Практическая работа №1

Тема: «Алгоритмы поиска».

Цель работы: изучить алгоритмы поиска.

Поиск является одним из наиболее часто встречаемых действий в программировании. Существует множество различных алгоритмов поиска, которые принципиально зависят от способа организации данных. У каждого алгоритма поиска есть свои преимущества и недостатки. Поэтому важно выбрать тот алгоритм, который лучше всего подходит для решения конкретной задачи.

Поставим задачу поиска в линейных структурах. Пусть задано множество данных, которое описывается как массив, состоящий из некоторого количества элементов. Проверим, входит ли заданный ключ в данный массив. Если входит, то найдем номер этого элемента массива, то есть, определим первое вхождение заданного ключа (элемента) в исходном массиве.

Таким образом, определим общий алгоритм поиска данных:

Шаг 1. Вычисление элемента, что часто предполагает получение значения элемента, ключа элемента и т.д.

Шаг 2. Сравнение элемента с эталоном или сравнение двух элементов (в зависимости от постановки задачи).

Шаг 3. Перебор элементов множества, то есть прохождение по элементам массива.

Основные идеи различных алгоритмов поиска сосредоточены в методах перебора и стратегии поиска.

Рассмотрим основные алгоритмы поиска в линейных структурах более подробно.

Линейный (последовательный) поиск — алгоритм нахождения заданного значения произвольной функции на некотором отрезке. Данный алгоритм является простейшим алгоритмом поиска и в отличие, например, от двоичного поиска, не накладывает никаких ограничений на функцию и имеет простейшую реализацию. Поиск значения функции осуществляется простым сравнением очередного рассматриваемого значения (как правило поиск происходит слева направо, то есть от меньших значений аргумента к большим) и, если значения совпадают (с той или иной точностью), то поиск считается завершённым.

Алгоритм линейного поиска:

Шаг 1. Полагаем, что значение переменной цикла i=0.

Шаг 2. Если значение элемента массива x[i] равно значению ключа key, то возвращаем значение, равное номеру искомого элемента, и алгоритм завершает работу. В противном случае значение переменной цикла увеличивается на единицу i=i+1.

Шаг 3. Если i<k, где k – число элементов массива x, то выполняется Шаг 2, в противном случае – работа алгоритма завершена и возвращается значение равное -1.

```
def line_search(array, key):
    for i in range(len(array)):
        if array[i] == key:
            return i
```

Рис. 1 Линейный поиск

При наличии в массиве нескольких элементов со значением key данный алгоритм находит только первый из них (с наименьшим индексом).

