Практическая работа №1

Тема: Алгоритмы поиска.

Цель работы: Изучить алгоритмы поиска, построить графики зависимости времени от начального, среднего и конечного элемента в массиве.

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.

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.

Последовательный (линейный) поиск.

Последовательный (линейный) поиск – это простейший вид поиска заданного элемента на некотором множестве, осуществляемый путем последовательного сравнения очередного рассматриваемого значения с искомым до тех пор, пока эти значения не совпадут.

Идея этого метода заключается в следующем. Множество элементов просматривается последовательно в некотором порядке, гарантирующем, что будут просмотрены все элементы множества (например, слева направо). Если в ходе просмотра множества будет найден искомый элемент, просмотр прекращается с положительным результатом; если же будет просмотрено все множество, а элемент не будет найден, алгоритм должен выдать отрицательный результат.

Алгоритм последовательного поиска:

```
def search_posled(massive, x):
    nX = -1
for i in range(len(massive)):
    if massive[i]==x:
        nX = i
            break
if nX != -1:
    print("Нашли под номером",nX)
else:
    print("Не нашли")
```

На рисунке 1 представим диаграмму деятельностей для алгоритма последовательного поиска. Диаграмма деятельности – UML-диаграмма, на которой показаны действия, состояния которых описано на диаграмме состояний. Под деятельностью понимается спецификация исполняемого поведения в виде координированного последовательного и параллельного выполнения подчинённых элементов – вложенных видов деятельности и

отдельных действий, соединённых между собой потоками, которые идут от выходов одного узла ко входам другого.

