

Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт цифрового развития
Кафедра инфокоммуникаций

ОТЧЕТ
По практической работе №2
Дисциплины «Алгоритмизация»

Выполнил:

Пустяков Андрей Сергеевич

2 курс, группа ИВТ-б-о-22-1,

09.03.01 «Информатика и
вычислительная техника (профиль)
«Программное обеспечение средств
вычислительной
техники и автоматизированных
систем», очная форма обучения

(подпись)

Руководитель практики:

Воронкин Р. А. кандидат технических
наук, доцент, доцент кафедры
инфокоммуникаций

(подпись)

Ставрополь, 2023 г.

Тема: Нахождение чисел Фибоначчи и НОД двух чисел.

Цель: рассмотреть примеры наивного алгоритма нахождения чисел Фибоначчи через рекурсию и сравнить его с улучшенным алгоритмом. Рассмотреть примеры наивного алгоритма нахождения НОД и улучшенный алгоритм Евклида и сравнить их.

Ход работы:

Задание 1.

Необходимо найти n-ое число Фибоначчи используя рекурсивный алгоритм.

Код программы наивного рекурсивного алгоритма нахождения числа Фибоначчи с измерением времени работы (рис. 1).

```
import time # Импортирование модуля времени
print("Введите n-ое число Фибоначчи, которое требуется вычислить:")
n = int(input())

3 usages new *
def fib_recursive(n):
    if n <= 1:
        return n
    else:
        return fib_recursive(n - 1) + fib_recursive(n - 2)

start = time.time() # Начало отсчета
print("F(", n, ") =", fib_recursive(n))
end = time.time() # Конец отсчета

print("Время выполнения алгоритма:", (end - start))
```

Рисунок 1 – Использование рекурсивной функции.

Блок-схема программы с использованием рекурсивного алгоритма (рис. 2).