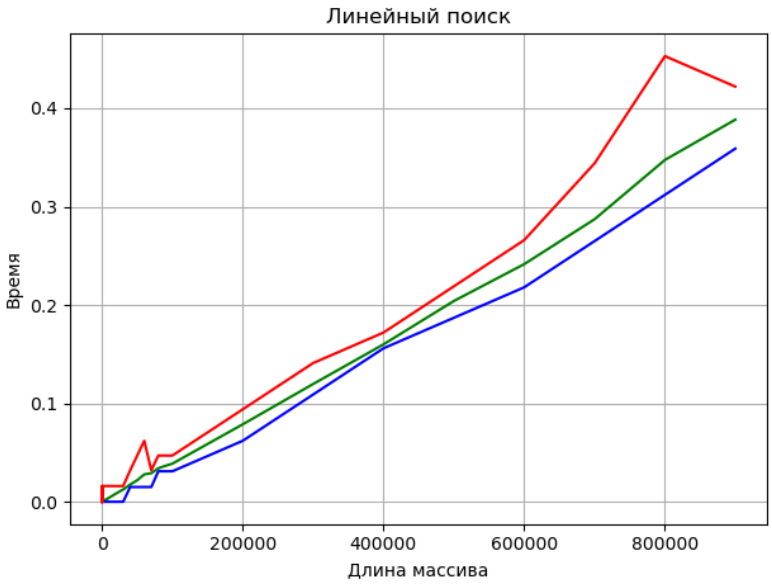

Рис. 2 График линейного поиска

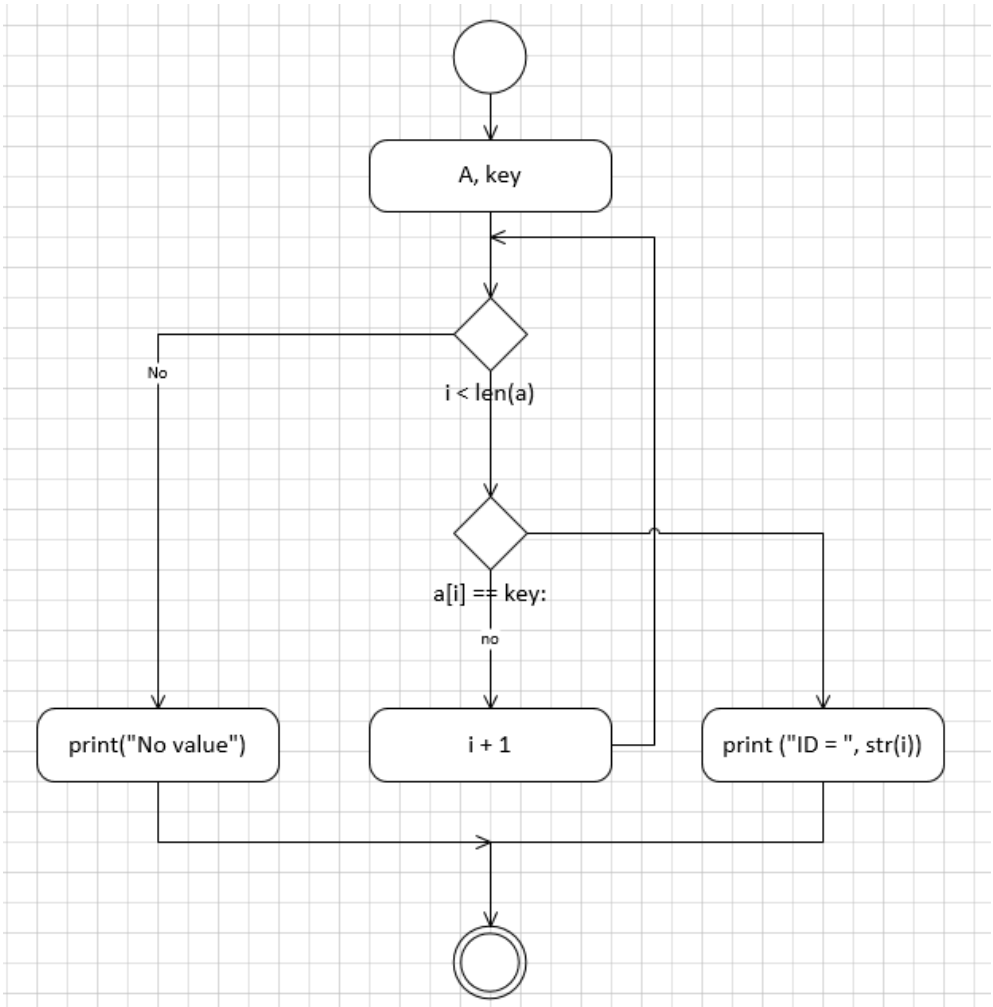

Рис. 3 Диаграмма деятельности линейного поиска

Двоичный (бинарный) поиск — классический алгоритм поиска элемента в отсортированном массиве (векторе). Используется в информатике, вычислительной математике и математическом программировании. Частным случаем двоичного поиска является метод бисекции, который применяется для поиска корней заданной непрерывной функции на заданном отрезке. Если у нас есть массив, содержащий упорядоченную последовательность данных, то очень эффективен двоичный поиск. Бинарный поиск позволяет найти данный элемент в отсортированном массиве или определить, что он не встречается в данном массиве за O(log n) действий, где n - количество элементов в массиве.

