Практическая работа №1.

Тема: «Алгоритмы поиска».

Цель работы: изучить алгоритмы поиска.

Одним из важнейших действий со структурированной информацией является поиск. **Поиск** – процесс нахождения конкретной информации в ранее созданном множестве данных. Обычно данные представляют собой записи, каждая из которых имеет хотя бы один ключ. *Ключ поиска* – это поле записи, по значению которого происходит поиск. Ключи используются для отличия одних записей от других. Целью поиска является нахождение всех записей (если они есть) с данным значением ключа.

Структуру данных, в которой проводится поиск, можно рассматривать как *таблицу символов* (таблицу имен или таблицу идентификаторов) – структуру, содержащую ключи и данные, и допускающую две операции – вставку нового элемента и возврат элемента с заданным ключом. Иногда таблицы символов называют *словарями* по аналогии с хорошо известной системой упорядочивания слов в алфавитном порядке: слово – ключ, его толкование – данные.

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. Вычислительная сложность алгоритма $O(n)$.

В связи с малой эффективностью по сравнению с другими алгоритмами линейный поиск обычно используют, только если отрезок поиска содержит очень мало элементов, тем не менее, линейный поиск не требует дополнительной памяти или обработки/анализа функции, так что может работать в потоковом режиме при непосредственном получении данных из любого источника. Также линейный поиск часто используется в виде линейных алгоритмов поиска максимума/минимума.

В качестве примера можно рассмотреть поиск значения функции на множестве целых чисел, представленной таблично.

Диаграмма деятельностей представлена на рисунке 1, а исходный код на языке Python представлен на листинге 1.

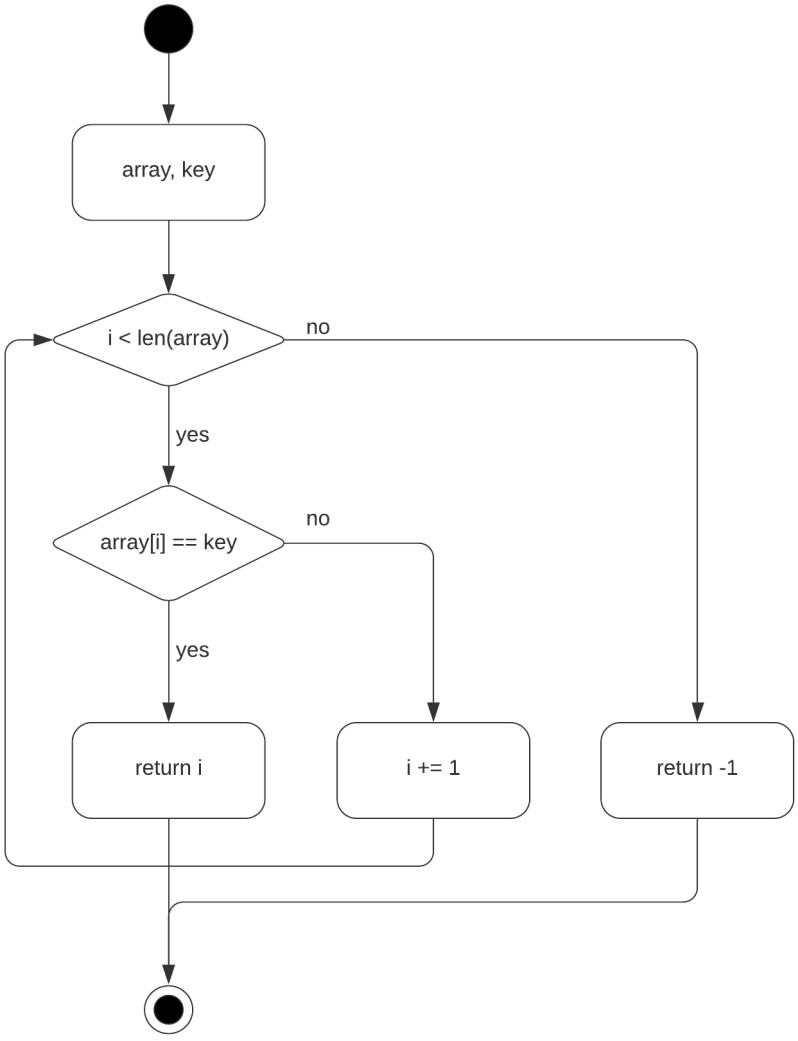

Рисунок 1. Диаграмма деятельностей для линейного поиска.

