**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра САПР**

**КУРСОВАЯ РАБОТА**

**по дисциплине «Алгоритмы и структуры данных».**

**Тема: Сравнение различных алгоритмов поиска в массиве**.

|  |  |  |
| --- | --- | --- |
| Студентка гр. 3354 |  | Чикарёва М.Д. |
| Преподаватель |  | Пестерев Д. О. |

**Санкт-Петербург**

**202****4**

# Задание на курсовую работу

Студентка Чикарёва М.Д.

Группа 3354.

Тема работы: сравнение различных алгоритмов поиска в массиве.

Исходные данные: язык написания кода - python, среды для заключительной сборки - google collaboratory, первоначальное написание - pycharm.

Содержание пояснительной записки: «Содержание», «Аннотация», «Введение», «Теоретическая часть», «Практическая часть», «Раздел со всем кодом», «Заключение».

Предполагаемый объём пояснительной записки: не менее 20 страниц.

Дата выдачи задания: 19. 11. 2024

Дата сдачи задания: 20. 12. 2024

Дата защиты задания:

|  |  |  |
| --- | --- | --- |
| Студентка гр. 3354 |  | Чикарёва М.Д. |
| Преподаватель |  | Пестерев Д. О. |

# Анотация

В данной курсовой работе будут рассматриваться несколько алгоритмов поиска в массиве для того, чтобы сравнить их друг с другом и узнать, какой из них в каком случае эффективнее использовать, почему именно их лучше выбирать для той или иной задачи.

Конкретные рассматриваемые алгоритмы: линейный поиск, бинарный поиск, интерполяционный поиск, поиск прыжками и поиск Фибоначчи.

В практической части работы все функции будут воспроизведены на одинаковых массивах для каждого случая расположения элементов в массиве (упорядоченный, произвольный) будет посчитано время выполнения, также будет произведено сравнение только тех алгоритмов, которые работают на отсортированных массивах.

**Summary**

In this course work, several array search algorithms will be considered in order to compare them with each other and find out which one is more effective to use in which case, and why it is better to choose them for a particular task.

The specific algorithms considered are linear search, binary search, interpolation search, jump search and Fibonacci search.

In the practical part of the work, all functions will be reproduced on the same arrays for each case of the arrangement of elements in the array (ordered, arbitrary), the execution time will be calculated, and only those algorithms that work on sorted arrays will be compared.

# Содержание

[Задание на курсовую работу 2](#_Toc22049)

[Анотация 3](#_Toc9139)

[Содержание 4](#_Toc5147)

[Введение 5](#_Toc6137)

[Теоретическая часть 6](#_Toc11208)

[Линейный поиск (linear search) 6](#_Toc30228)

[Бинарный поиск (binary search) 8](#_Toc6715)

[Интерполяционный поиск (interpolation search) 10](#_Toc10263)

[Поиск прыжками (jump search) 12](#_Toc17726)

[Поиск Фибоначчи (fibonacci search) 15](#_Toc14473)

[Общая итоговая таблица 18](#_Toc24986)

[Практическая часть 19](#_Toc29692)

[Сравнение отсортированного/неотсортированного массивов. 19](#_Toc4288)

[Сравнение алгоритмов, эффективных только на отсортированных массивах. 21](#_Toc2216)

[Сравнение алгоритмов на упорядоченном отсортированном массиве. 22](#_Toc6279)

[Раздел со всем кодом 23](#_Toc7061)

[Заключение 29](#_Toc28191)

# Введение

Эффективный поиск данных – одна из наиболее важных задач в программировании и информатике. От скорости поиска зависит производительность приложений и удобство работы с ними.

Умение выбрать нужный алгоритм для конкретной задачи является ключевым навыком для разработчиков. Именно правильно подобранный алгоритм отличает быстрое, надежное и стабильное приложение от приложения, которое падает от простого запроса.

В данном исследовании мы проводим сравнительный анализ пяти популярных алгоритмов поиска: линейного, бинарного, интерполяционного, поиска прыжками и поиска Фибоначчи. Все алгоритмы, за исключением линейного, требуют предварительной сортировки данных, что накладывает ограничения на их применение в определенных случаях.от

Целью этой работы является не только изучение теоретической основы данных алгоритмов, но и анализ их производительности на примере неотсортированного, отсортированного и отсортированного упорядоченно массивов, что позволит определить их практическую ценность для решения различных задач. Мы рассмотрим как их асимптотическую сложность, так и их фактическое время выполнения.

# Теоретическая часть

## Линейный поиск (linear search)

1. Асимптотическая временная и пространственная сложность.

|  |  |  |  |
| --- | --- | --- | --- |
| Tbest | Taverage | Tworst | S’(n) |
| O(1) | O(n) | O(n) | O(1) |

1. Алгоритм поиска.

Линейный поиск - метод поиска элементов в массиве, его ещё называют последовательным поиском. Он является простейшим алгоритмом, так как ищет желаемый элемент последовательно, и работает как с отсортированными, так и с не отсортированными массивами.

Алгоритм перебирает каждый элемент массива от начала до конца и сравнивает его с искомым значением. Если элемент найден, возвращается его индекс, иначе - -1.

1. Функция временной сложности.

n - размер массива, в котором нужно выполнить поиск. Элемент можно найти между индексами от 0 до n-1. В **худшем случае** алгоритм попытается сопоставить все элементы массива с элементом поиска, так как в выбранном массиве n элементов, то количество операций будет равно n и сложность будет O(n).

Для оценки **среднего случая** нужно определить вероятность нахождения элемента в каждом из возможных индексов массива. Длина массива - n, тогда вероятность нахождения искомого значения в каждом из индексов равно 1/n. Общее количество операций: 1+2+3+…+n = -> Tav(n) = (1+2+3+…+(n-1)+n)= = ; Tav(n) ≈ .

В **лучшем случае** нужно найти элемент, находящийся в первой позиции массива, при такой ситуации алгоритм линейного поиска не будет искать все n элементов массива, поэтому сложность будет O(1), т.е. постоянное время.

1. Асимптотическая оценка функций временной сложности.