Алгоритм бинарного поиска:

1. Определение значения элемента в середине структуры данных. Полученное значение сравнивается с ключом.
2. Если ключ меньше значения середины, то поиск осуществляется в первой половине элементов, иначе – во второй.
3. Поиск сводится к тому, что вновь определяется значение серединного элемента в выбранной половине и сравнивается с ключом.
4. Процесс продолжается до тех пор, пока не будет найден элемент со значением ключа или не станет пустым интервал поиска.

```
def binar_search(array, key):

    value = key
    a = array

    mid = len(a) // 2
    low = 0
    high = len(a) - 1

    while a[mid] != value and low <= high:
        if value > a[mid]:
            low = mid + 1
        else:
            high = mid - 1
        mid = (low + high) // 2

    if low > high:
        return -1

    else:
        return mid
```

Рис. 4 Бинарный поиск

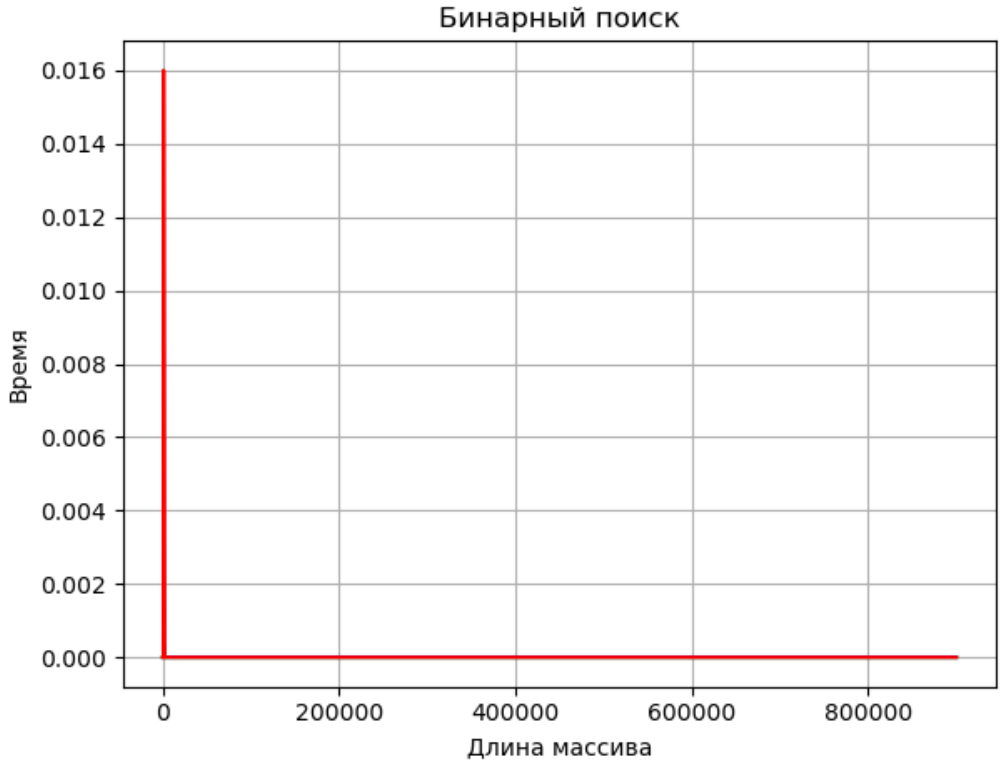

Рис. 5 График бинарного поиска

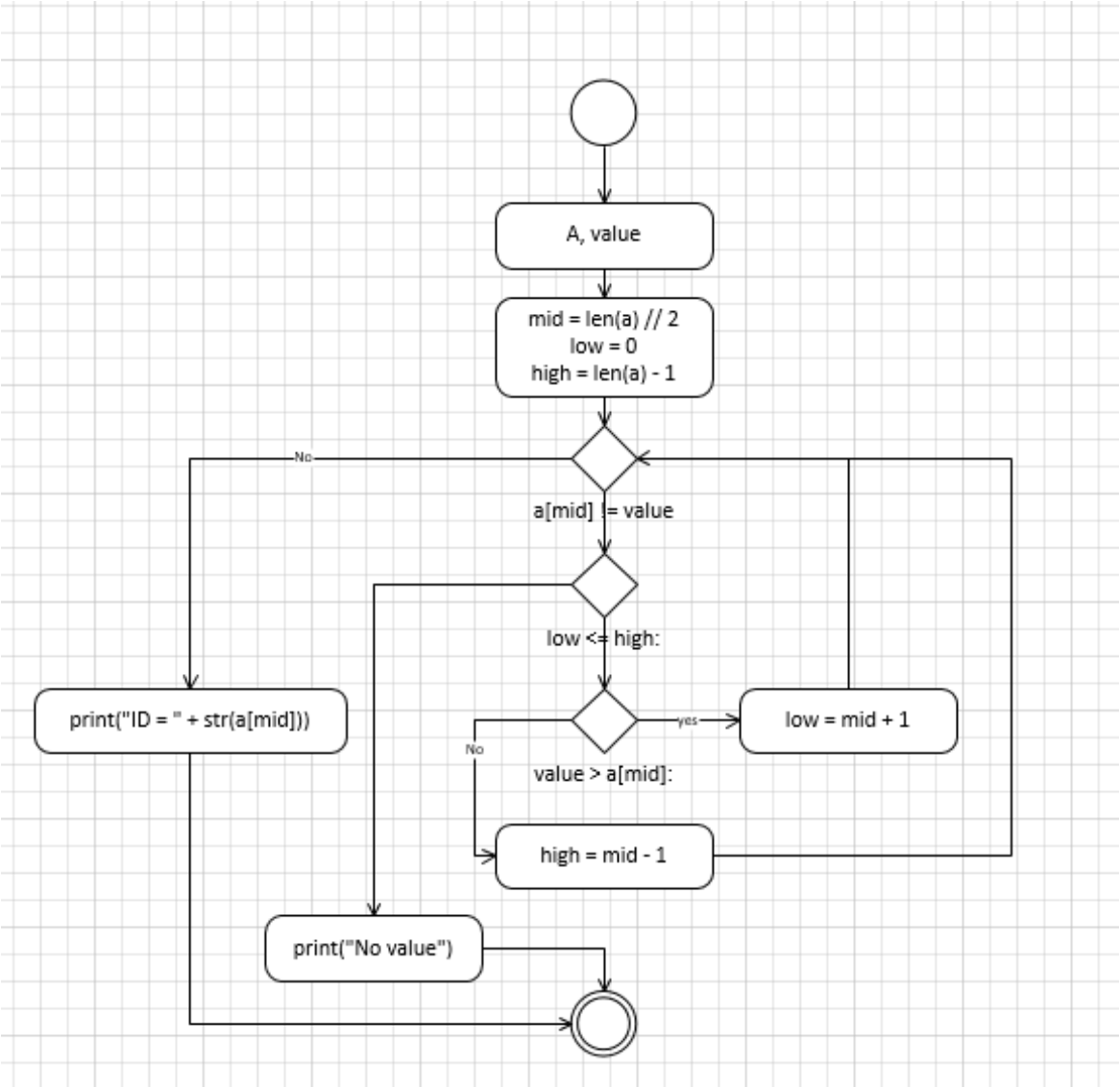

Рис. 3 Диаграмма деятельности бинарного поиска

Интерполяционный поиск

Этот алгоритм поиска работает на месте измерения требуемого значения. Для правильной работы этого алгоритма сбор данных должен быть отсортированным и равномерно распределенным. Первоначально позиция зонда – это позиция самого среднего элемента коллекции. Если совпадение происходит, то возвращается индекс элемента. Если средний элемент больше, чем элемент, то позиция зонда снова вычисляется в подмассиве справа от среднего элемента. В противном случае элемент ищется в подмассиве слева от среднего элемента. Этот

