обратиться к ресурсам из файла, для каждого запроса NodeJS назначает свою функцию обратного вызова событию ответа. Когда для каждого из них ресурс становится доступен, вызывается нужная функция обратного вызова, и запрос удовлетворяется. В промежутке NodeJS-приложение может обрабатывать другие запросы либо для того же приложения, либо для какого-нибудь другого.

Хотя приложение не обрабатывает запросы в параллельном режиме, в зависимости от своей загруженности и конструкции можно даже не заметить задержки в ответе. А что лучше всего, приложение очень экономно относится к памяти и к другим ограниченным ресурсам. [1]

## 2 Наибольший общий делитель и наименьшее общее кратное

Наибольший общий делитель целых  $a, b$  — это такой их общий делитель, который делится на любой общий делитель этих чисел.

Наименьшее общее кратное целых  $a, b$  — это такое наименьшее натуральное число, которое делится на  $a$  и  $b$ .

Если числа  $a$  и  $b$  представить в виде  $a = p_1^{d_1} \cdot \dots \cdot p_n^{d_n}$   $b = p_1^{e_1} \cdot \dots \cdot p_n^{e_n}$ , где  $p_1, \dots, p_n$  — простые числа,  $e_1, \dots, e_n$  и  $d_1, \dots, d_n$  — целые неотрицательные числа (некоторые из них могут быть равны нулю, если соответствующего простого числа нет в разложении). Тогда НОД и НОК этих чисел можно представить в следующем виде:

$$\text{НОД}(a, b) = p_1^{\min(d_1, e_1)} \cdot \dots \cdot p_n^{\min(d_n, e_n)}$$

$$\text{НОК}(a, b) = p_1^{\max(d_1, e_1)} \cdot \dots \cdot p_n^{\max(d_n, e_n)}$$

Пусть  $a$  и  $b$  — два каких-то целых числа, не равных одновременно нулю; рассмотрим совокупность всех натуральных чисел, на которые делятся и  $a$  и  $b$ . Эта совокупность, несомненно, конечная, так как если, например,  $a \neq 0$ , то никакое число, большее чем  $a$ , не может быть делителем  $a$ . Отсюда следует, что число общих делителей  $a$  и  $b$  конечно; пусть через  $d$  обозначен наибольший из них. Число  $d$  называется общим наибольшим делителем  $a$  и  $b$ , и мы условимся обозначать его  $d = (a, b)$ . Если  $a$  и  $b$  — достаточно большие числа, например  $a = 24157819$ ,  $b = 39088167$ , то попытки найти общий наибольший делитель с помощью непосредственных проб довольно утомительны [2]. Короткий и вполне надежный метод вытекает из алгоритмов нахождения НОД.

### 3 Алгоритмы нахождения наибольшего общего делителя

Для вычисления НОД двух чисел существуют следующие алгоритмы:

1. Античный алгоритм Евклида (через разности);
2. Алгоритм Евклида (через остатки);
3. Нахождение НОД методом перебора;
4. Бинарный алгоритм Стейна.

НОК можно вычислить при помощи следующей формулы

$$\text{НОК} = \frac{ab}{\text{НОД}(a, b)}.$$

#### 3.1 Античный алгоритм Евклида

Суть алгоритма заключается в том, что если даны два числа, и одно из них равно нулю, то в ответ записывается большее из них. Если ни одно из них не равно нулю, тогда вычитаем из большего числа меньшее, а потом снова проверяем условие равенства одного из них нулю.

#### 3.2 Алгоритм Евклида

В алгоритме Евклида, если одно из чисел равно нулю, то в ответ записывается наибольшее из них. Иначе большему числу присваиваем остаток от деления его на меньшее, а потом снова проверяем условие равенства одного из них нулю.

#### 3.3 Алгоритм нахождения НОД методом перебора

В данном случае мы выбираем  $\max(a, b)$  и уменьшаем это число на единицу до тех пор, пока  $a$  и  $b$  не станут одновременно кратны ему.