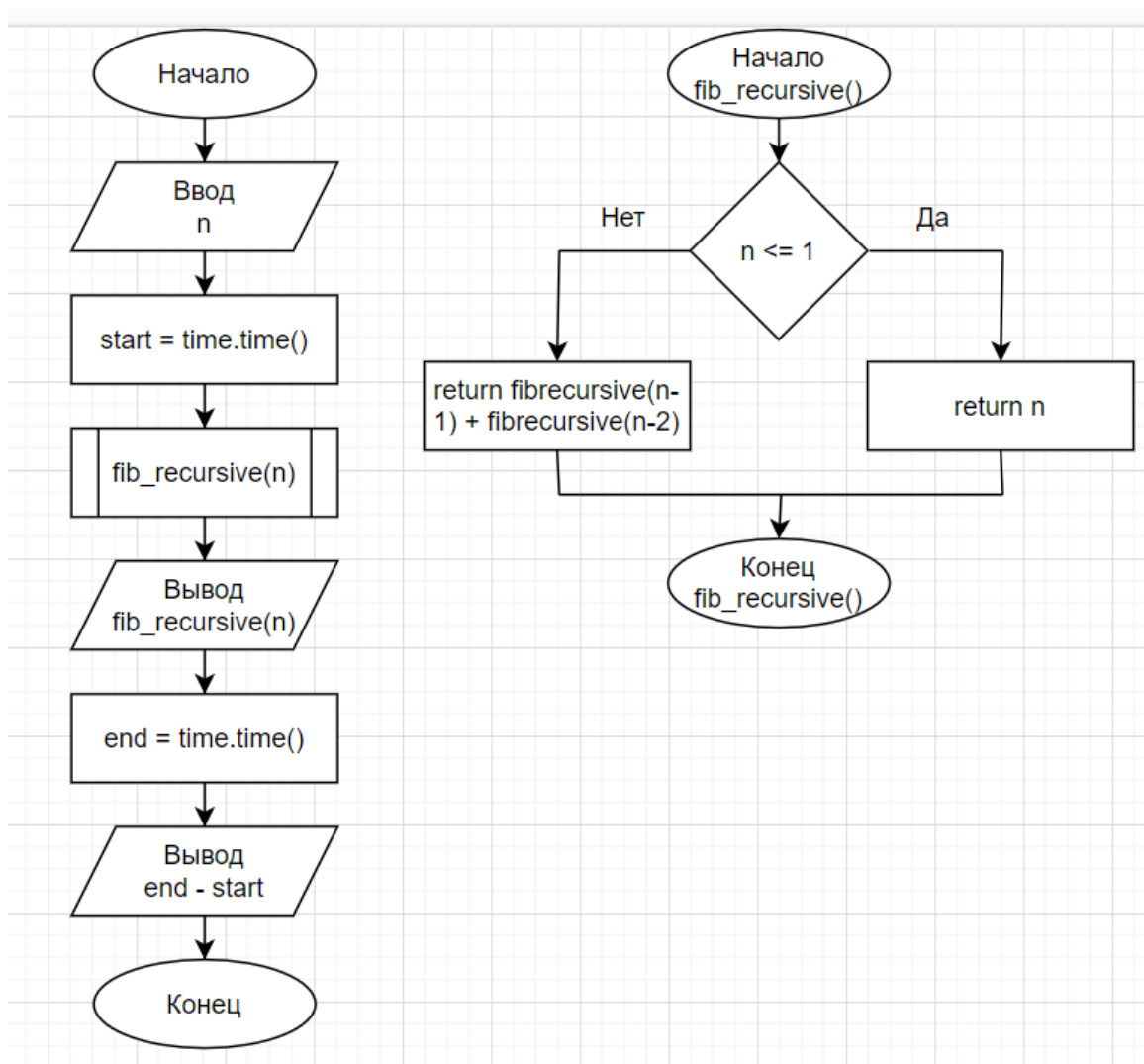


Рисунок 2 – Блок-схема с рекурсивной функцией.

Код программы улучшенного алгоритма нахождения чисел Фибоначчи (с использованием списка для записи промежуточных вычислений) (рис. 3).

```

1 import time # Импортирование модуля времени
2 print("Введите n-ое число Фибоначчи, которое требуется вычислить:")
3 n = int(input())
4
5 1 usage new *
6 def fib_list(n):
7     fib_numbers = [0]*100 # Пустой лист для записи чисел
8     fib_numbers[1] = 1
9     for i in range(2, n + 1):
10         fib_numbers[i] = fib_numbers[i - 1] + fib_numbers[i - 2]
11     print(fib_numbers)
12     return fib_numbers[n]
13
14 start = time.time() # Начало отсчета
15 print(n, "число Фибоначчи =", fib_list(n))
16 end = time.time() # Конец отсчета
17
18 print("Время выполнения алгоритма:", (end - start))

```

Рисунок 3 – Код программы улучшенного алгоритма.

Блок схема программы улучшенного алгоритма (рис. 4).