Листинг 1. Линейный поиск на языке Python.

```
def linear_search(array, key):
    for i in range(len(array)):
        if array[i] == key:
            return i
        else:
            return -1
```

В качестве исходного ключа для этого и последующих поисков принято значение key = 51 и длина массива от 100 до 1000000, по 50 итераций для каждой длины. Массивы отсортированы по возрастанию. Красной линией обозначено максимальное время поиска из 50 итераций, синей – минимальное, зеленой – среднее время поиска за 50 итераций. Результаты времени поиска представлены на рисунке 2.

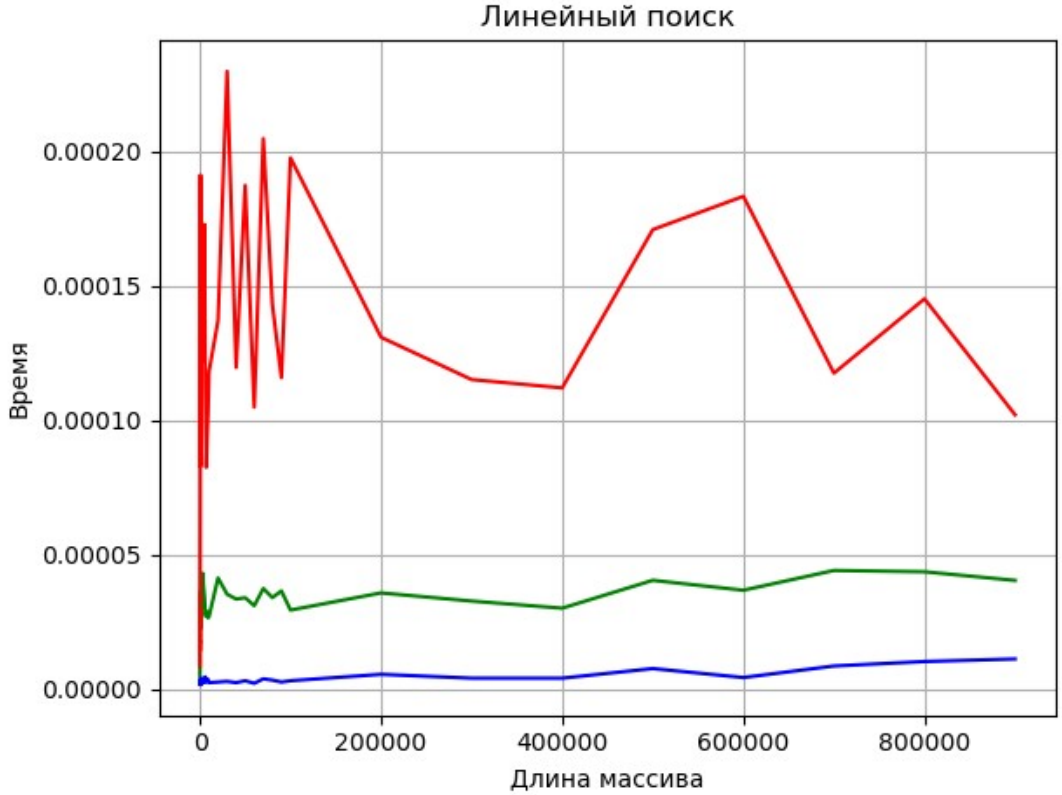

Рисунок 2. Результаты времени поиска для линейного поиска.

Двоичный (бинарный) поиск (также известен как **метод деления пополам** или **дихотомия**) — классический алгоритм поиска элемента в отсортированном массиве (векторе), использующий дробление массива на половины. Используется в информатике, вычислительной математике и математическом программировании. Вычислительная сложность алгоритма $O(\log n)$.

Частным случаем двоичного поиска является метод бисекции, который применяется для поиска корней заданной непрерывной функции на заданном отрезке.

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 для поиска.

Диаграмма деятельностей представлена на рисунке 3, а исходный код на языке Python представлен на листинге 2.