Худший случай: O(n);

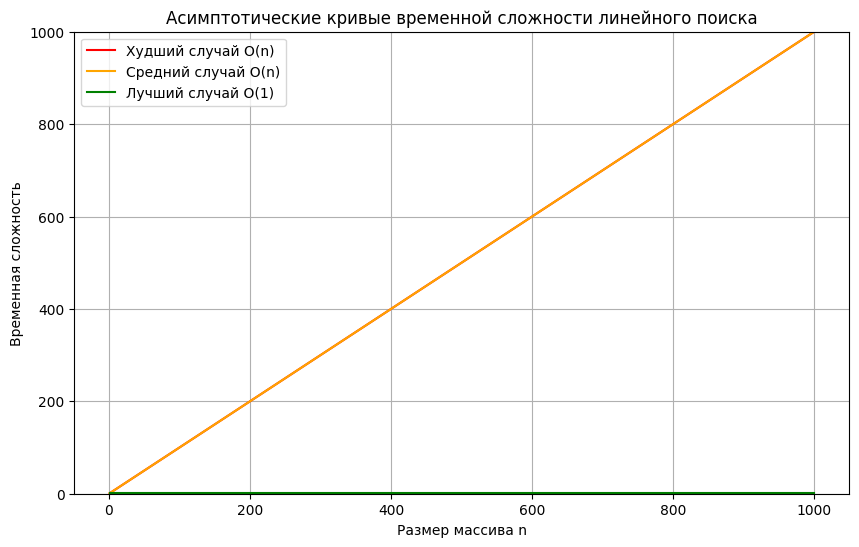
Средний случай: Tav(n) ≈ = O(n);

Лучший случай: O(1).

1. Функция и асимптотическая оценка пространственной сложности.

O(1), так как использует константное количество дополнительной памяти, не нужно хранить или использовать какие-либо временные переменные в функции линейного поиска.

1. График.



## Бинарный поиск (binary search)

1. Асимптотическая временная и пространственная сложность.

|  |  |  |  |
| --- | --- | --- | --- |
| Tbest | Taverage | Tworst | S’(n) |
| O(1) | O(logn) | O(logn) | O(1) |

1. Алгоритм поиска.

Бинарный поиск — это более эффективный алгоритм, который работает только с отсортированными списками. Он делит список пополам и сравнивает средний элемент с искомым значением, повторяя процесс для половины, в которой может находиться искомый элемент.

1. Функция временной сложности.

**Худший случай** происходит, когда искомый элемент не находится в массиве или находится в его начале. Каждое деление массива уменьшает его размер вдвое и продолжается до тех пор, пока размер массива не станет равным 1. Пусть n - кол-во элементов в массиве, после первого сравнения останется n/2, затем n/4 и т.п.

Можно записать как = 1 -> k = log2n.

**Средний случай** происходит, когда искомый элемент присутствует в массиве и его расположение случайно.Так же, как и в худшем случае, на каждой итерации размер блока поиска уменьшается до n/2k, где k - кол-во итераций, необходимых для нахождения элемента. Так как размер такой же, то число проходов тоже, поэтому Tav(n) = O(logn).

**Лучший случай** происходит тогда, когда искомый элемент находится в середине массива, значит, алгоритм завершит работу за одну операцию, если число элементов в массиве больше 0, поэтому Tbest(n) = O(1).

1. Асимптотическая оценка функций временной сложности.

Худший случай: O(logn);

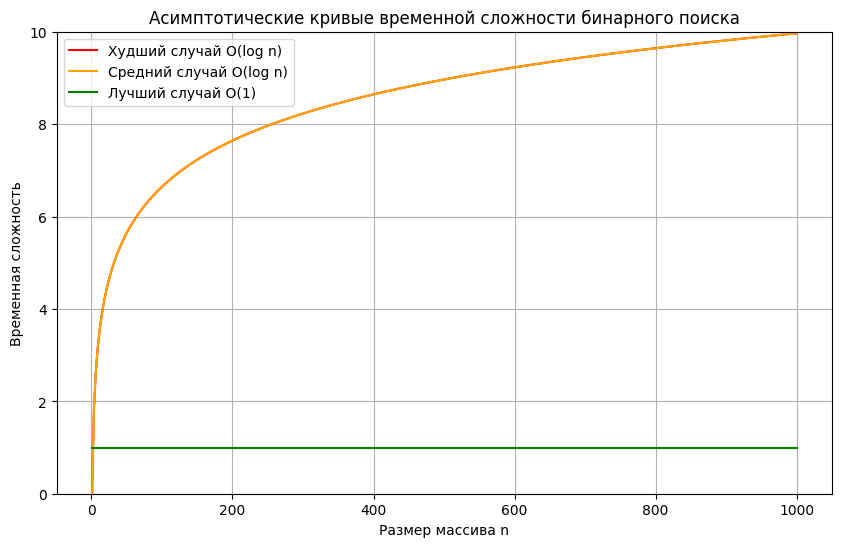
Средний случай: O(logn);

Лучший случай: O(1).

1. Функция и асимптотическая оценка пространственной сложности.

Пространственная сложность двоичного поиска равна O(1), что означает, что он требует постоянного количества дополнительной памяти независимо от размера входного массива. Это связано с тем, что двоичный поиск — это итеративный алгоритм, который не требует дополнительных структур данных или рекурсии, растущей вместе с размером входных данных. Однако мы также можем реализовать двоичный поиск рекурсивно.

1. График.



## Интерполяционный поиск (interpolation search)

1. Асимптотическая временная и пространственная сложность.

|  |  |  |  |
| --- | --- | --- | --- |
| Tbest | Taverage | Tworst | S’(n) |
| O(1) | O(loglogn) | O(n) | O(1) |

1. Алгоритм поиска.

Интерполяционный поиск является ещё одним алгоритмом «разделяй и властвуй», аналогичным бинарному поиску. Но в отличии от бинарного поиска, он не всегда начинает поиск с середины. Интерполяционный поиск вычисляет вероятную позицию искомого элемента по формуле:

index = low + [(target-arr[low])\*(high-low) / (arr[high]-arr[low])]

В этой формуле используются следующие переменные:

- arr — наш входной массив.

- target — искомый элемент.

- index — вероятный индекс искомого элемента. Он вычисляется как более высокое значение, когда значение val ближе по значению к элементу в конце массива (arr[high]), и более низкое, когда значение val ближе по значению к элементу в начале массива (arr[low]).

- low — начальный индекс массива.

- high — последний индекс массива.

Алгоритм осуществляет поиск путем вычисления значения индекса:

- Если значение найдено (когда arr[index] == target), возвращается индекс.