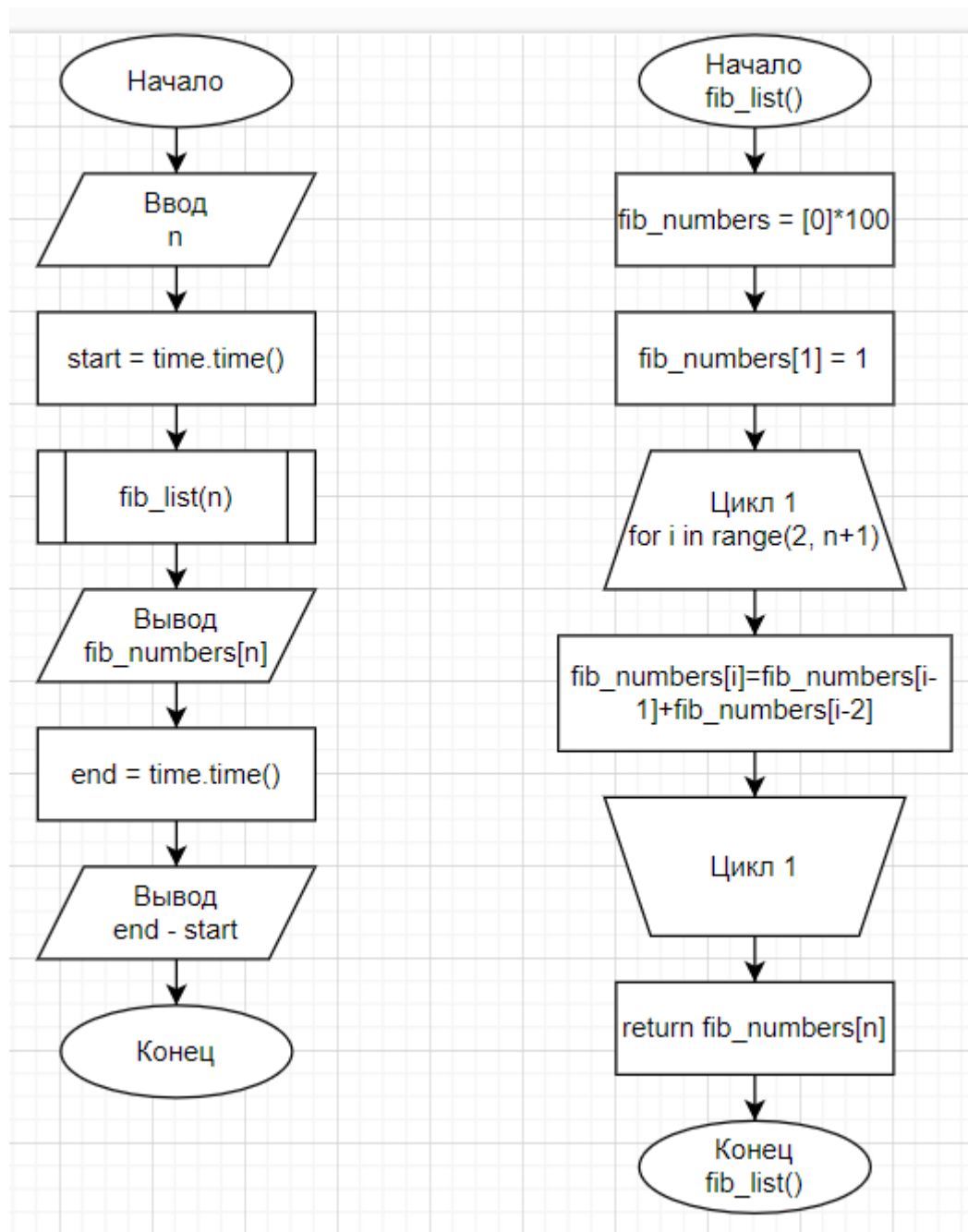


Рисунок 4 – Блок-схема программы улучшенного алгоритма.

График роста функций двух алгоритмов: наивного с рекурсией и улучшенного с использованием списка для записи чисел Фибоначчи (рис. 5).



Рисунок 5 – График сравнения скорости роста двух алгоритмов.

Скорость роста наивного алгоритма гораздо больше скорости роста улучшенного.

Задание 2.

Необходимо найти наибольший общий делитель двух неотрицательных целых чисел.

Код программы наивного алгоритма нахождения НОД двух неотрицательных целых чисел (рис. 6).

```
FibRecursive.py  FibList.py  Naive_GCD.py ×
1  import time # Импортирование модуля времени
2  print("Введите число a:")
3  number_a = int(input())
4  print("Введите число b:")
5  number_b = int(input())
6
7
8  def naive_gcd(a, b):
9      gcd = 1
10     for d in range(2, max(a, b)):
11         if a % d == 0 and b % d == 0:
12             gcd = d
13     return gcd
14
15
16 start = time.time() # Начало отсчета
17 print("НОД чисел a и b =", naive_gcd(number_a, number_b))
18 end = time.time() # Конец отсчета
19
20 print("Время выполнения алгоритма:", (end - start), "сек.")
21
```

Рисунок 6 – Код программы наивного алгоритма нахождения НОД.

Блок-схема наивного алгоритма нахождения НОД двух неотрицательных целых чисел (рис. 7).