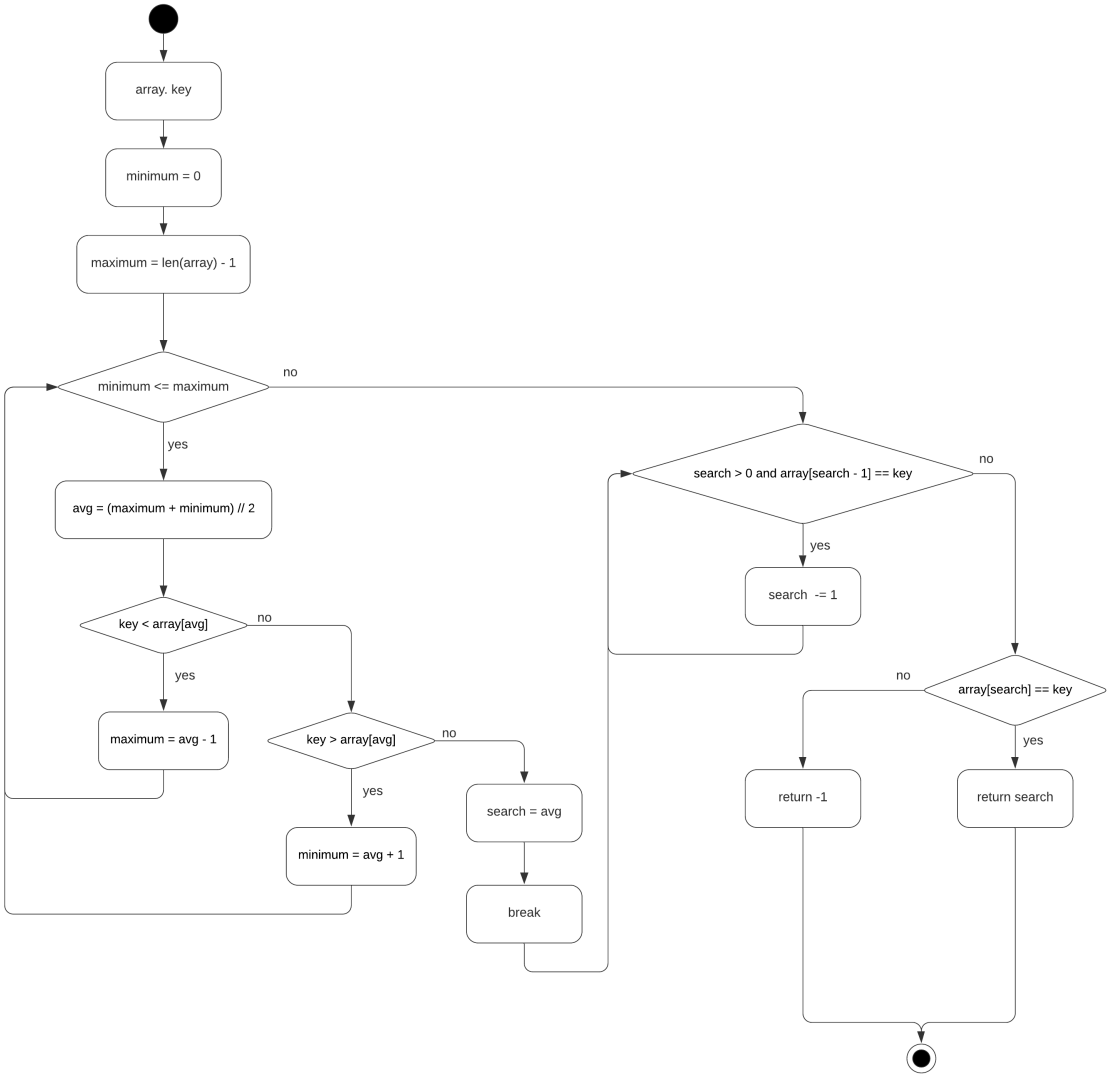

Рисунок 3. Диаграмма деятельностей для бинарного поиска.