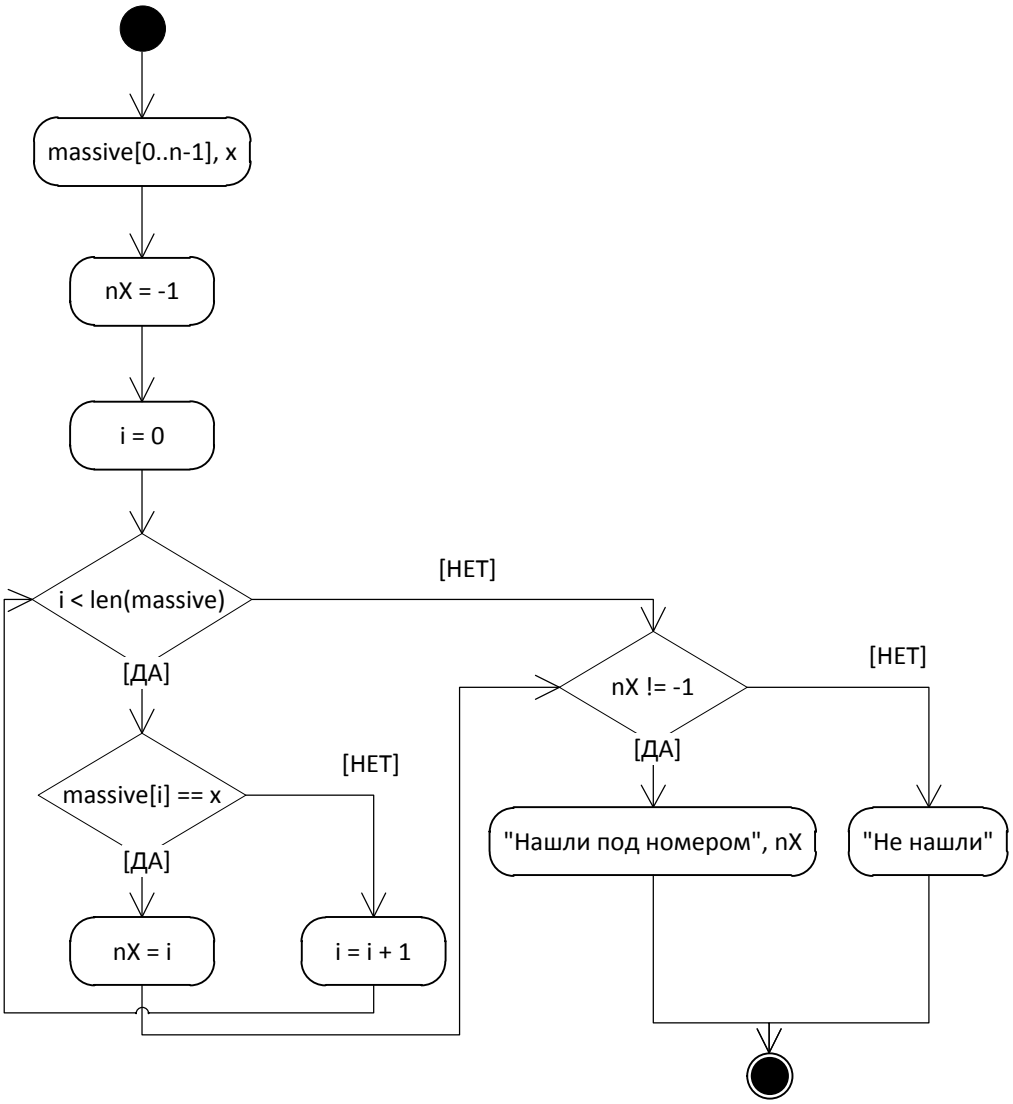

Рисунок 1 – Диаграмма деятельностей для алгоритма последовательного поиска.

Для того чтобы понять, как быстро работает алгоритм, при помощи дополнительных библиотек построим гистограмму, на которой будет указано зависимость времени от трех различных чисел, необходимых найти (возьмем начальное значение, середину и конечное значение). Так как последовательный поиск при конечном значение затрачивает довольно много времени, приведем график отдельно. Гистограмма для последовательного поиска представлена на рисунке 2. Гистограмма – способ графического представления табличных данных. Количественные соотношения некоторого показателя представлены в виде прямоугольников, площади которых пропорциональны. Чаще всего для

удобства восприятия ширину прямоугольников берут одинаковую, при этом их высота определяет соотношения отображаемого параметра.