#### 3.4 Бинарный алгоритм Стейна

В 1961 году израильский физик и программист Джозеф Стейн (Josef Stein) представил совершенно иной алгоритм нахождения наибольшего общего делителя [3], использующий, прежде всего, на бинарную арифметику. Этому новому алгоритму совершенно не нужны команды, совершающие операции деления. Основанный исключительно на операциях вычитания, он проверяет,

является ли число четным, и делит пополам четные числа (что соответствует в бинарной арифметике сдвигу вправо). Бинарный алгоритм поиска наибольшего общего делителя основан прежде всего на четырех простых фактах относительно положительных целых чисел  $a$  и  $b$ :

1. Если оба числа четны, то

$$\text{НОД}(2a, 2b) = 2 \text{НОД}(a, b).$$

2. Если первое число четно, а второе нечетно, то

$$\text{НОД}(2a, 2b + 1) = \text{НОД}(a, 2b + 1).$$

3. Также как и в алгоритме Евклида,

$$\text{НОД}(a, b) = \text{НОД}(a - b, b).$$

4. Если оба числа нечетные, то  $a - b$  - четно и

$$|a - b| < \max(a, b).$$

## 4 Реализация алгоритмов

В данном разделе приведены описания следующих алгоритмов:

1. Античный алгоритм Евклида (через разности);
2. Алгоритм Евклида (через остатки);
3. Нахождение НОД методом перебора;
4. Бинарный алгоритм Стейна.

Листинг программ, реализующих эти алгоритмы можно увидеть в приложениях А—Г. Стоит заметить, что алгоритмы представленные в этих приложениях были написаны на языке C++ в среде Visual Studio 2010, и их тестирование на скорость вычисления поставленной задачи проводилось в операционной системе Windows 8, на компьютере HP Envy m6 1154er с процессором Intel Core i5 3210M, тактовая частота которого 2.5 ГГц.

### 4.1 Подробное описание алгоритмов

Далее приведены краткие рекомендации по работе с представленными программами, блок-схемы и скриншоты работы программ.

#### 4.1.1 Античный алгоритм Евклида

На вход программе должно быть подано два целых положительных числа (те самые числа  $a$  и  $b$ ) и нажата клавиша Enter. Далее во второй, третьей и четвертой строке появятся  $\text{НОД}(a, b)$ ,  $\text{НОК}(a, b)$  и время работы программы в миллисекундах соответственно. Общий вид реализованного алгоритма представлен на рисунке 1. Код программы можно увидеть в приложении А.

На рисунке 2 можно увидеть скриншот работы программы.

#### 4.1.2 Алгоритм Евклида

Здесь также, как и в предыдущем описании, на вход программе должно быть подано два целых положительных числа (те самые числа  $a$  и  $b$ ) и нажата клавиша Enter. Далее во второй, третьей и четвертой строке появятся  $\text{НОД}(a, b)$ ,  $\text{НОК}(a, b)$  и время работы программы в миллисекундах соответственно. Код программы можно увидеть в приложении Б. Общий вид реализованного алгоритма представлен на рисунке 3.

На рисунке 4 можно увидеть скриншот работы программы.