- Если значение target меньше arr[index], то значение индекса пересчитывается по формуле для левого подмассива.

- Если значение target больше arr[index], то значение индекса пересчитывается по формуле для правого подмассива.

1. Функция временной сложности.

В **худшем случае** алгоритм может превратиться в линейный поиск, такое происходит тогда, когда искомый элемент находится на краю диапазона или если данные распределены неравномерно (например, когда все элементы имеют одинаковое значение). В таком случае алгоритм будет проверять каждый элемент друг за другом, до n. -> Tw(n) = O(n).

В **среднем случае** мы предполагаем, что искомый элемент может находиться в любом месте массива и если он равномерно распределён, то алгоритм эффективнее и использует информацию о распределении значений.

n - кол-во элементов в массиве, на каждой итерации поиска алгоритм вычисляет позицию по формуле, приведённой раньше.

Пусть T(n) - среднее кол-во проб, необходимых для поиска ключа в массиве размером n.

Пусть C - ожидаемое кол-во проб, необходимых для сокращения пространства поиска размером x до.

Согласно лемме C будет ограничена константой Cn. -> T(n) ≤ C’ + T().

Предположим, что n =z^2^k для некоторых k.

T(n) = T(z^2^k) ≤ C’ + T(z^2^k-1) ≤ C’ + C’ + T(z^2^k-2) ≤ C’ + C’…+ T(z) = kC’ + T(z).

И потому, что n =z^2^k и z ≥ 2 если n > 1, log2(z) ≥ 1.

Следовательно, k = log2(log2(n)) -> T(n) ≤ C’ x log(log(n)) + T(z).

**Лучший случай** происходит, когда искомый элемент находится на первой позиции и алгоритм находит его с первой попытки, алгоритм выполняет только одно сравнение, независимо от общего числа элементов в массиве -> Tb (n) = O(1).

1. Асимптотическая оценка функций временной сложности.

Худший случай: O(n);

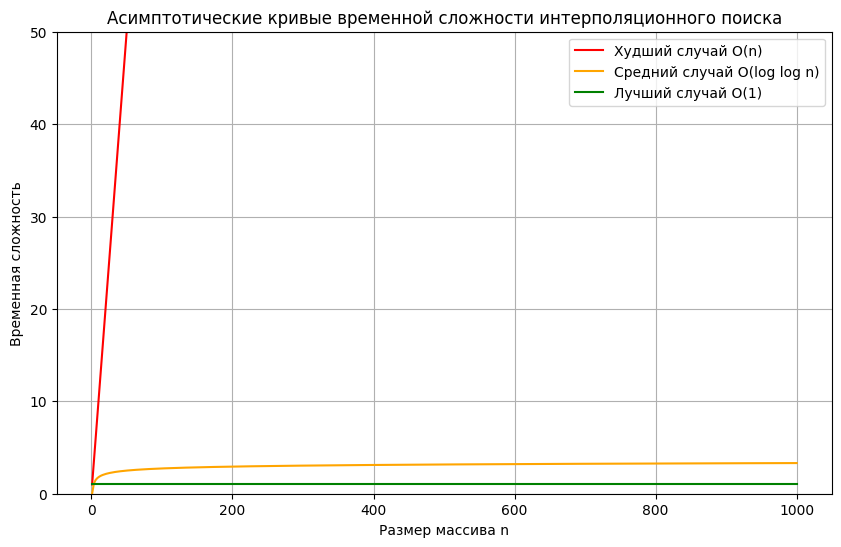
Средний случай: O(log(log(n));

Лучший случай: O(1).

1. Функция и асимптотическая оценка пространственной сложности.

Интерполяционный поиск использует всего несколько переменных для отслеживания нижней и верхней границ поиска: low, high и index. Эти переменные имеют фиксированный размер и занимают постоянное пространство - O(1).

1. График.



## Поиск прыжками (jump search)

1. Асимптотическая временная и пространственная сложность.

|  |  |  |  |
| --- | --- | --- | --- |
| Tbest | Taverage | Tworst | S’(n) |
| O(1) | O() | O() | O(1) |

1. Алгоритм поиска.

Поиск прыжками - алгоритм поиска, который работает на отсортированных массивах. Он является альтернативой бинарному поиску и линейному поиску и сочетает в себе их достоинства.

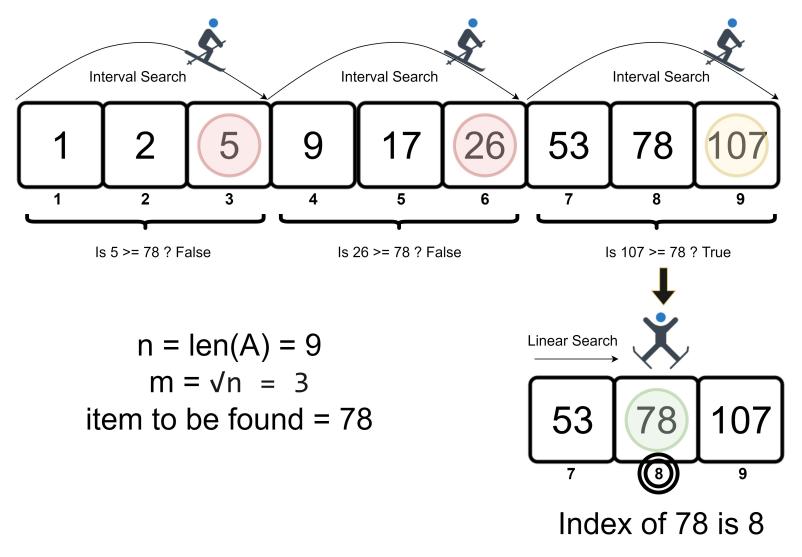
Сначала происходит разделение массива на блоки фиксированного размера. Дальше с начала массива начинаются «прыжки» по блокам, нужное (искомое) значение ищется при сравнении кандидата на поиск в каждом блоке. Поскольку массив отсортирован, кандидатом на поиск является наибольшее значение в блоке. При сравнении искомого с кандидатом на поиск алгоритм может выполнить одно из трёх действий:

- если кандидат меньше, чем нужное число, мы проверяем следующий блок;

- если кандидат больше искомого, то в этом блоке может находиться нужное значени, поэтому алгоритм выполняет линейный поиск в этом блоке;

- если кандидат совпадает с искомым, то возвращается кандидат.