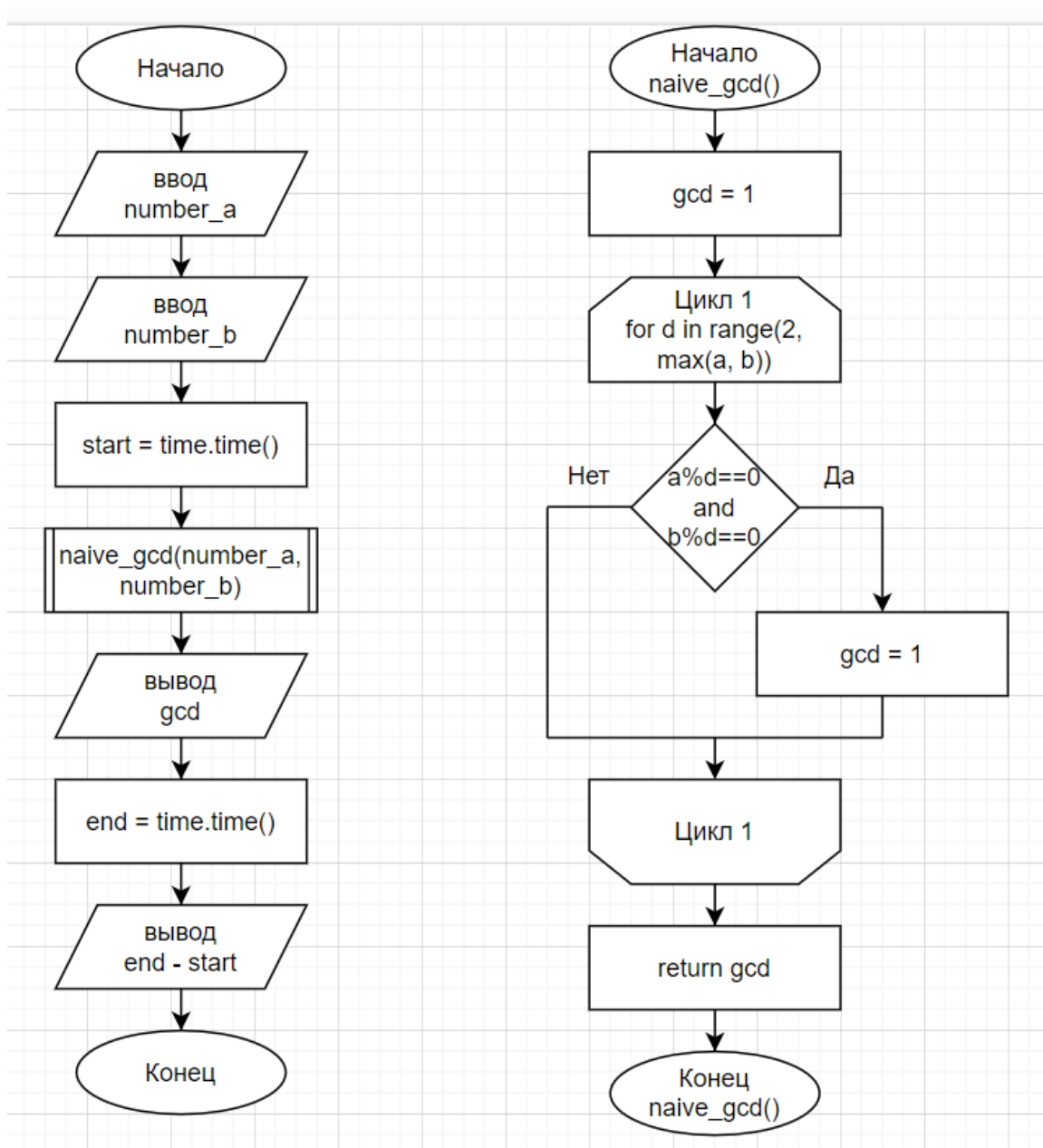