Листинг 2. Бинарный поиск на языке Python.

```
def binary_search(array, key):
    minimum = 0
    maximum = len(array) - 1
    search = 0
    while minimum <= maximum:
        avg = (maximum + minimum) // 2
        if key < array[avg]:
            maximum = avg - 1
        elif key > array[avg]:
            minimum = avg + 1
```

```
        else:
            search = avg
            break
    while search > 0 and array[search - 1] == key:
        search -= 1
    if array[search] == key:
        return search
    else:
        return -1
```

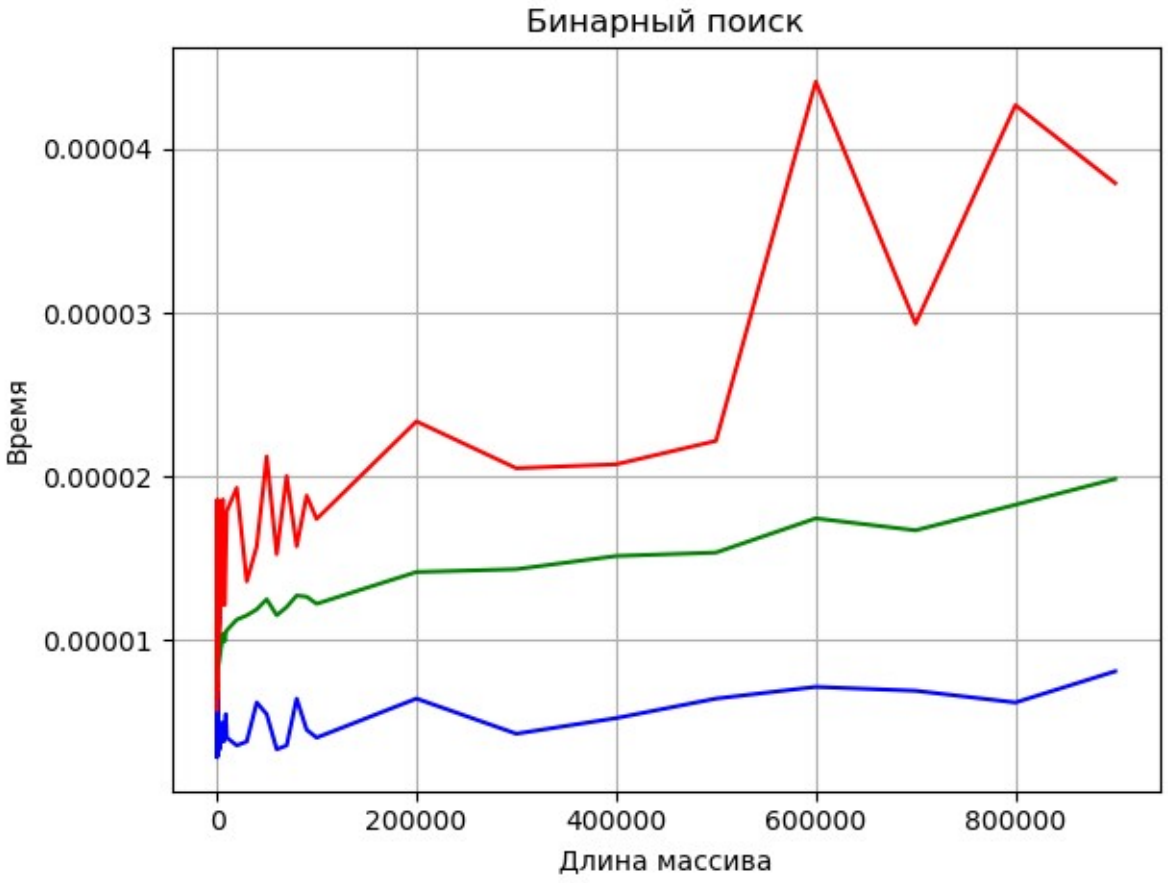

Рисунок 4. Результаты времени поиска для бинарного поиска.

Интерполяционный поиск (**интерполирующий поиск**) основан на принципе поиска в телефонной книге или, например, в словаре. Вместо сравнения каждого элемента с искомым, как при линейном поиске, данный алгоритм производит предсказание местонахождения элемента: поиск происходит подобно двоичному поиску, но вместо деления области поиска на две части, интерполирующий поиск производит оценку новой области поиска по расстоянию между ключом и текущим значением элемента. Другими словами, бинарный поиск учитывает лишь знак разности между ключом и текущим значением, а интерполирующий ещё учитывает и модуль этой разности и по данному значению производит предсказание позиции следующего элемента для проверки. В среднем интерполирующий поиск производит log(log(*N*)) операций, где *N* есть число элементов, среди которых

производится поиск. Число необходимых операций зависит от равномерности распределения значений среди элементов. В плохом случае (например, когда значения элементов экспоненциально возрастают) интерполяционный поиск может потребовать до O(*N*) операций.