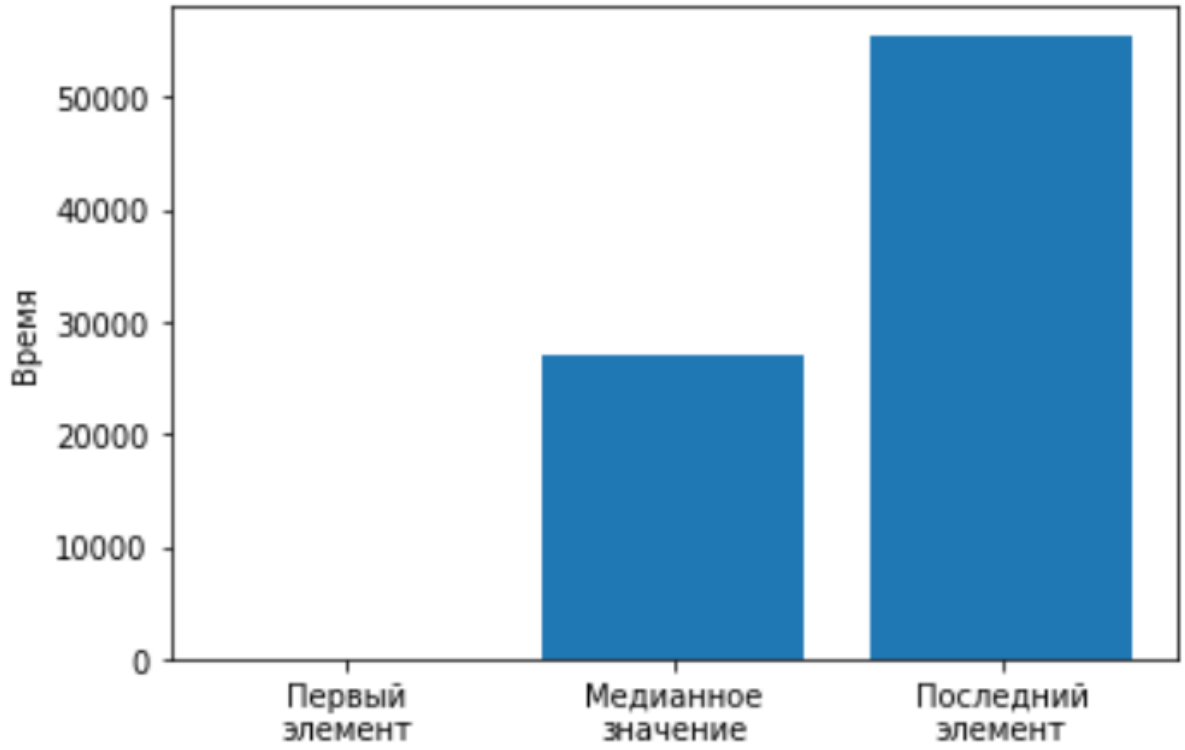

Рисунок 2 – Гистограмма для последовательного поиска, время представлено в микросекундах.

Бинарный (двоичный) поиск.

Бинарный (двоичный, дихотомический) поиск – это поиск заданного элемента на упорядоченном множестве, осуществляемый путем неоднократного деления этого множества на две части таким образом, что искомый элемент попадает в одну из этих частей. Поиск заканчивается при совпадении искомого элемента с элементом, который является границей между частями множества или при отсутствии искомого элемента. Бинарный поиск применяется к отсортированным множествам и заключается в последовательном разбиении множества пополам и поиска элемента только в одной половине на каждой итерации. Таким образом, идея этого метода заключается в следующем. Поиск нужного значения среди элементов упорядоченного массива (по возрастанию или по убыванию) начинается с определения значения центрального элемента этого массива. Значение данного элемента сравнивается с искомым значением и в зависимости от результатов сравнения предпринимаются определенные

действия. Если искомое и центральное значения оказываются равны, то поиск завершается успешно. Если искомое значение меньше центрального или больше, то формируется массив, состоящий из элементов, находящихся слева или справа от центрального соответственно. Затем поиск повторяется в новом массиве.

Алгоритм бинарного поиска

Шаг 1. Определить номер среднего элемента массива.

Шаг 2. Если значение среднего элемента массива равно искомому, то возвращаем значение, равное номеру искомого элемента, и алгоритм завершает работу.

Шаг 3. Если искомое значение больше значения среднего элемента, то возьмем в качестве массива все элементы справа от среднего, иначе возьмем в качестве массива все элементы слева от среднего (в зависимости от характера упорядоченности). Перейдем к Шагу 1.

Алгоритм бинарного поиска:

```
def search_binar(massive, x):
    mid = len(massive) // 2
    low = 0
    high = len(massive) - 1

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

    if low > high:
        print("Не нашли")
    else:
        print("Нашли под номером", mid)
```

На рисунке 3 представим диаграмму деятельностей для алгоритма бинарного поиска.