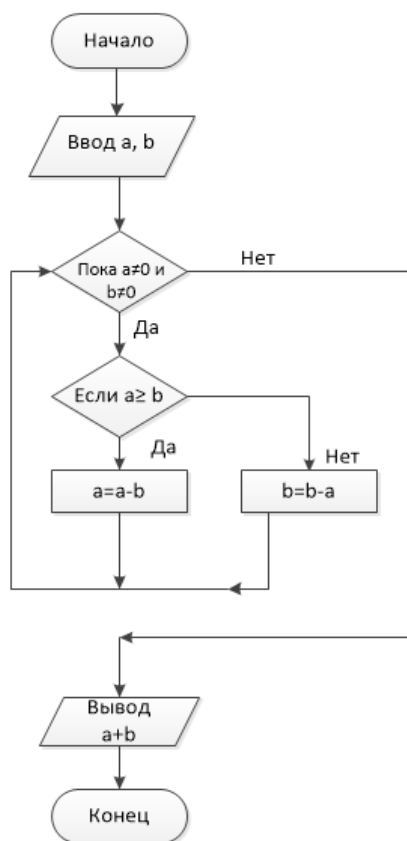


Рисунок 1 – Блок-схема античного алгоритма Евклида

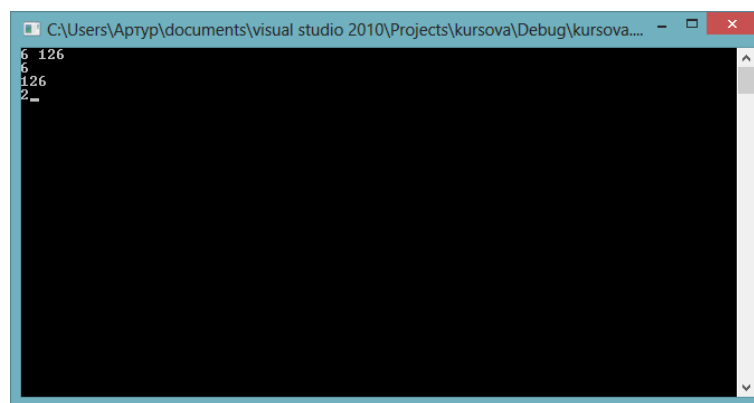


Рисунок 2 – Работа античного алгоритма Евклида

#### 4.1.3 Алгоритм нахождения НОД методом перебора

Здесь также, как и в двух предыдущих описаниях, на вход программе должно быть подано два целых положительных числа (те самые числа  $a$  и  $b$ ) и нажата клавиша Enter. Далее во второй, третьей и четвертой строке появятся  $\text{НОД}(a, b)$ ,  $\text{НОК}(a, b)$  и время работы программы в миллисекундах соответственно. Код программы можно увидеть в приложении В. На рисунке 5 представлена блок-схема реализованного алгоритма.

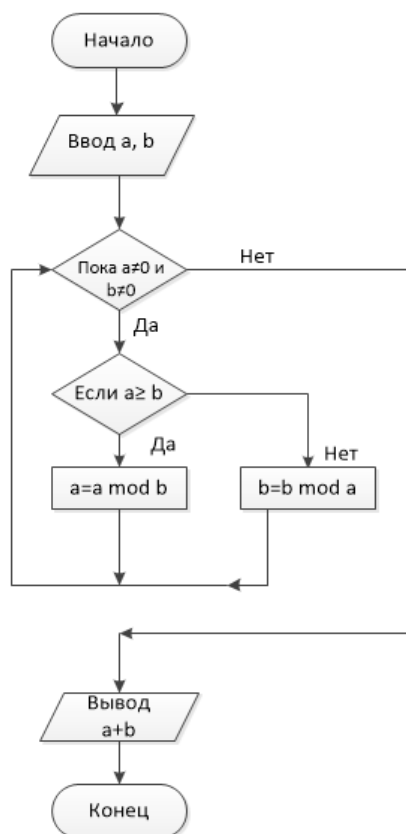


Рисунок 3 – Блок-схема алгоритма Евклида

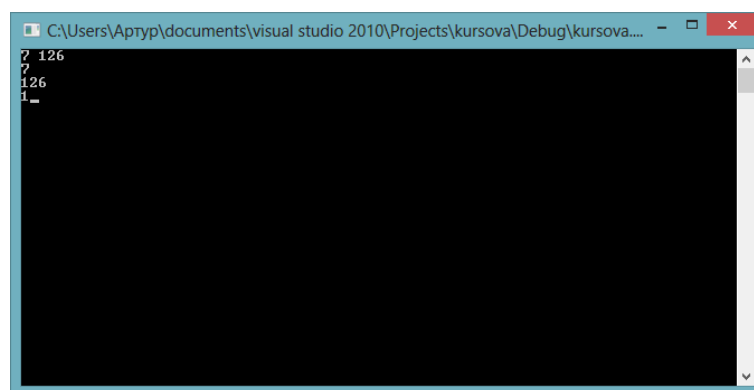


Рисунок 4 – Работа алгоритма Евклида

На рисунке 6 можно увидеть скриншот работы программы.

#### 4.1.4 Бинарный алгоритм Стейна

Здесь также, как и в трех предыдущих описаниях, на вход программе должно быть подано два целых положительных числа  $a$  и  $b$  и нажата клавиша Enter. Далее во второй, третьей и четвертой строке появятся  $\text{НОД}(a, b)$ ,  $\text{НОК}(a, b)$  и время работы программы в миллисекундах соответственно. Код программы можно увидеть в приложении Г. Общий вид реализованного алго-