процесс продолжается и для подмассива, пока размер подмассива не уменьшится до нуля.

```
def inter_search(array, key):
    minimum = 0
    maximum = len(array) - 1
    ret = 0
    while array[minimum] < key < array[maximum]:
        mid = int(minimum + (maximum - minimum) * (key - array[minimum]) / (array[maximum]
- array[minimum]))
        if array[mid] == key:
            ret = mid
            break
        elif array[mid] > key:
            maximum = mid - 1
        else:
            minimum = mid + 1

    if array[minimum] == key:
        ret = minimum
    if array[maximum] == key:
        ret = maximum
    while ret > 0 and array[ret - 1] == key:
        ret -= 1
    if array[ret] == key:
        return ret
    else:
        return -1
```

Рис. 4 Интерполяционный поиск

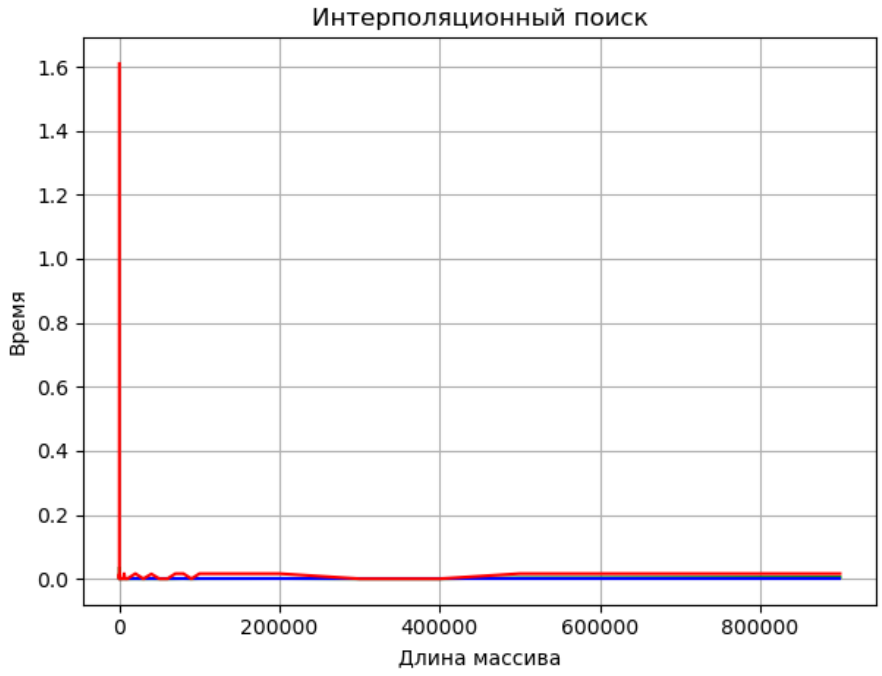

Рис. 5 График интерполяционного поиска

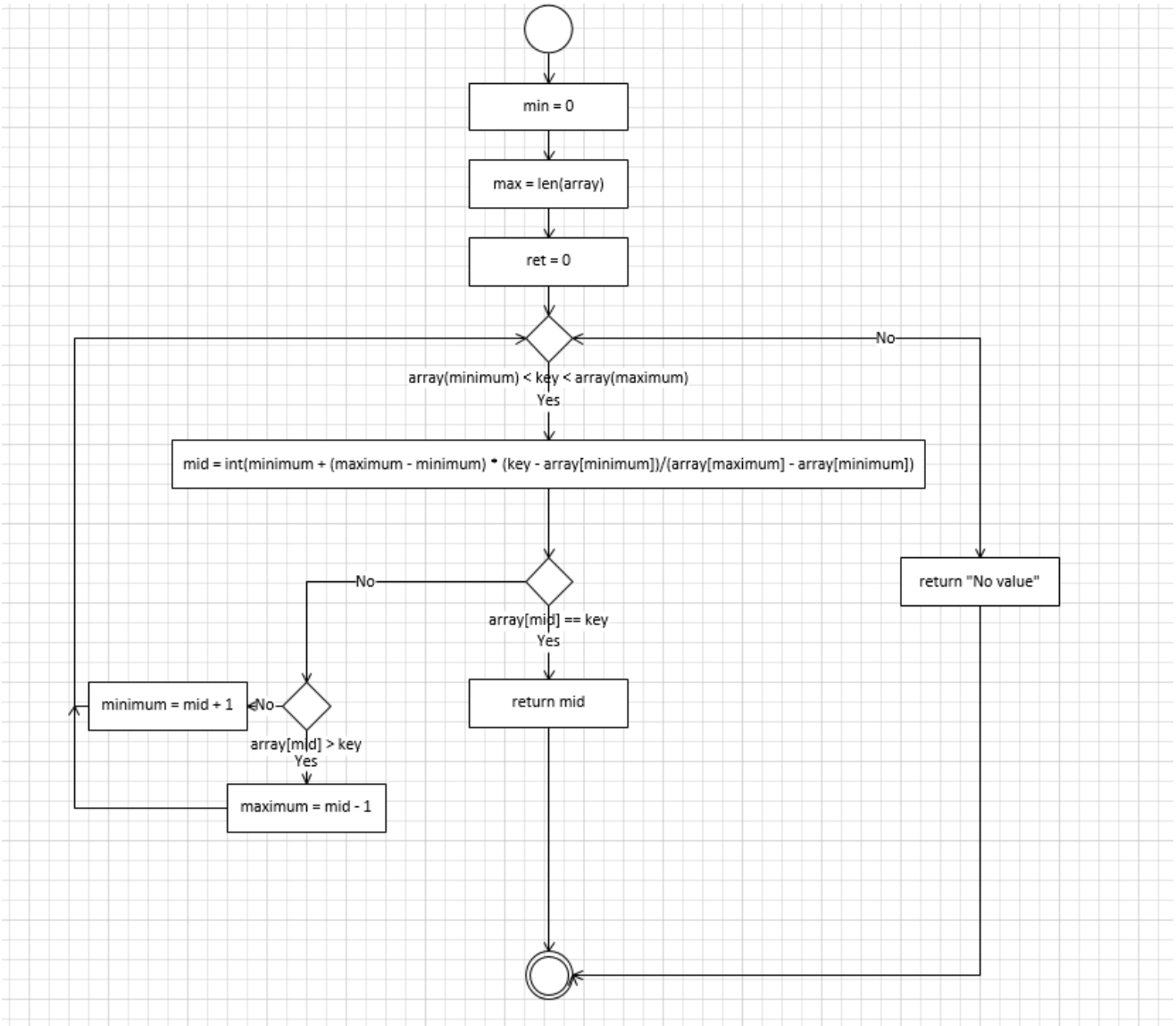

Рис. 6 Диаграмма деятельности интерполяционного поиска

Поиск Фибоначчи

Пусть искомый элемент будет x. Идея состоит в том, чтобы сначала найти наименьшее число Фибоначчи, которое больше или равно длине данного массива. Пусть найденное число Фибоначчи будет fib (m-е число Фибоначчи). Мы используем (m-2) -ое число Фибоначчи в качестве индекса (если это действительный индекс). Пусть (m-2) -ое число Фибоначчи будет i, мы сравним arr [i] с x, если x одно и то же, мы вернем i. Иначе, если x больше, мы возвращаемся для подмассива после i, иначе мы возвращаемся для подмассива до i.

Ниже приведен полный алгоритм

Пусть arr [0..n-1] будет входным массивом, а элемент для поиска будет x.