Исходное множество должно быть упорядочено по возрастанию весов. Первоначальное сравнение осуществляется на расстоянии шага d, который определяется по формуле:

$$d = \left[\frac{(j-i)(K-K_i)}{K_j-K_i}\right],$$

где i - номер первого рассматриваемого элемента;

j - номер последнего рассматриваемого элемента;

K - отыскиваемый ключ;

K_i, K_j - значения ключей в позициях i и j;

[] - целая часть числа.

Идея метода заключается в следующем: шаг d меняется после каждого этапа по формуле, приведенной выше.

Диаграмма деятельностей представлена на рисунке 5, а исходный код на языке Python представлен на листинге 3.

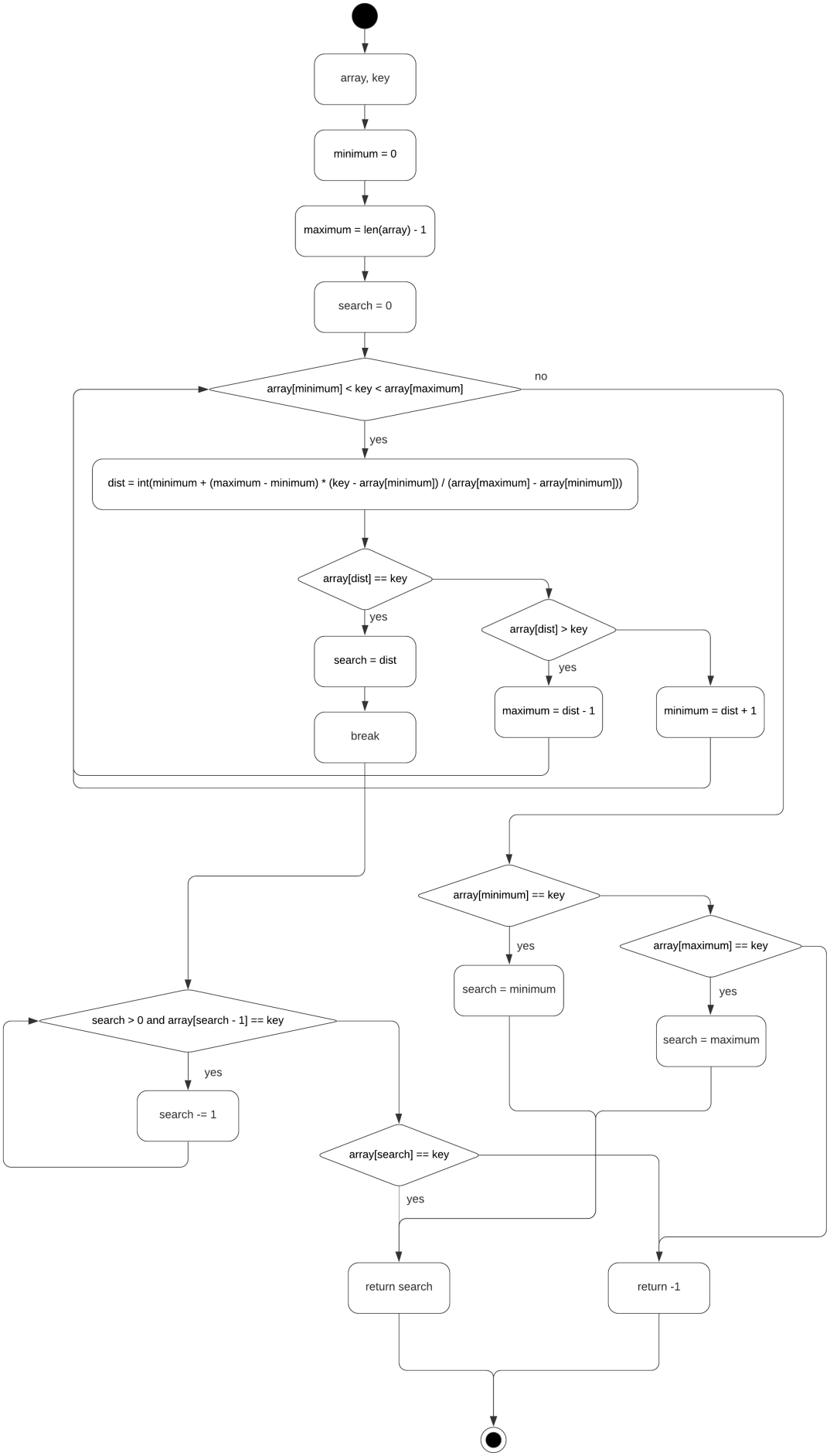

Рисунок 5. Диаграмма деятельностей для интерполяционного поиска.