Размер блока выбирается как квадратичный корень из длины массива. Таким образом, массивы длиной n имею размер блока , так как это в среднем обеспечивает наилучшую производительность для большинства массивов.



1. Функция временной сложности.

В **худшем случае**, если искомый элемент находится в самом конце массива или вообще отсутствует, алгоритму придётся проверить все блоки и выполнить линейный поиск по последнему блоку. Количество блоков будет равно n/ = . -> линейный поиск внутри блока будет занимать время O().

В **среднем случае**, как правило, алгоритм будет находить искомый элемент в промежутке между первым и последним блоком. Поскольку он не будет выполнять полный линейный поиск, количество необходимого времени будет меньше, чем в худшем случае, но всё равно основано на размере блока.

Tw(n) = O()(прыжки) + O(/2)(линейный поиск) = O().

**Лучший случай** происходит, когда искомый элемент находится в первом проверенном блоке или в самом начале, в этом случае алгоритм выполнит только один прерывающий поиск и сразу же найдет элемент. -> Tb(n) = O(1).

1. Асимптотическая оценка функций временной сложности.

Худший случай: O();

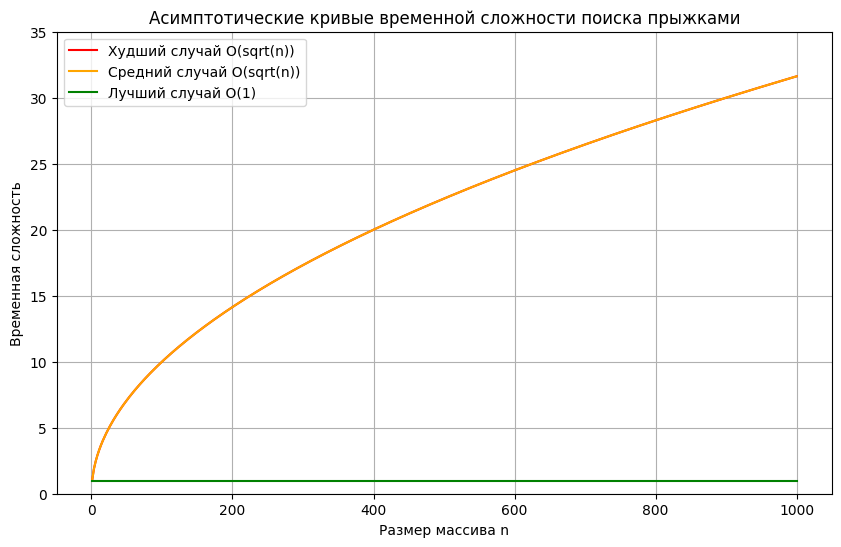
Средний случай: O();

Лучший случай: O(1).

1. Функция и асимптотическая оценка пространственной сложности.

Поскольку поиск прыжками использует только фиксированный набор дополнительных переменных и не требует дополнительной памяти, зависящей от размера входного массива, то пространственная сложность будет O(1).

1. График.



## Поиск Фибоначчи (fibonacci search)

1. Асимптотическая временная и пространственная сложность.

|  |  |  |  |
| --- | --- | --- | --- |
| Tbest | Taverage | Tworst | S’(n) |
| O(1) | O() | O() | O(1) |

1. Алгоритм поиска.

Поиск Фибоначчи — это еще один алгоритм «разделяй и властвуй», который имеет сходство как с бинарным поиском, так и с jump search. Он получил свое название потому, что использует [числа Фибоначчи](https://ru.wikipedia.org/wiki/%D0%A7%D0%B8%D1%81%D0%BB%D0%B0_%D0%A4%D0%B8%D0%B1%D0%BE%D0%BD%D0%B0%D1%87%D1%87%D0%B8" \t "https://pythonist.ru/algoritmy-poiska-na-python/_blank) для вычисления размера блока или диапазона поиска на каждом шаге.

Числа Фибоначчи  — это последовательность чисел 0, 1, 1, 2, 3, 5, 8, 13, 21 …, где каждый элемент является суммой двух предыдущих чисел.

Алгоритм использует числа Фибоначчи для деления массива на части, а не просто вдоль середины, что является характерным для бинарного поиска. Зная, что Fn = Fn-1 + Fn-2, можно использовать предшествующие числа Фибоначчи, чтобы определить размеры разделов. Например, если длина массива составляет n, то выбор индекса элемента будет равен индексу Fk, где Fk - наибольшее число Фибоначчи, которое меньше или равно n. После нахождения индекса текущего элемента, поиск продолжается либо в левой (если искомый элемент меньше текущего), либо в правой (если больше) части массива, аналогично процедуре бинарного поиска.

1. Функция временной сложности.

В **худшем случае** поиск Фибоначчи будет аналогичен бинарному поиску, поскольку алгоритм будет сокращать размер массива с каждым шагом. Если на каждом шаге алгоритм сокращает размер массива на Fk-2 при текущем Fk, можно утверждать, что количество шагов, необходимых для завершения поиска, составляет Tw(n) = O(logn). В **среднем случае** поведение алгоритма будет аналогично худшему, поскольку он также использует числа Фибоначчи для деления массива на подмассивы. **Лучший случай** происходит тогда, когда искомый элемент находится в массиве на первой проверке. Это может произойти, если элемент, который нужно найти, соответствует элементу по индексу Фибоначчи на первой итерации -> Tb(n) = O(1).

1. Асимптотическая оценка функций временной сложности.

Худший случай: O(logn);

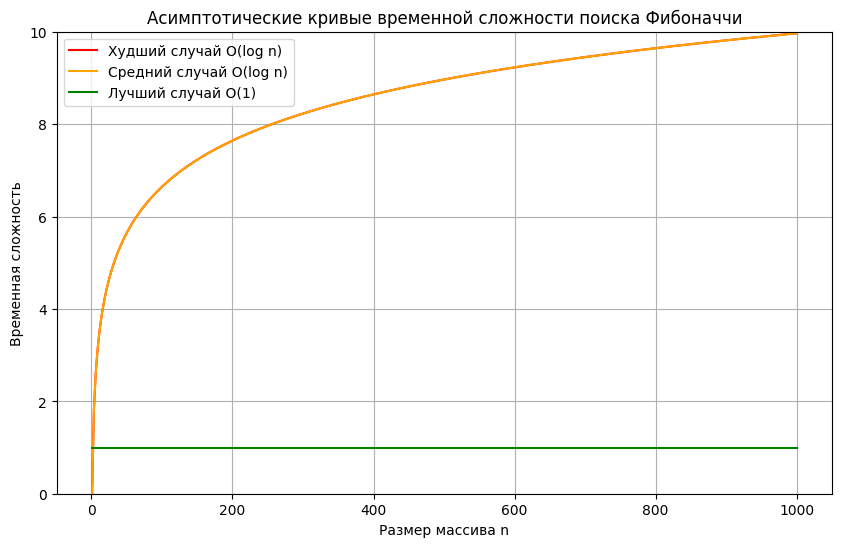
Средний случай: O(logn);

Лучший случай: O(1).