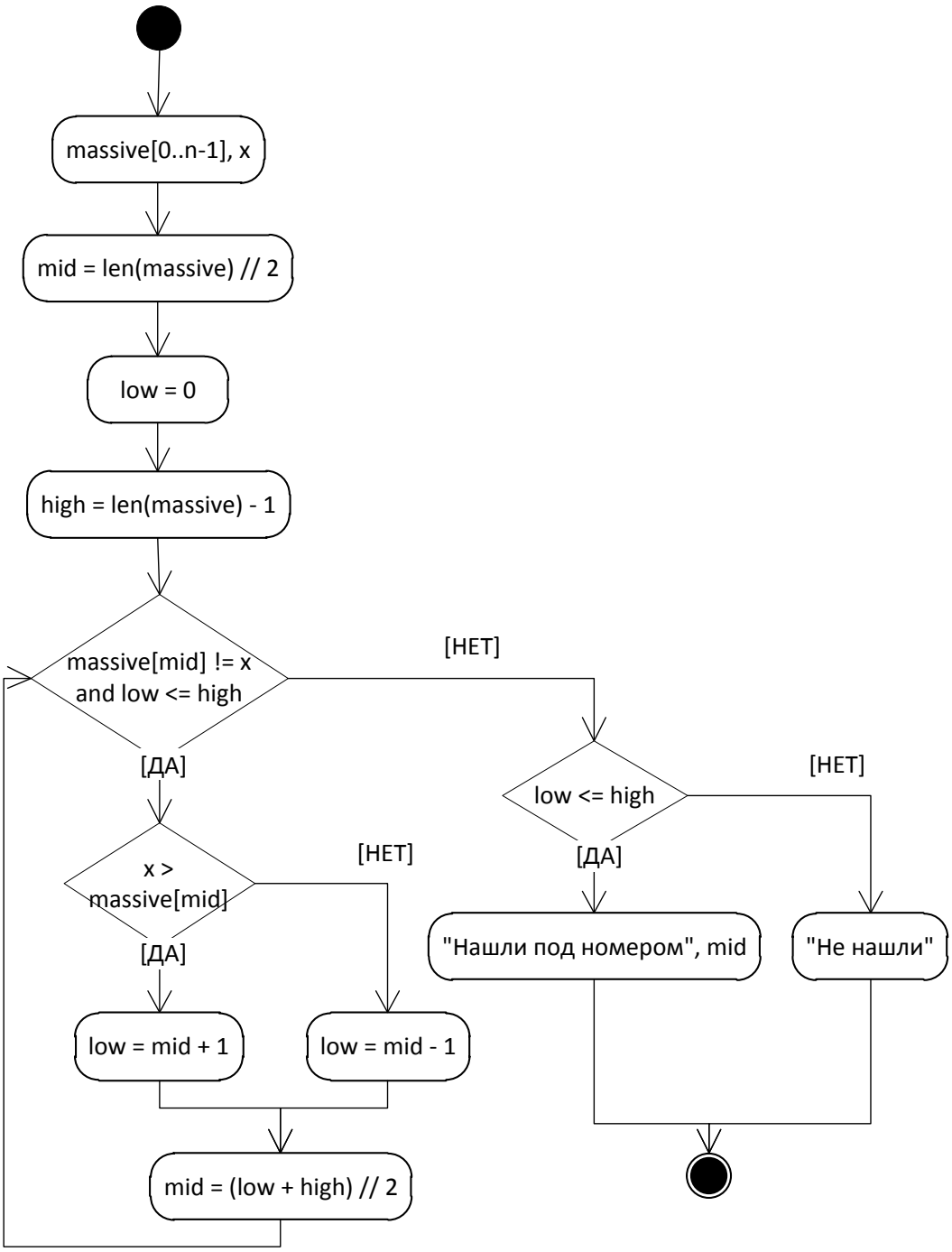

Рисунок 3 – Диаграмма деятельностей для алгоритма бинарного поиска.

Интерполяционный поиск.

В основе интерполяционного поиска лежит операция интерполирование. Интерполирование – нахождение промежуточных значений величины по имеющемуся дискретному набору известных значений. Интерполяционный поиск работает только с упорядоченными массивами; он похож на бинарный, в том смысле, что на каждом шаге вычисляется некоторая область поиска,

которая, по мере выполнения алгоритма, сужается. Но в отличие от двоичного, интерполяционный поиск не делит последовательность на две равные части, а вычисляет приблизительное расположение ключа (искомого элемента), ориентируясь на расстояние между искомым и текущим значением элемента.

Интерполяционный поиск в эффективности, как правило, превосходит двоичный, в среднем требуя log(log(N)) операций. Тем самым время его работы составляет O(log(log(N))). Но если, к примеру, последовательность экспоненциально возрастает, то скорость увеличиться до O(N), где N (как и в предыдущем случае) – общее количество элементов в списке. Наибольшую производительность алгоритм показывает на последовательности, элементы которой распределены равномерно относительно друг друга.

Алгоритм интерполяционного поиска:

```
def search_intpol(massive, x):
    idx0 = 0
    idxn = (len(massive) - 1)

    while idx0 <= idxn and x >= massive[idx0] and x <= massive[i
dxn]:
        mid = idx0 + int(((float(idxn - idx0)/(massive[idxn] -
massive[idx0])) * (x - massive[idx0])))
        if massive[mid] == x:
            print ("Нашли под номером", mid)
            return

        if massive[mid] < x:
            idx0 = mid + 1
    print ("Не нашли")
```

На рисунке 4 представим диаграмму деятельностей для алгоритма интерполяционного поиска.