Листинг 3. Интерполяционный поиск на языке Python.

```
def interpolational_search(array, key):
    minimum = 0
    maximum = len(array) - 1
    search = 0
    while array[minimum] < key < array[maximum]:
        dist = int(minimum + (maximum - minimum) * (key -
array[minimum]) / (array[maximum] - array[minimum]))
        if array[dist] == key:
            search = dist
            break
        elif array[dist] > key:
            maximum = dist - 1
        else:
            minimum = dist + 1
    if array[minimum] == key:
        search = minimum
    elif array[maximum] == key:
        search = maximum
    while search > 0 and array[search - 1] == key:
        search -= 1
    if array[search] == key:
        return search
    else:
        return -1
```

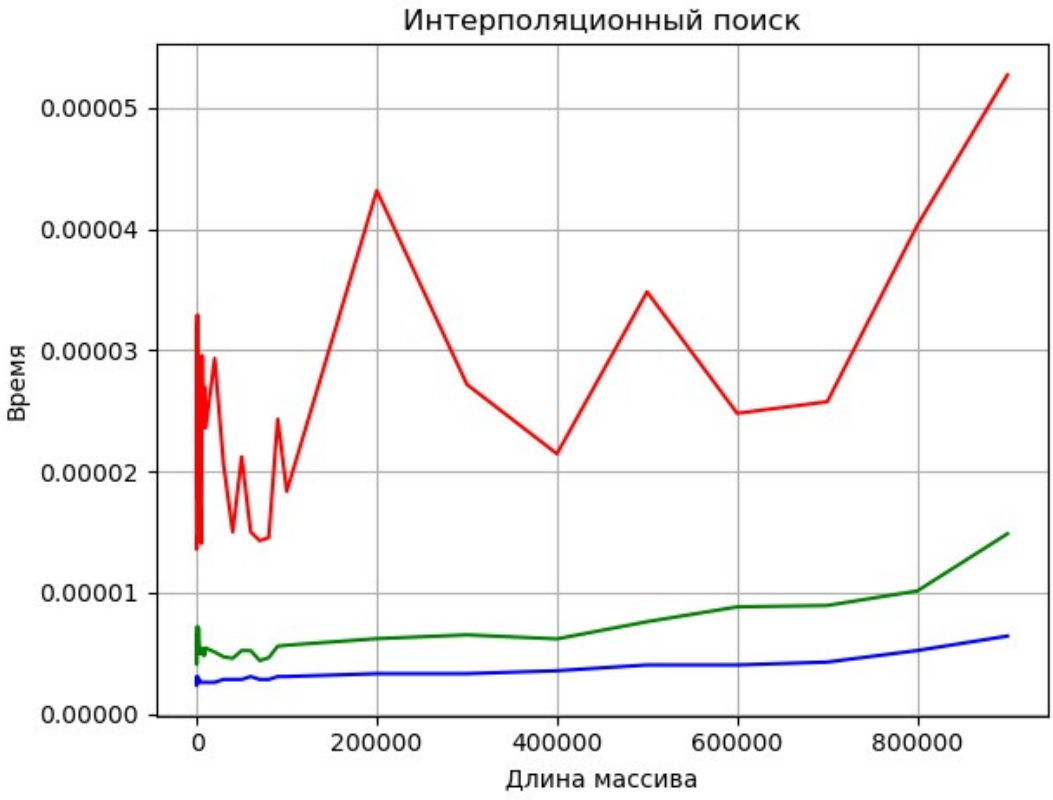

Рисунок 6. Результаты времени поиска для интерполяционного поиска.