1. Функция и асимптотическая оценка пространственной сложности.

Фибоначчи использует только фиксированный набор дополнительных переменных и не требует дополнительной памяти, зависящей от размера входного массива, поэтому пространственная сложность O(1).

1. График.



## Общая итоговая таблица

| Алгоритм | Временная сложность | Пространственная сложность | Описание |
| --- | --- | --- | --- |
| Линейный поиск | O(n) | O(1) | Перебор всех элементов |
| Бинарный поиск | O(log n) | O(1) | Делит массив пополам |
| Интерполяционный поиск | O(log log n) | O(1) | Использует распределение значений |
| Поиск прыжками | O(√n) | O(1) | Прыжки по блокам |
| Поиск Фибоначчи | O(log n) | O(1) | Использует числа Фибоначчи |

# Практическая часть

В данном разделе будет произведено сравнение алгоритмов поиска, чтобы выявить, какие из них эффективнее и когда.

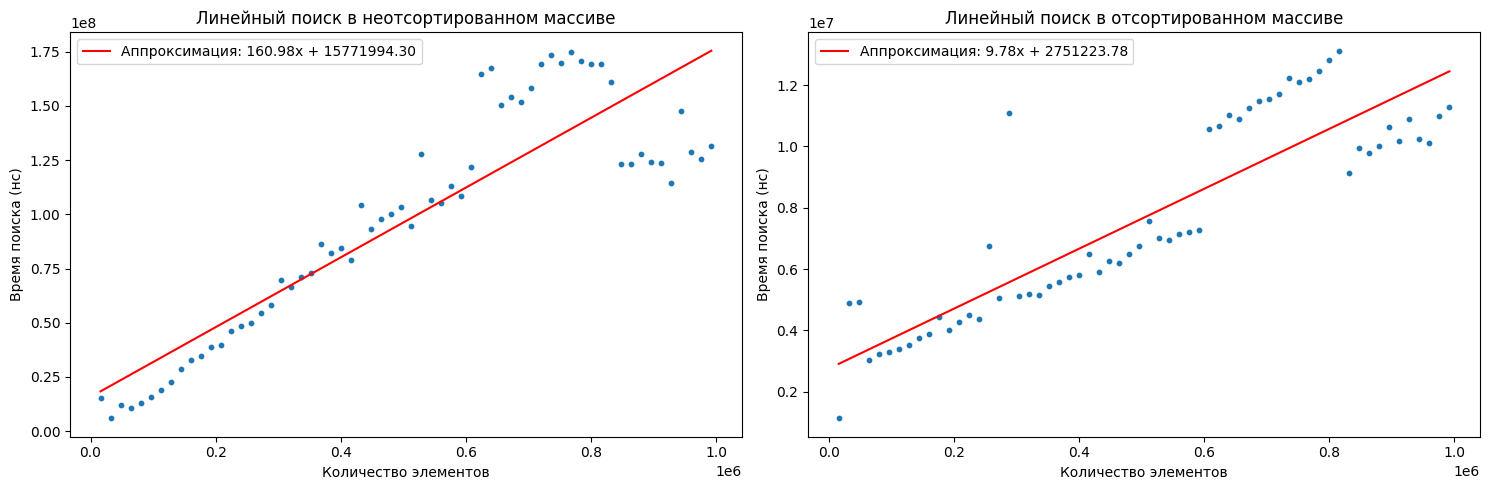
Категории сравнения будут следующими: каждый алгоритм на отсортированном и неотсортированном массивах (чтобы проверить, совпадает ли с теоретическими утверждениями), все алгоритмы, работающие только на отсортированных массивах на одинаковом массиве большого размера (отсортированном) и на массиве отсортировванном равномерно.

Брать будем 1000000 чисел, перемешаем их и заполним массивы, создав при этом второй, который будет отсортирован; на каждый 16000-ый элемент будем запускать таймер, который замерит время поиска элемента (который будет выбран из чисел массива). Для каждого полученного набора данных составим аппроксимированные графики зависимости времени от размера. Оси абсцисс соответствуют кол-ву элементов в массиве, оси ординат — времени выполнения в наносекундах.

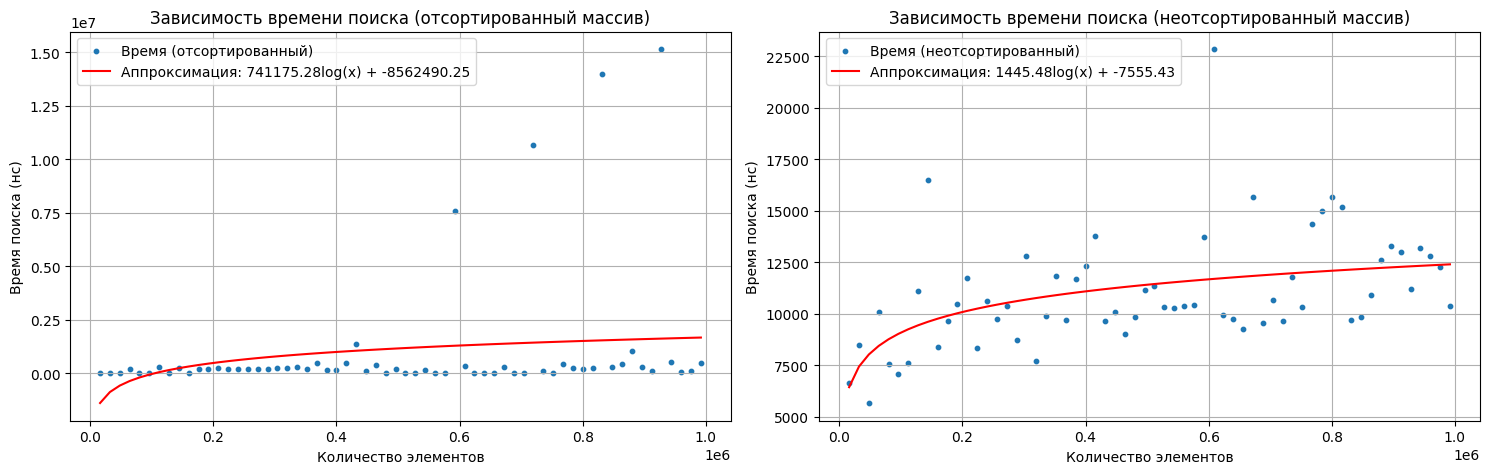
Уравнение регрессионной кривой — .