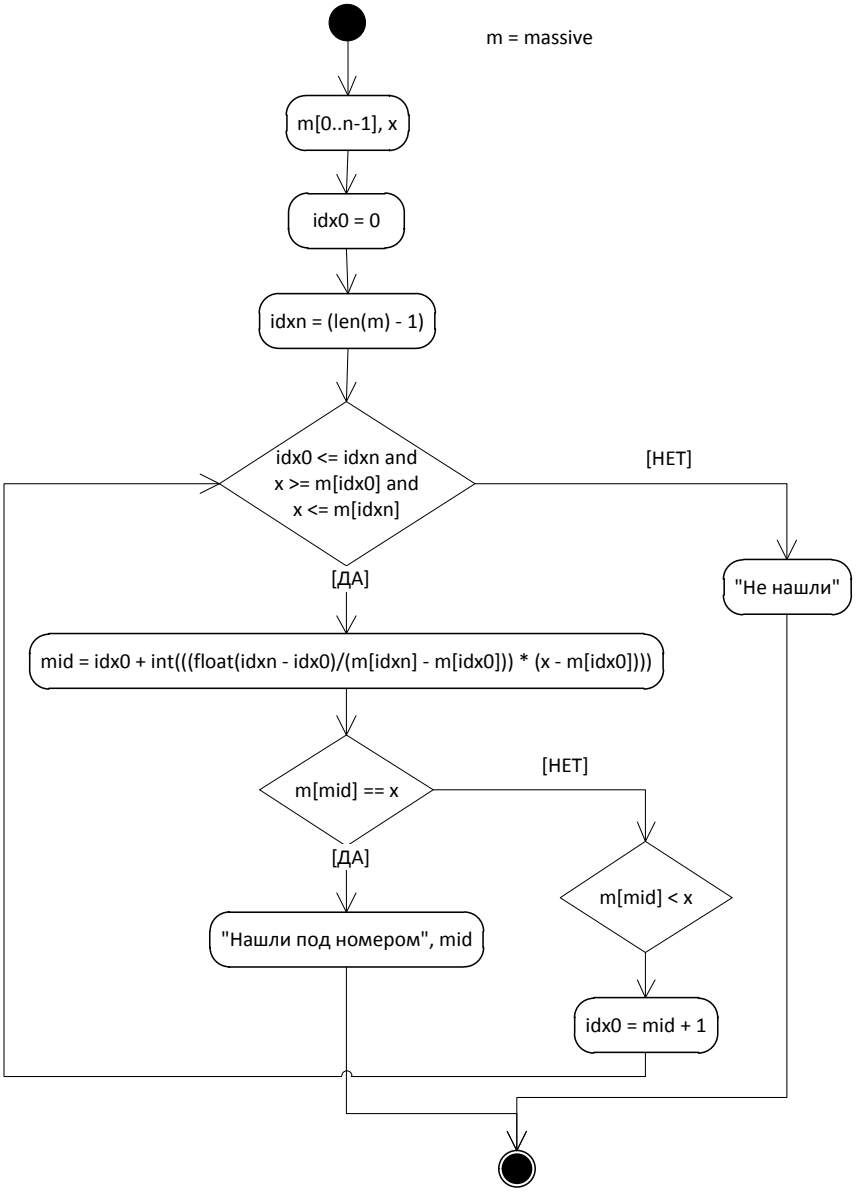

Рисунок 4 – Диаграмма деятельностей для алгоритма интерполяционного поиска.

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 m (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 massive [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 massive[0..n-1] будет входным массивом, а элемент для поиска будет x.

Найдём наименьшее число Фибоначчи, большее или равное n. Пусть это число будет m [m-е число Фибоначчи]. Пусть два предшествующих ему числа Фибоначчи — это m1 [(m-1) -ое число Фибоначчи] и m2 [(m-2) -ое число Фибоначчи].

В то время как массив имеет элементы для проверки:

Сравним x с последним элементом диапазона, охватываемого m2. Если x совпадает, вернуть индекс

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

Если x больше элемента, переместим 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, проверим, не равняется ли m1 = 1.

Если да, сравните x с этим оставшимся элементом.

Если совпадают, верните индекс.