В компьютерной науке **метод поиска Фибоначчи** – это метод поиска отсортированного массива с использованием алгоритма "разделяй и властвуй", который сужает возможные местоположения с помощью чисел

Фибоначчи. По сравнению с бинарным поиском, где сортированный массив делится на две равные по размеру части, одна из которых рассматривается далее, поиск Фибоначчи делит массив на две части, которые имеют размеры, являющиеся последовательными числами Фибоначчи. В среднем это приводит к примерно 4% приросту количества выполняемых сравнений, но имеет то преимущество, что для вычисления индексов доступных элементов массива требуется только сложение и вычитание, в то время как классический двоичный поиск требует битового сдвига, деления или умножения-операций, которые были менее распространены в то время, когда был впервые опубликован Поиск Фибоначчи. Поиск Фибоначчи имеет среднюю и наихудшую сложность O(log n).

Диаграмма деятельностей представлена на рисунке 7, а исходный код на языке Python представлен на листинге 4.

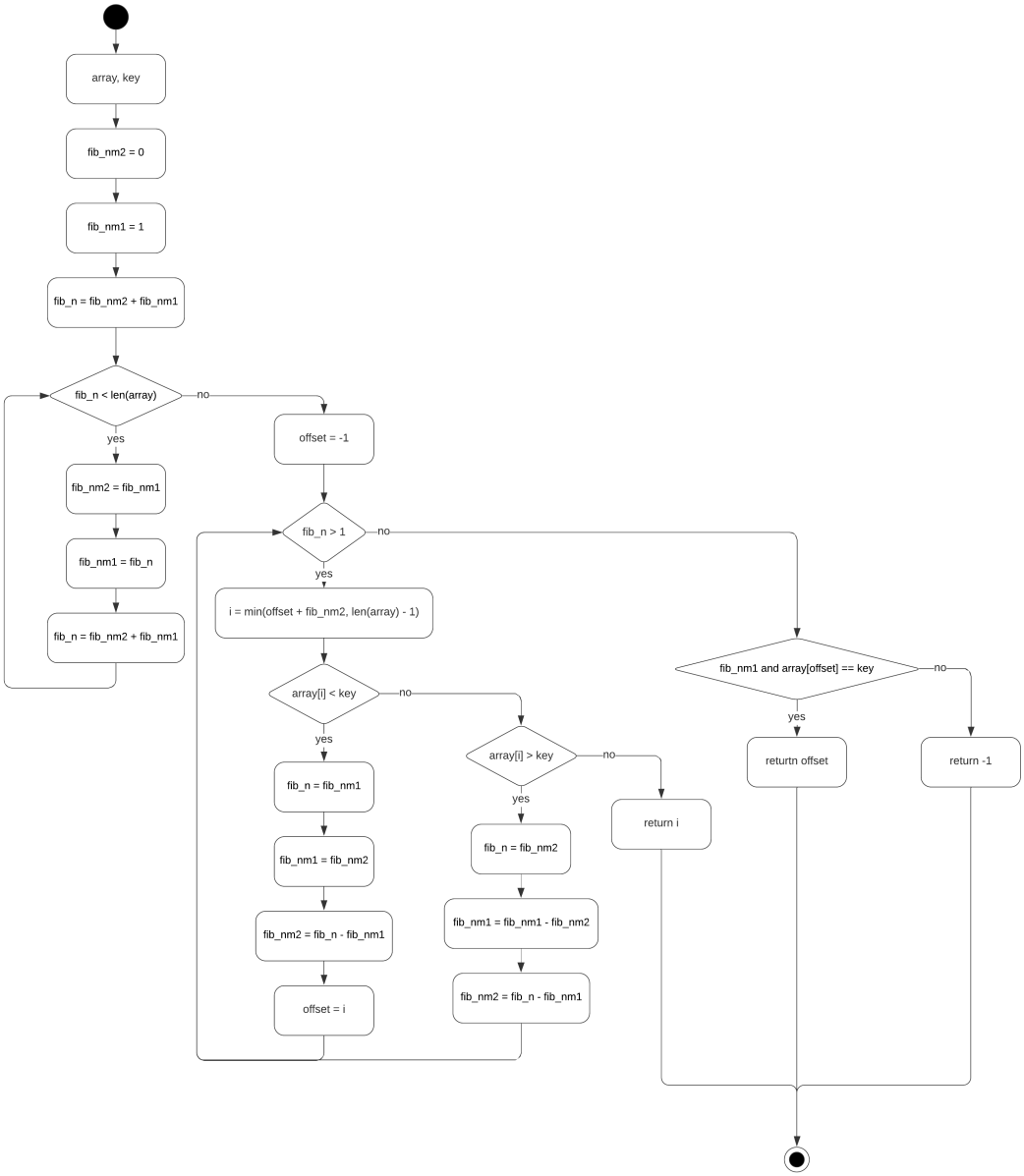

Рисунок 7. Диаграмма деятельностей для Фибоначчиева поиска.