Найдите наименьшее число Фибоначчи, большее или равное n. Пусть это число будет fibM [m-е число Фибоначчи]. Пусть два предшествующих ему числа Фибоначчи — это fibMm1 [(m-1) -ое число Фибоначчи] и fibMm2 [(m-2) -ое число Фибоначчи].

В то время как массив имеет элементы для проверки:

Сравните x с последним элементом диапазона, охватываемого fibMm2

Если x совпадает, вернуть индекс

Иначе Если x меньше, чем элемент, переместите три переменные Фибоначчи на две Фибоначчи вниз, что указывает на удаление приблизительно двух третей оставшегося массива.

Если x больше элемента, переместите три переменные Фибоначчи на одну Фибоначчи вниз. Сброс смещения до индекса. Вместе они указывают на устранение приблизительно первой трети оставшегося массива.

Поскольку для сравнения может остаться один элемент, проверьте, не равняется ли fibMm1 1. Если да, сравните x с этим оставшимся элементом. Если совпадают, верните индекс.

```
def fib_search(array, key):
    x = key
    arr = array
    n = len(array)
    fibMMm2 = 0
    fibMMm1 = 1
    fibM = fibMMm2 + fibMMm1
    while (fibM < n):
        fibMMm2 = fibMMm1
        fibMMm1 = fibM
        fibM = fibMMm2 + fibMMm1
    offset = -1;
    while (fibM > 1):
        i = min(offset+fibMMm2, n-1)
        if (arr[i] < x):
            fibM = fibMMm1
            fibMMm1 = fibMMm2
            fibMMm2 = fibM - fibMMm1
            offset = i
        elif (arr[i] > x):
            fibM = fibMMm2
            fibMMm1 = fibMMm1 - fibMMm2
            fibMMm2 = fibM - fibMMm1
        else:
            return i
    if (fibMMm1 and arr[offset+1] == x):
        return offset+1;
    return -1
```

Рис. 7 Поиск Фибоначчи.