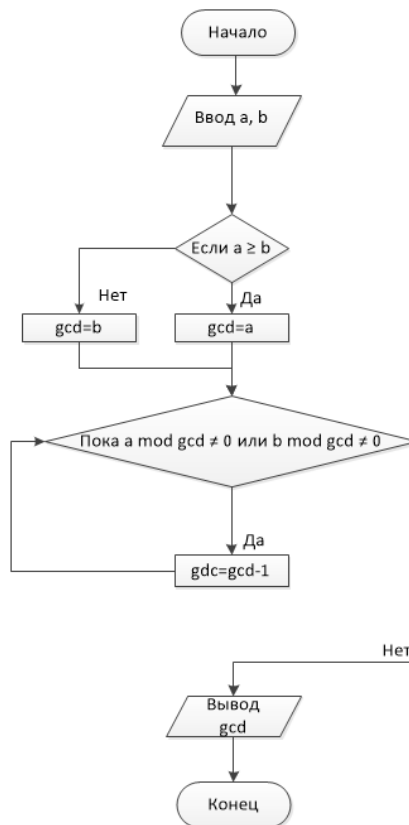


Рисунок 5 – Блок-схема алгоритма для нахождения НОД методом перебора

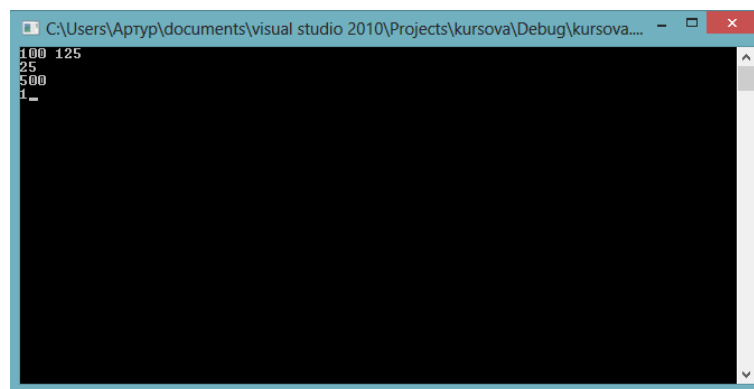


Рисунок 6 – Работа алгоритма для нахождения НОД методом перебора

ритма представлен в виде блок-схемы на рисунке 7.

На рисунке 8 можно увидеть скриншот работы программы.

## 4.2 Экспериментальные данные

Время работы алгоритма Евклида оценивается теоремой Ламе, которая устанавливает поразительную связь с последовательностью Фибоначчи:

Если  $a > b \geq 1$  и  $b < F_n$  для некоторого  $n$ , то алгоритм Евклида выполнит не более  $n - 2$  рекурсивных вызовов.