## Сравнение отсортированного/неотсортированного массивов.

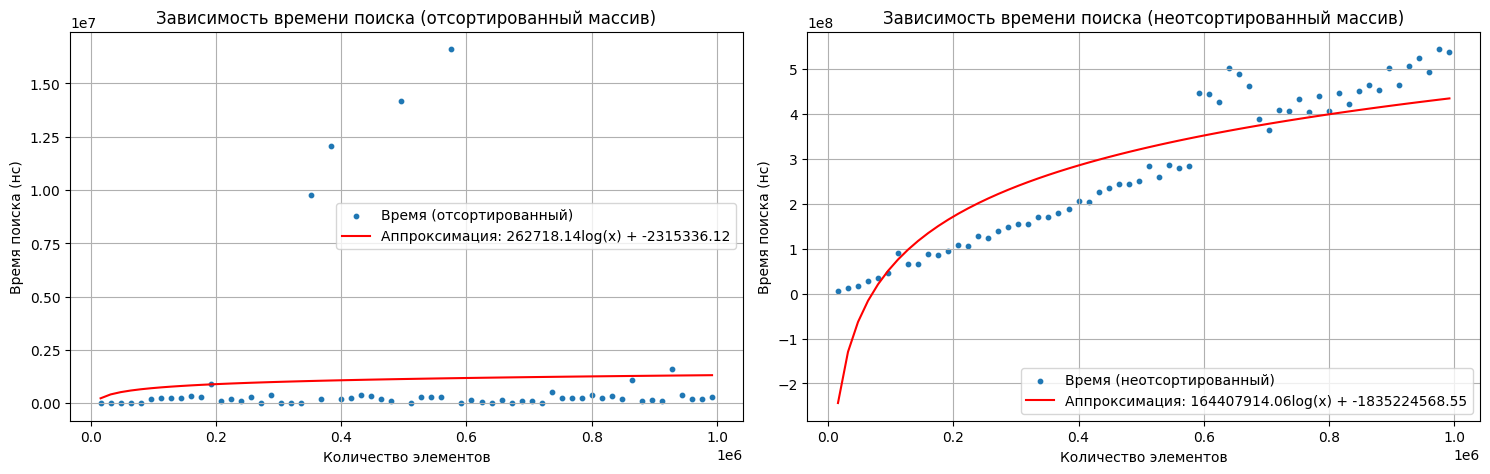
- Линейный поиск.



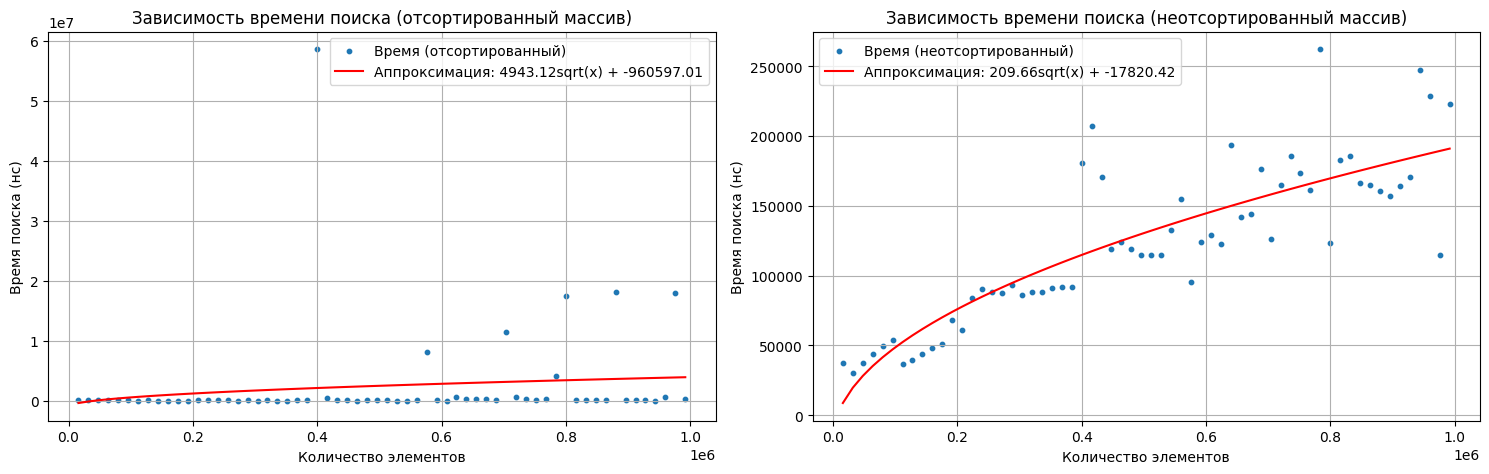
- Бинарный поиск.