Листинг 4. Фибоначчиев поиск на языке Python.

```
def fibonaccian_search(array, key):
    fib_nm2 = 0
```

```
    fib_nm1 = 1
    fib_n = fib_nm2 + fib_nm1
    while fib_n < len(array):
        fib_nm2 = fib_nm1
        fib_nm1 = fib_n
        fib_n = fib_nm2 + fib_nm1
    offset = -1
    while fib_n > 1:
        i = min(offset + fib_nm2, len(array) - 1)
        if array[i] < key:
            fib_n = fib_nm1
            fib_nm1 = fib_nm2
            fib_nm2 = fib_n - fib_nm1
            offset = i
        elif array[i] > key:
            fib_n = fib_nm2
            fib_nm1 = fib_nm1 - fib_nm2
            fib_nm2 = fib_n - fib_nm1
        else:
            return i
    if fib_nm1 and array[offset] == key:
        return offset
    return -1
```

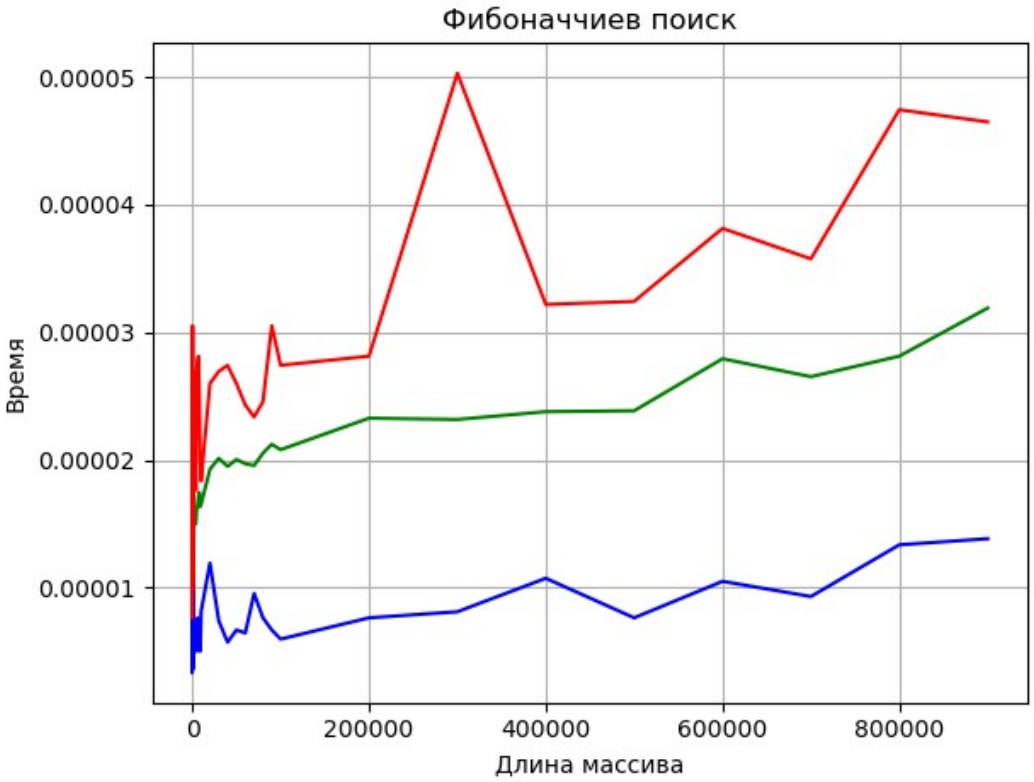

Рисунок 8. Результаты времени поиска для Фибоначчиева поиска.

Вывод: в ходе выполнения данной практической работы были изучены алгоритмы линейного, бинарного и интерполяционного поисков.