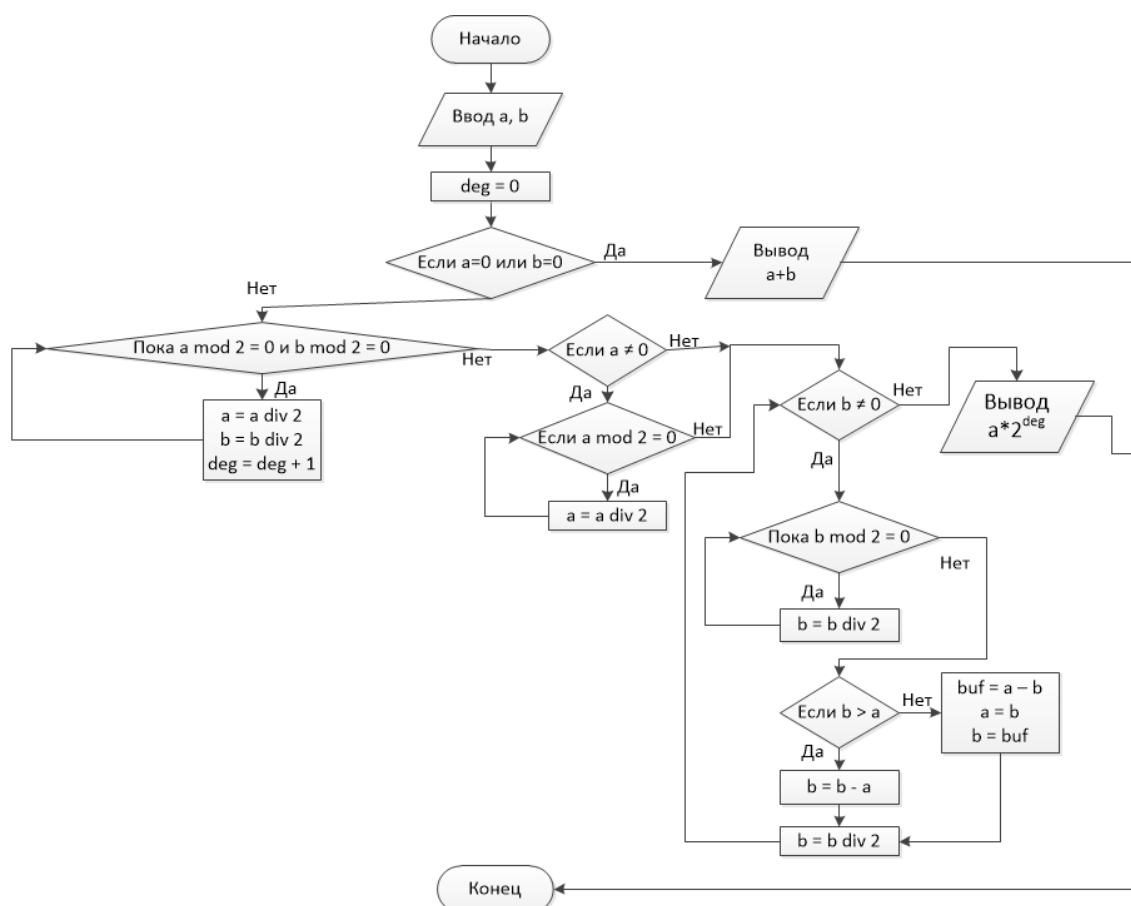


Рисунок 7 – Блок-схема бинарного алгоритма Стейна

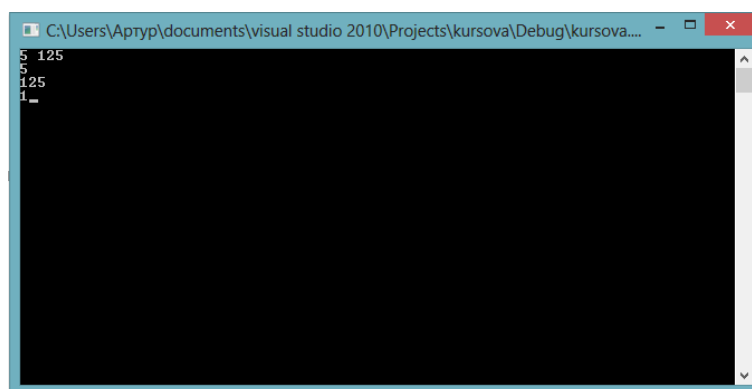


Рисунок 8 – Работа бинарного алгоритма Стейна

Более того, можно показать, что верхняя граница этой теоремы — оптимальная. При  $a = F_n$ ,  $b = F_{n-1}$  будет выполнено именно  $n - 2$  рекурсивных вызова. Иными словами, последовательные числа Фибоначчи — наихудшие входные данные для алгоритма Евклида.

Учитывая, что числа Фибоначчи растут экспоненциально (как константа в степени  $n$ ), получаем, что алгоритм Евклида выполняется за  $O(\log \min(a, b))$  операций умножения.

Поскольку НОК выражается через НОД, то скорость его вычисления напрямую зависит от выбранного алгоритма нахождения НОД.

С учетом этого, было необходимо написать программу для нахождения  $n$ -го числа Фибоначчи ( $1 \leq n \leq 15000$ ) (см. Приложение Д). Алгоритм был реализован на языке Delphi 7.0 в рамках решения задачи [4]. С помощью данного алгоритма были вычислены 19-е, 20-е, 24-е, 25-е, 44-е, 45-е, 49-е, 50-е числа Фибоначчи:

$$F_{19} = 4181;$$

$$F_{20} = 6765;$$

$$F_{24} = 46368;$$

$$F_{25} = 75025;$$

$$F_{44} = 701408733;$$

$$F_{45} = 1134903170;$$

$$F_{49} = 7778742049;$$

$$F_{50} = 12586269025.$$

Они были использованы, как входные данные, для проверки скорости работы программ. То есть вместо чисел  $a$  и  $b$  подставлялись соседние числа Фибоначчи, а именно  $F_{19}$  и  $F_{20}$ ,  $F_{24}$  и  $F_{25}$ ,  $F_{44}$  и  $F_{45}$ ,  $F_{49}$  и  $F_{50}$  соответственно.

На рисунке 9 можно увидеть диаграмму, на которой показано сколько времени потребовалось каждому алгоритму, чтобы найти  $\text{НОД}(a, b)$ .