Алгоритм поиска Фибоначчи представлен на следующей странице:

```
def search_fib(massive, x):
    n = len(massive)
    m2, m1 = 0, 1
    m = m2 + m1
    while (m < n):
        m2, m1 = m1, m
        m = m2 + m1
    mid = -1;
    while (m > 1):
        i = min(mid+m2, n-1)
        if (massive[i] < x):
            m, m1 = m1, m2
            m2 = m - m1
            mid = i
        elif (massive[i] > x):
            m = m2
            m1 = m1 - m2
```

```
            m2 = m - m1
        else:
            print("Нашли под номером", i)
            return
    if ((m1 != 0) and (n < mid + 1) and (massive[mid+1] == x)):
        print("Нашли под номером", mid+1)
    else:
        print("Не нашли")
```

На рисунке 5 представим диаграмму деятельностей для алгоритма поиска Фибоначчи.

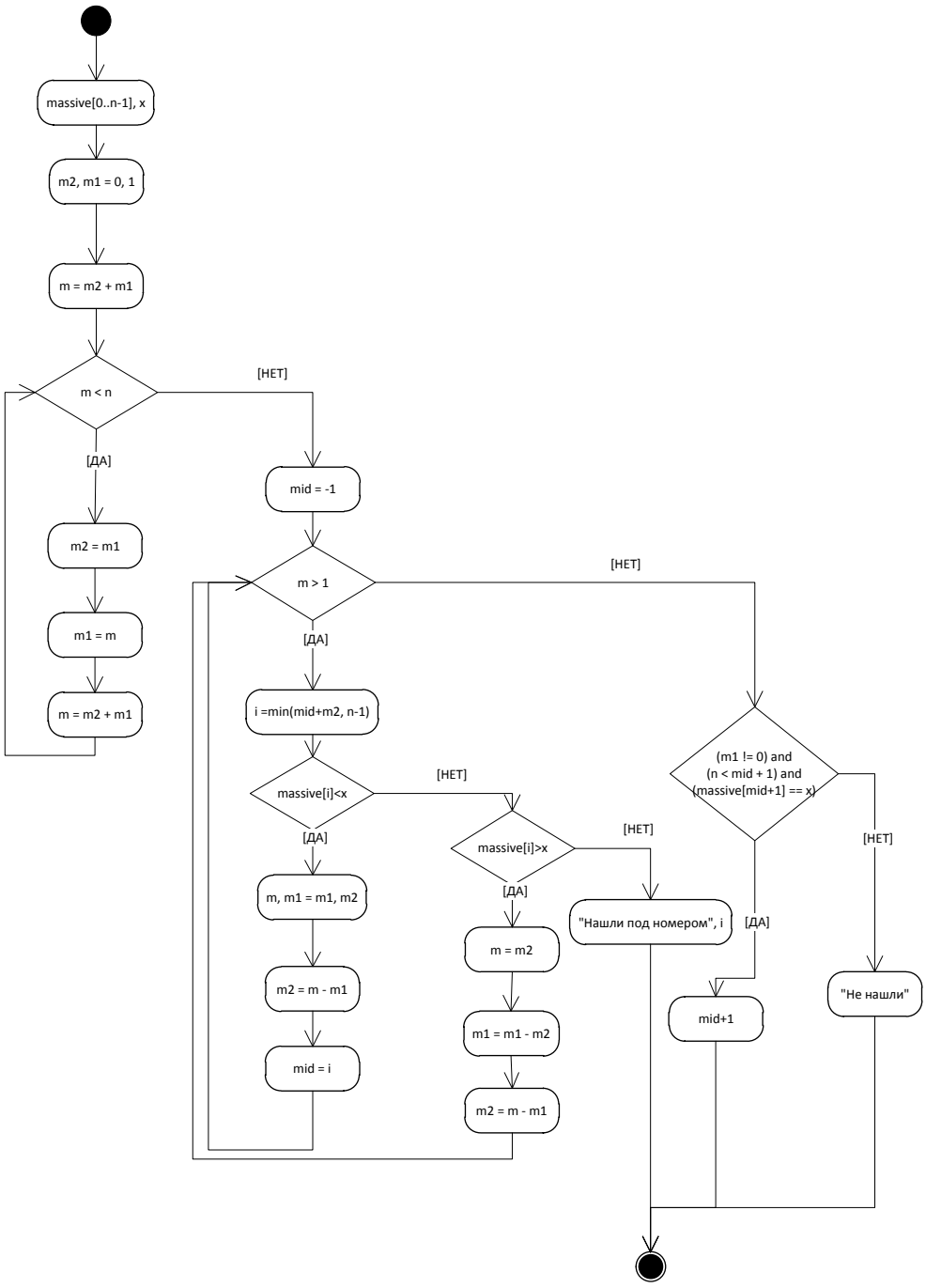

Рисунок 5 – Диаграмма деятельностей для алгоритма поиска Фибоначчи.

Работоспособность программы при задачи любого количества чисел массива с клавиатуры, а так-же вводе искомого числа представлена на рисунке 6.