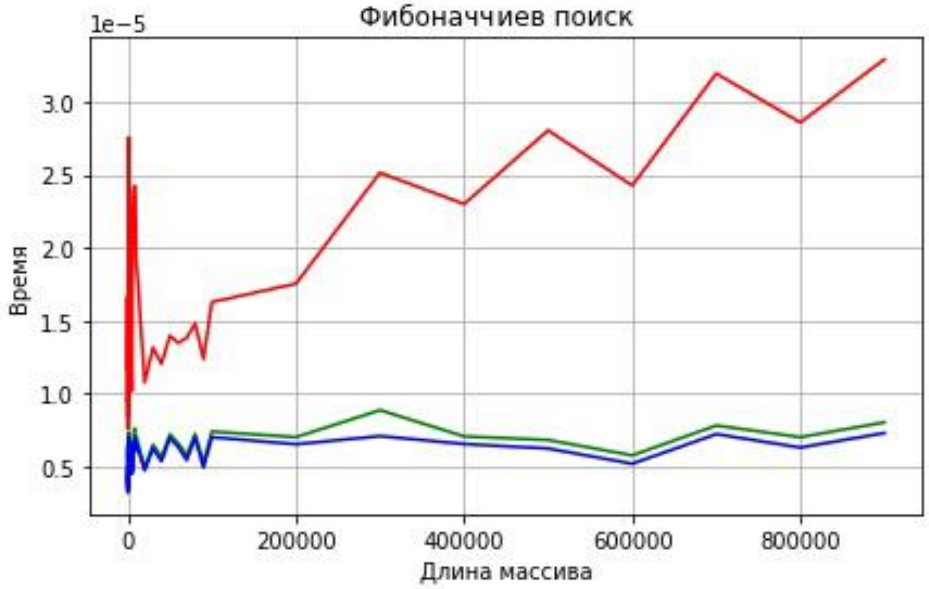

Рис. 8 График поиска Фибоначчи

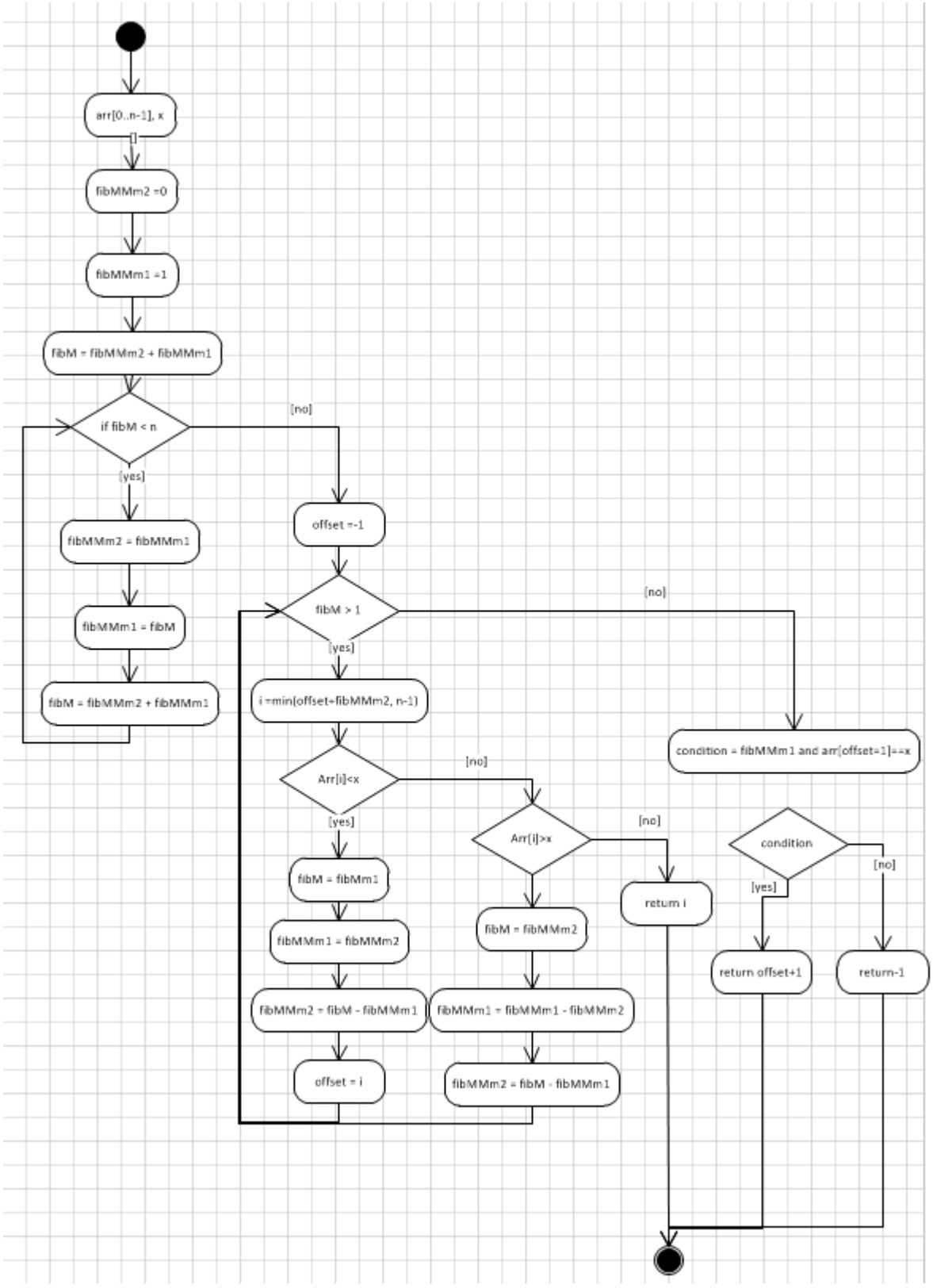

Рис. 9 Диаграмма деятельности поиска Фибоначчи

Вывод: в ходе выполнения работы были изучены алгоритмы поиска.