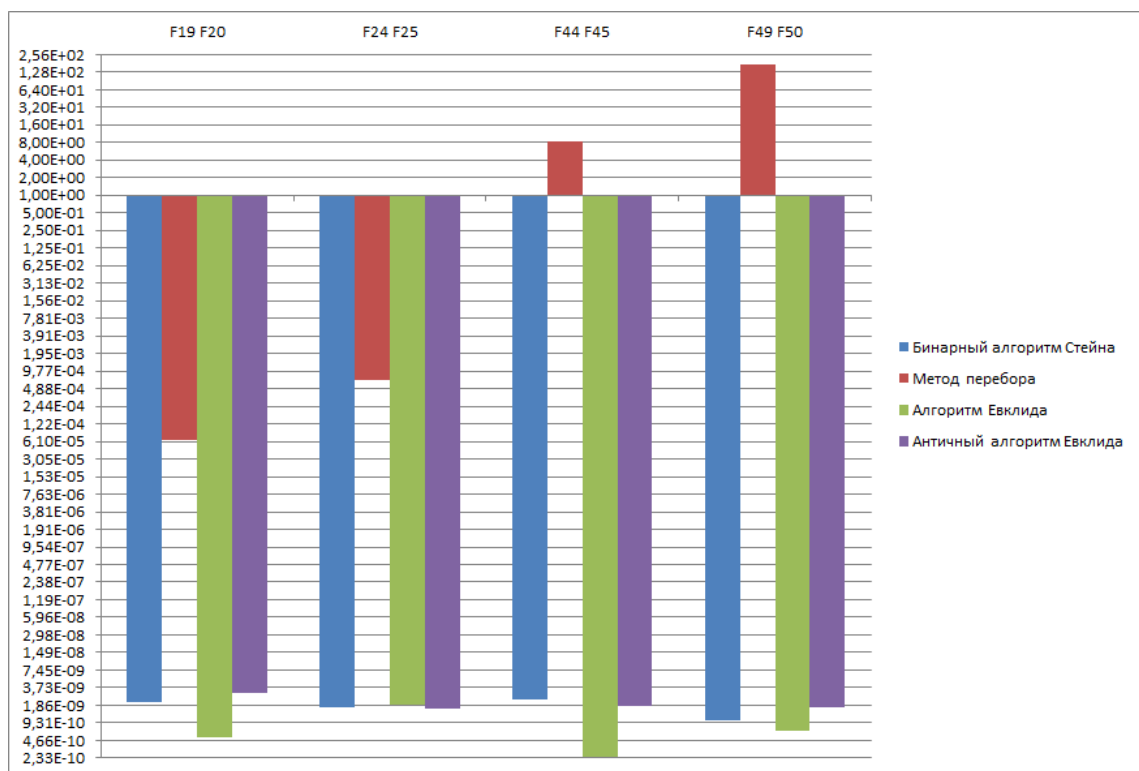


Рисунок 9 – время работы программ

## ЗАКЛЮЧЕНИЕ

В настоящей работе были рассмотрены понятия НОД и НОК, рассмотрены и изложены античный алгоритм Евклида (через разности), алгоритм Евклида (через остатки), метод перебора для нахождения НОД, бинарный алгоритм Стейна.

На языке C++ в среде Microsoft Visual Studio рассмотренные алгоритмы были реализованы. Для проведения сравнительного анализа была написана вспомогательная программа, реализующая алгоритм нахождения  $n$ -го числа Фибоначчи. Проведенные эксперименты показали, что алгоритм Евклида является наиболее эффективным из всех рассмотренных.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

- 1 *Кнут, Д. Э.* Искусство программирования Том 2 / Д. Э. Кнут. — М.: Вильямс, 2002.
- 2 *Курант, Р.* Что такое математика? / Р. Курант, Г. Роббинс. — М.: МЦНМО, 2000.
- 3 *Stein, J.* Computational problems associated with racah algebra / J. Stein // *Journal of Computational Physics*. — 1967. — Vol. 1. — P. 397.
- 4 *Герасименко, М. Ю.* 138. Последовательность Фибоначчи [Электронный ресурс]. — URL: <http://acm.sgu.ru/univer/problem.php?contest=0&problem=138> (Дата обращения 27.05.2013) Загл. с экр. Яз. рус.



## ПРИЛОЖЕНИЕ А

### Листинг античного алгоритма Евклида, написанного на Microsoft Visual Studio C++ 2010

Код приложения obalg.cpp.

```
1 // обычный алгоритм Евклида через вычитания
2 #include "stdafx.h"
3 #include <iostream>
4 #include <conio.h>
5 #include <math.h>
6 #include "windows.h"
7
8 using namespace std;
9
10 unsigned long long int gcd(unsigned long long int a, unsigned long long int b)
11 {
12     while (a && b) //пока оба числа не равны нулю
13         if (a >= b) //если a больше или равно b
14             a -= b; //вычитаем из a число b
15         else //иначе
16             b -= a; //вычитаем из b число a
17     return a | b; //возвращаем a + b
18 }
19
20 int main()
21 {
22     //объявление переменных, предназначенных
23     //для вычисления времени работы программы
24     LARGE_INTEGER start, finish, freq;
25
26     unsigned long long int a, b, nod;
27     //a,b - исходные числа
28     //nod - наибольший общий делитель
29     cin >> a >> b;
30
31     // запоминаем частоту операций
32
33     QueryPerformanceFrequency( &freq );
34
35     // засекаем начало работы алгоритма
```

```

36
37     QueryPerformanceCounter( &start );
38
39     nod = gcd(a,b);
40
41     cout << nod << endl;//наибольший общий делитель
42
43     cout << a*b/nod << endl;//наименьшее общее кратное
44
45     // time - время в секундах
46
47     double time = (finish.QuadPart - start.QuadPart) / (double)freq.QuadPart;
48
49     cout << time << endl;
50
51     getch();
52     return 0;
53 }