```
Введите число элементов в массиве (от 1), n = 15252
Введите искомое число, x = 55
Последовательный поиск:
Нашли под номером 55
Затраченное время =  910 мкс
Бинарный поиск:
Нашли под номером 55
Затраченное время =  196 мкс
Интерполяционный поиск:
Нашли под номером 55
Затраченное время =  197 мкс
Фибоначчиев поиск:
Нашли под номером 55
Затраченное время =  232 мкс
```

Рисунок 6 – Работоспособность программы.

Для анализа быстроты работы алгоритмов бинарного, интерполяционного и Фибоначчи поисков, были построены гистограммы зависимостей времени (миллисекунды) от искомого числа в массиве [0, 1.000.000]. Гистограмма, представленная на рисунке 7, отображает зависимость времени от начального элемента массива.

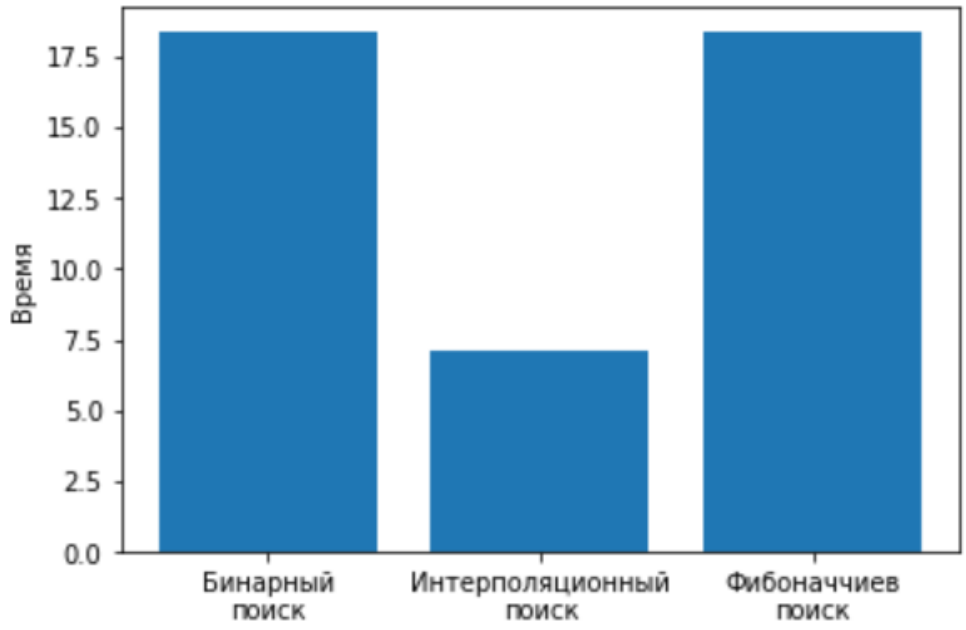

Рисунок 7 – Зависимость времени от поиска начального элемента в массиве.

Гистограмма, представленная на рисунке 8, отображает зависимость времени от среднего элемента массива.