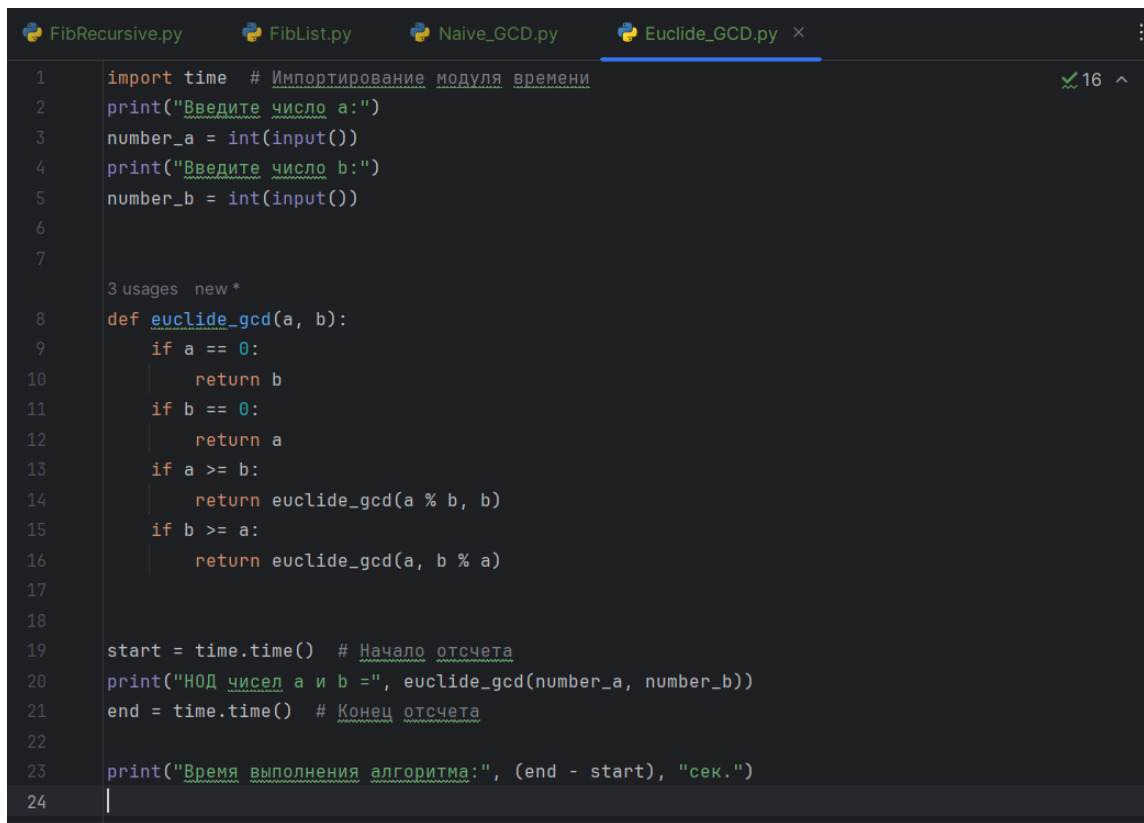