```

## ПРИЛОЖЕНИЕ Б

### Листинг алгоритма Евклида, написанного на Microsoft Visual Studio C++ 2010

Код приложения ostalg.cpp.

```
1 // обычный алгоритм Евклида через остатки
2 #include "stdafx.h"
3 #include <iostream>
4 #include <conio.h>
5 #include <math.h>
6 #include "windows.h"
7
8 using namespace std;
9
10 unsigned long long int gcd(unsigned long long int a, unsigned long long int b)
11 {
12     while (a && b) //пока оба числа не равны нулю
13         if (a >= b) //если a больше или равно b
14             a %= b; //a присваиваем остаток от деления его на b
15         else //иначе
16             b %= a; //b присваиваем остаток от деления его на a
17     return a | b; //возвращаем a + b
18 }
19
20 int main()
21 {
22     //объявление переменных, предназначенных
23     //для вычисления времени работы программы
24     LARGE_INTEGER start, finish, freq;
25
26     unsigned long long int a, b, nod;
27     //a,b - исходные числа
28     //nod - наибольший общий делитель
29     cin >> a >> b;
30
31     // запоминаем частоту операций
32
33     QueryPerformanceFrequency( &freq );
34
35     // засекаем начало работы алгоритма
```

```

36
37     QueryPerformanceCounter( &start );
38
39     nod = gcd(a,b);
40
41     cout << nod << endl;//наибольший общий делитель
42
43     cout << a*b/nod << endl;//наименьшее общее кратное
44
45     // time - время в секундах
46
47     double time = (finish.QuadPart - start.QuadPart) / (double)freq.QuadPart;
48
49     cout << time << endl;
50
51     getch();
52     return 0;
53 }

```

## ПРИЛОЖЕНИЕ В

### Листинг алгоритма для нахождения НОД методом перебора, написанного на Microsoft Visual Studio C++ 2010

Код приложения pralg.cpp.

```
1 // Алгоритм для нахождения НОД методом перебора
2 #include "stdafx.h"
3 #include <iostream>
4 #include <conio.h>
5 #include <math.h>
6 #include "windows.h"
7
8 using namespace std;
9
10 int main()
11 {
12     //объявление переменных, предназначенных
13     //для вычисления времени работы программы
14     LARGE_INTEGER start, finish, freq;
15
16     unsigned long long int a, b, gcd;
17     //a,b - исходные числа
18     //gcd - наибольший общий делитель
19
20     cin >> a >> b;
21
22     // запоминаем частоту операций
23
24     QueryPerformanceFrequency( &freq );
25
26     // засекаем начало работы алгоритма
27
28     QueryPerformanceCounter( &start );
29
30     if (a >= b) gcd = a; //выбираем max(a,b)
31     else gcd = b;
32
33
34     //пока a и b не кратны gcd, уменьшаем значение
35     //gcd на единицу
```

```

36     while (a % gcd != 0 || b % gcd != 0) gcd--;
37
38     // засекаем окончание работы алгоритма
39     QueryPerformanceCounter( &finish );
40
41
42     cout << gcd << endl;//наибольший общий делитель a и b
43
44     cout << a*b/gcd << endl;//наименьшее общее кратное a и b
45     // time - время в секундах
46
47     double time = (finish.QuadPart - start.QuadPart) / (double)freq.QuadPart;
48
49     cout << time << endl;
50
51     getch();
52     return 0;
53 }