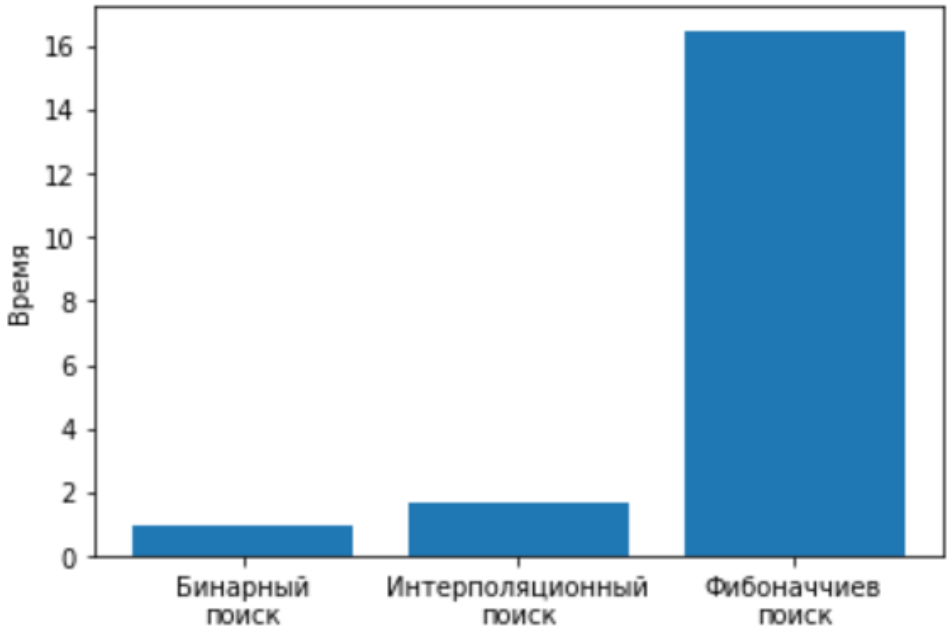

Рисунок 7 – Зависимость времени от поиска среднего элемента в массиве.

Гистограмма, представленная на рисунке 8, отображает зависимость времени от конечного элемента массива.

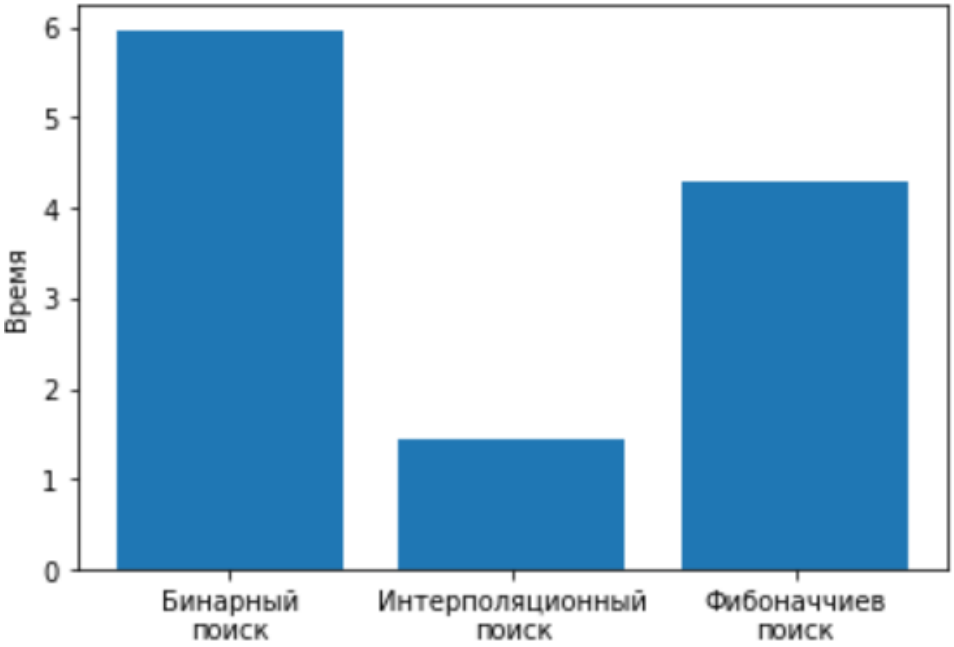

Рисунок 8 – Зависимость времени от поиска конечного элемента в массиве.

Вывод. Мы изучили алгоритмы поиска (последовательный, бинарный, интерполяционный, Фибоначчи) и построили графики зависимости времени от начального, среднего и конечного числа в массиве.