Рисунок 7 – Блок-схема наивного алгоритма нахождения НОД.

Код программы алгоритма Евклида нахождения НОД двух неотрицательных целых чисел (рис. 8).



```
1 import time # Импортирование модуля времени
2 print("Введите число a:")
3 number_a = int(input())
4 print("Введите число b:")
5 number_b = int(input())
6
7
8 3 usages new *
9 def euclidean_gcd(a, b):
10     if a == 0:
11         return b
12     if b == 0:
13         return a
14     if a >= b:
15         return euclidean_gcd(a % b, b)
16     if b >= a:
17         return euclidean_gcd(a, b % a)
18
19 start = time.time() # Начало отсчета
20 print("НОД чисел a и b =", euclidean_gcd(number_a, number_b))
21 end = time.time() # Конец отсчета
22
23 print("Время выполнения алгоритма:", (end - start), "сек.")
24 |
```

Рисунок 8 – Код программы алгоритма Евклида.

Блок-схема алгоритма Евклида для нахождения наибольшего общего делителя двух неотрицательных целых чисел (рис. 9).

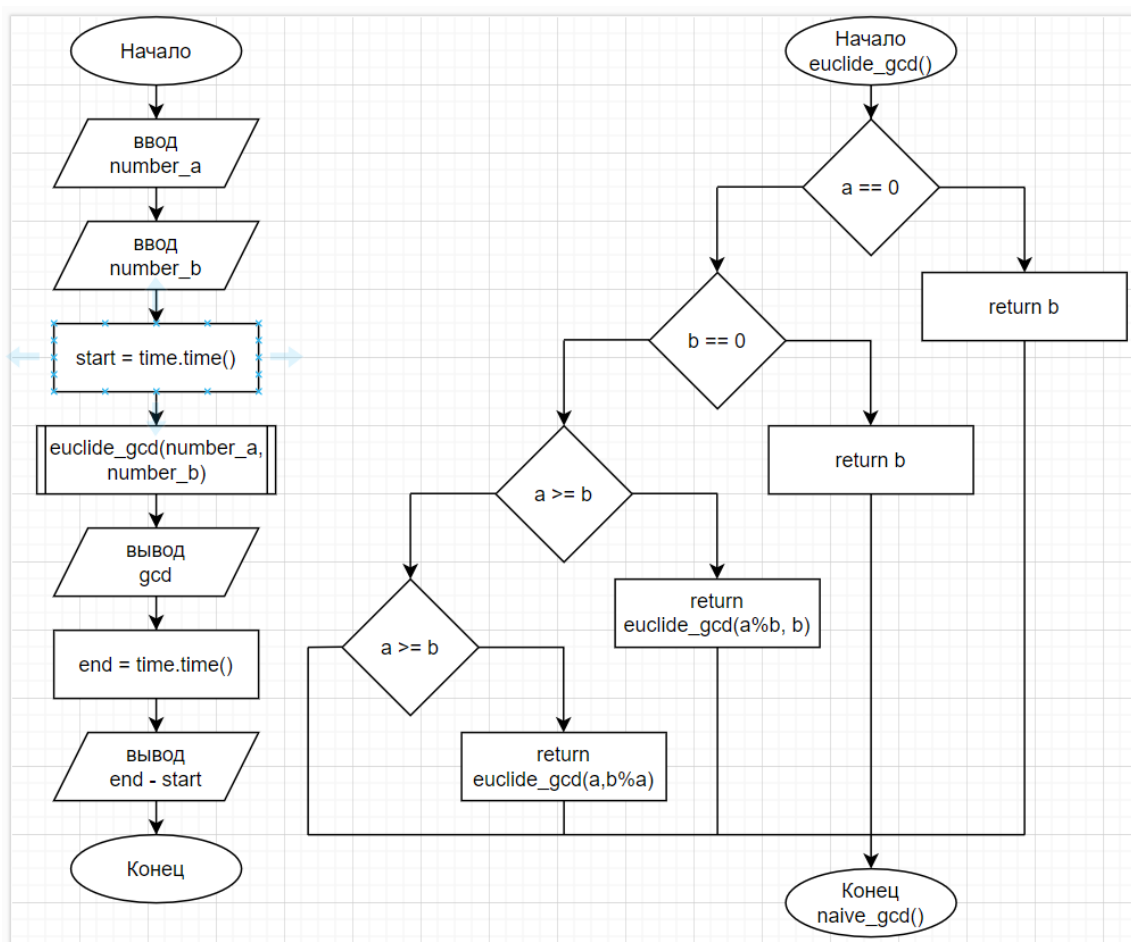


Рисунок 9 – Блок-схема алгоритма Евклида для нахождения НОД.

График скорости роста двух алгоритмов в зависимости от величины разрядности двух чисел, наибольший общий делитель которых необходимо вычислить (рис. 10).



Рисунок 10 – График оценки времени работы алгоритмов поиска НОД.

Скорость роста наивного алгоритма значительно возрастает в зависимости от разрядности двух исходных чисел в отличие от алгоритма Евклида, скорость работы которого зависит от величины чисел меньше.

Вывод: наивный алгоритм поиска чисел Фибоначчи работает гораздо дольше оптимизированного алгоритма. Большое время работы наивного рекурсивного алгоритма обуславливается большим количеством рекурсивных вызовов, которые возникают при относительно большом числе Фибоначчи, а быстрая скорость работы алгоритма с использованием массива обуславливается сохранением данных, которые были вычислены на предыдущих итерациях (единственный недостаток такого алгоритма – необходимо создавать большой список чисел Фибоначчи для относительно больших чисел Фибоначчи). Наивный алгоритм поиска НОД значительно уступает по скорости алгоритму Евклида. Это обуславливается тем, что при больших числах, НОД которых необходимо вычислить, итераций может происходить столько, на сколько велико большее из этих двух чисел, а в

алгоритме Евклида при каждом рекурсивном вызове одно из двух чисел сильно уменьшается, поэтому скорость работы слабо зависит от величины двух чисел. Время работы некоторых алгоритмов настолько велико, что использование таких алгоритмов неприемлемо для решения некоторых задач, таким образом очень важно выбирать алгоритмы, в которых происходит меньше итераций или рекурсивных вызовов. Наивные алгоритмы, которые первыми приходят на ум могут быть вполне корректны, но при этом они не всегда могут быть самыми оптимальными, а неочевидные алгоритмы могут быть оптимальнее наивных.