```

## ПРИЛОЖЕНИЕ Г

### Листинг бинарного алгоритма Стейна, написанного на Microsoft Visual Studio C++ 2010.

Код приложения binalg.cpp.

```
1 // бинарный алгоритм Стейна
2 #include "stdafx.h"
3 #include <iostream>
4 #include <conio.h>
5 #include <math.h>
6 #include "windows.h"
7
8 using namespace std;
9
10 unsigned long long int gcd(unsigned long long int a, unsigned long long int b)
11 {
12     unsigned long long int buf, deg = 0;
13
14     if (a == 0 || b == 0) //если a или b равно нулю
15         return a | b;    //возвращаем a + b
16
17     while (((a | b) & 1) == 0) //пока a и b нечётны
18     {
19         deg++; //увеличиваем степень двойки на единицу
20         a >>= 1; //делим a и b нацело на 2
21         b >>= 1;
22     }
23
24     if (a) //если a не равно нулю
25         while ((a & 1) == 0) //пока a чётно
26             a >>= 1; //делим его на 2
27
28     while (b) //пока b не равно нулю
29     {
30         while ((b & 1) == 0) //пока b чётно
31             b >>= 1; //делим его на 2
32
33         if (a < b) //если a < b
34             b -= a; //вычитаем из b число a
35         else //иначе
```

```

36     {
37         buf = a - b; //сохраняем a - b
38         a = b;       //присваиваем a число b
39         b = buf;     //присваиваем b сохранённую разность
40     }
41     b >>= 1;        //делим b нацело на 2
42 }
43
44 //возвращаем a умноженное на 2 в степени deg
45 return (a << deg);
46 }
47
48 int main()
49 {
50     //объявление переменных, предназначенных
51     //для вычисления времени работы программы
52     LARGE_INTEGER start, finish, freq;
53
54     unsigned long long int a, b, nod;
55     //a,b - исходные числа
56     //nod - наибольший общий делитель
57
58     cin >> a >> b;
59
60
61     // запоминаем частоту операций
62
63     QueryPerformanceFrequency( &freq );
64
65     // засекаем начало работы алгоритма
66
67     QueryPerformanceCounter( &start );
68
69     nod = gcd(a,b);
70
71     cout << nod << endl; //наибольший общий делитель
72
73     cout << a*b/nod << endl; //наименьшее общее кратное
74
75
76     // time - время в секундах

```



```
77
78     double time = (finish.QuadPart - start.QuadPart) / (double)freq.QuadPart;
79
80     cout << time << endl;
81
82     getch();
83     return 0;
84 }
```

## ПРИЛОЖЕНИЕ Д

### Листинг программы для нахождения чисел Фибоначчи, написанного на Delphi 7.0

Код приложения fib.dpr.

```
1 program fib; // алгоритм для нахождения чисел Фибоначчи
2           // использующий длинную арифметику
3 {$APPTYPE CONSOLE}
4 uses
5   SysUtils;
6
7   Const MaxDig = 1000; // максимальная длина числа
8         Osn = 10000; // основание системы счисления
9   Type TLong = Array[0..MaxDig] Of LongInt;
10  Var i, j, n : LongInt;
11      a, b, c : TLong;
12
13  // процедура для сложения a и b
14  Procedure SumLongTwo(A, B : TLong; Var C : TLong);
15    Var i, k : LongInt;
16  Begin
17    FillChar(C, SizeOf (C), 0); // заполняем массив C нулями
18
19    // присваиваем k наибольшее из длин чисел a и b
20    If A[0] > B[0] Then k := A[0]
21    Else k := B[0];
22
23    For i := 1 To k Do // C присваиваем A + B
24      Begin
25        C[i+1] := (C[i] + A[i] + B[i]) Div Osn;
26        C[i] := (C[i] + A[i] + B[i]) Mod Osn;
27      End;
28
29    // уточняем длину числа C
30    If C[k+1] = 0 Then C[0] := k
31    Else C[0] := k + 1
32  End;
33
34  // процедура для вывода на экран длинного числа
35  Procedure WriteLong(Const A : TLong);
```

```

36  Var ls, s : String;
37      i : LongInt;
38  Begin
39      //преобразуем число 0sn Div 10
40      //в строку ls
41      Str(0sn Div 10, ls);
42
43      Write(A[A[0]]); {выводим старшие цифры числа}
44
45      For i := A[0] - 1 DownTo 1 Do
46          Begin
47              Str(A[i], s); {преобразуем число A[i] в строку s}
48
49              {дополняем незначащими нулями}
50              While Length(s) < Length(ls) Do s := '0' + s;
51
52              Write(s)
53          End;
54      End;
55
56  Begin
57      ReadLn(n);
58
59      //присваиваем a, b, c значение 1
60      a[0] := 1;  a[1] := 1;
61      b := a;  c := a;
62
63      //вычисляем n-ое число Фибоначчи
64      For i := 3 To n Do
65          Begin
66              SumLongTwo(a,b,c);
67              a := b;
68              b := c;
69          End;
70
71      //вывод n-го числа Фибоначчи
72      WriteLong(c);
73
74      ReadLn;
75  End.

```