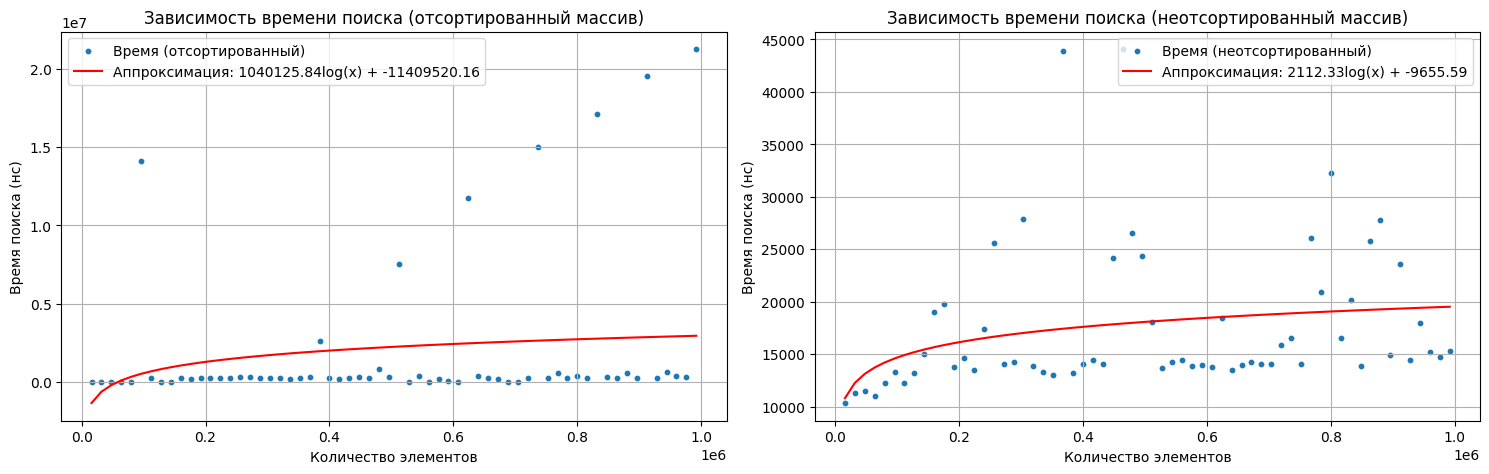
- Интерполяционный поиск.



- Поиск прыжками.

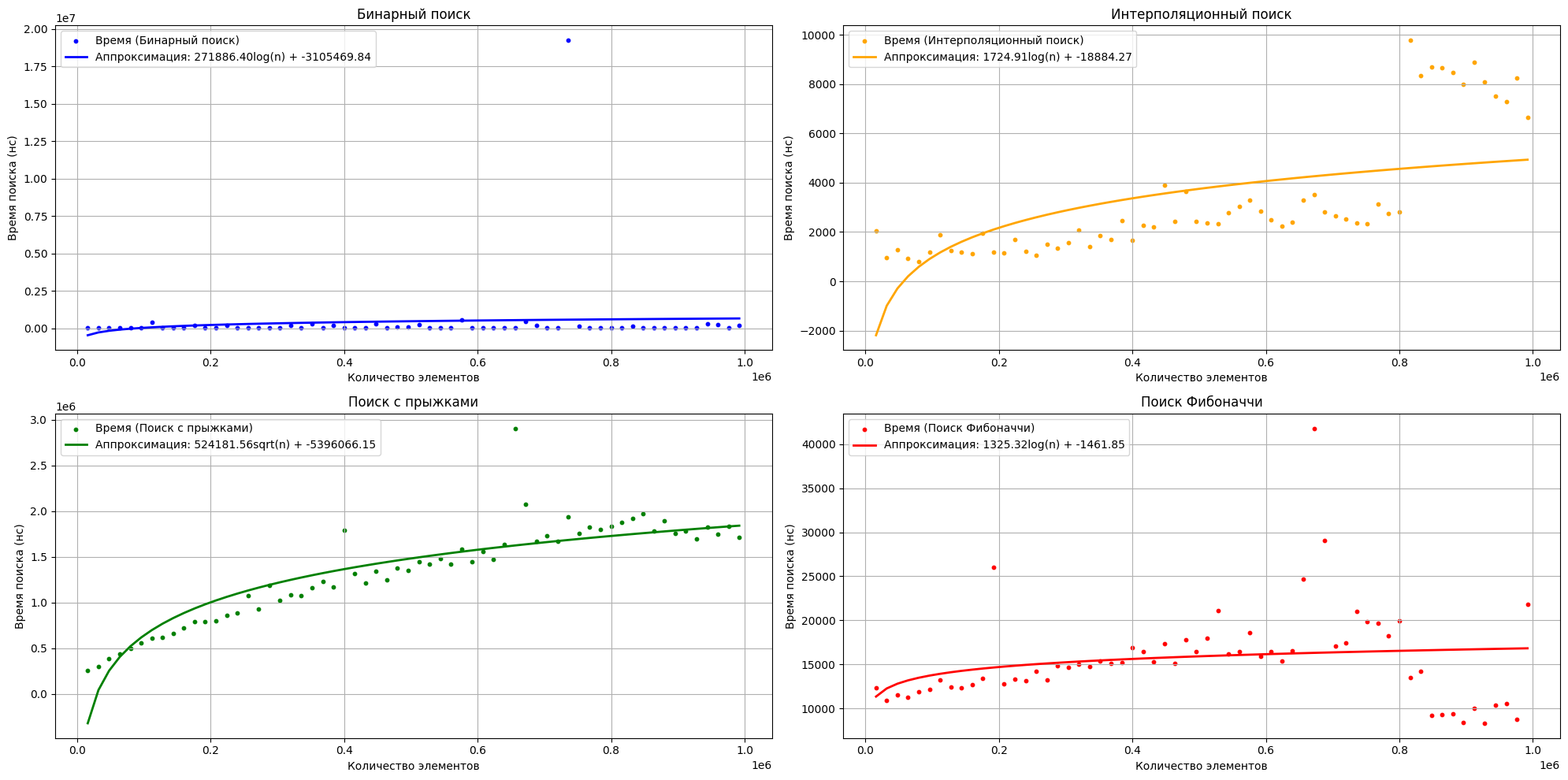


- Поиск Фибоначчи.



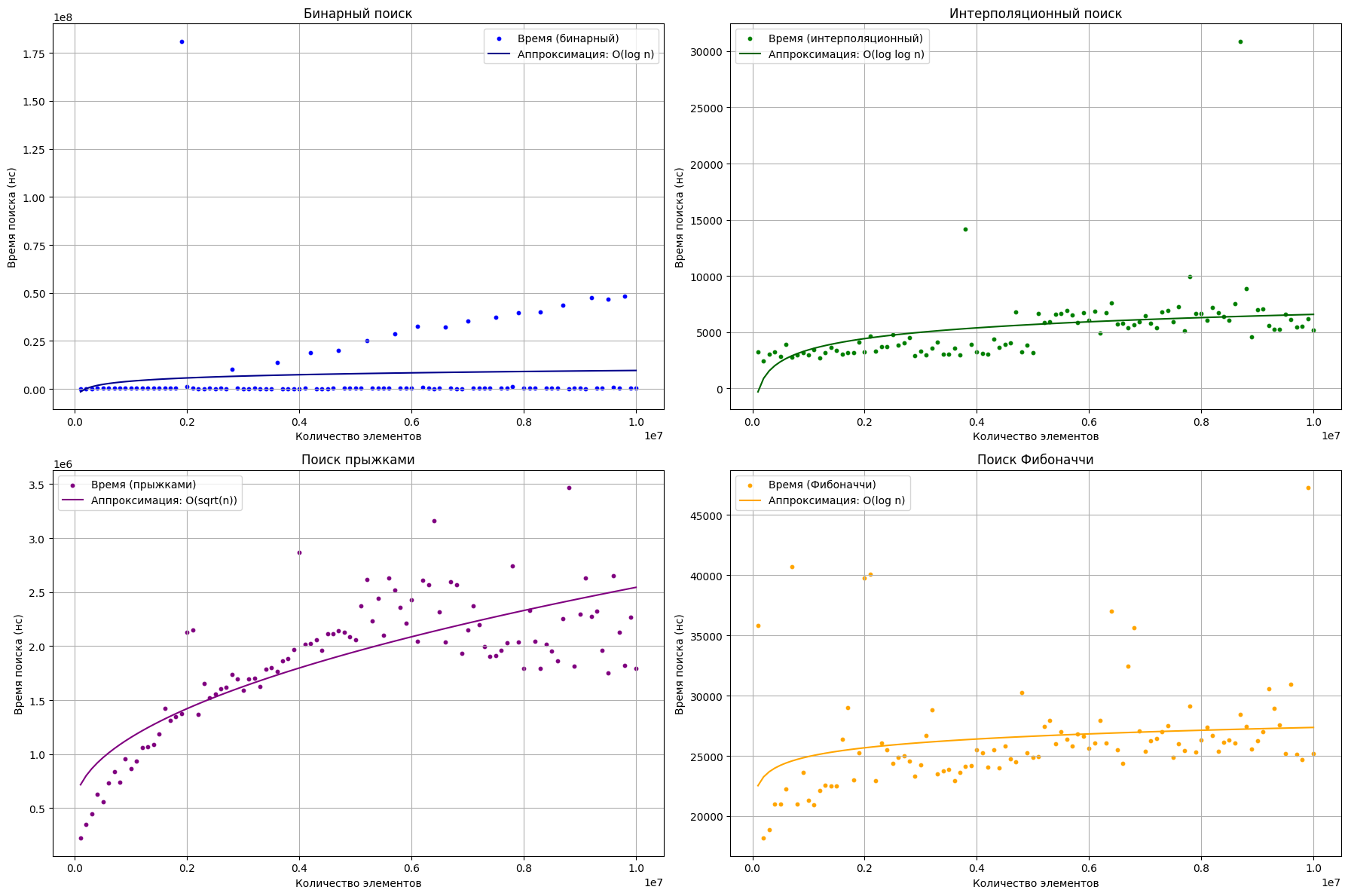
По графикам видно, что действительно все алгоритмы, кроме линейного, на отсортированных работают лучше.

## Сравнение алгоритмов, эффективных только на отсортированных массивах.



Видно, что в одинаковых условиях бинарный поиск всё же быстрее работает, таким образом, для отсортированных массивов бинарный поиск чаще является наиболее эффективным методом.

## Сравнение алгоритмов на упорядоченном отсортированном массиве.



Если данные будут равномерно распределены, то выгоднее будет интерполяционный поиск, но это требует особых условий (он стабильнее работает быстро в таком случае).

Поиск Фибоначчи может быть полезен в специфических сценариях, но в большинстве случаев бинарный будет выигрышнее. Поиск с прыжками испольузется редко из-за своей меньшей эффективности.

# Раздел со всем кодом

import numpy as np

import matplotlib.pyplot as plt

import random

import time

def linear\_search(arr, target):

    for i, num in enumerate(arr):

        if num == target:

            return i

    return -1

def binary\_search(arr, target):

    first = 0

    last = len(arr)-1

    index = -1

    while (first <= last) and (index == -1):

        mid = (first+last)//2

        if arr[mid] == target:

            index = mid

        else:

            if target<arr[mid]:

                last = mid -1

            else:

                first = mid +1

    return index

def interpolation\_search(arr, target):

    left, right = 0, len(arr) - 1

    while left <= right and arr[left] <= target <= arr[right]:

        if left == right:

            if arr[left] == target:

                return left

            return -1

        estimated\_pos = left + ((target - arr[left]) \* (right - left)) // (arr[right] - arr[left])

        if arr[estimated\_pos] == target:

            return estimated\_pos

        elif arr[estimated\_pos] < target:

            left = estimated\_pos + 1

        else:

            right = estimated\_pos - 1

    return -1

def jump\_search(arr, target):

    length = len(arr)

    jump = int(math.sqrt(length))

    prev = 0

    while arr[min(jump, length)-1] < target:

        prev = jump

        jump += int(math.sqrt(length))

        if prev >= length:

            return -1

    for i in range(prev, min(jump, length)):

        if arr[i] == target:

            return i

    return -1

def fibonacci\_search(arr, x):

    n = len(arr)

    fib\_m2 = 0

    fib\_m1 = 1

    fib\_m = fib\_m1 + fib\_m2

    while fib\_m < n:

        fib\_m2 = fib\_m1

        fib\_m1 = fib\_m

        fib\_m = fib\_m1 + fib\_m2

    offset = -1

    while fib\_m > 1:

        i = min(offset + fib\_m2, n - 1)

        if arr[i] < x:

            fib\_m = fib\_m1

            fib\_m1 = fib\_m2

            fib\_m2 = fib\_m - fib\_m1

            offset = i

        elif arr[i] > x:

            fib\_m = fib\_m2

            fib\_m1 -= fib\_m1

            fib\_m2 = fib\_m - fib\_m2

        else:

            return i

    if fib\_m1 and offset + 1 < n and arr[offset + 1] == x:

        return offset + 1

    return -1

numbers = list(range(1000000))

random.shuffle(numbers)

data = []

search\_value = random.choice(numbers)

sorted\_numbers = sorted(numbers)

for i in range(16000, 1000001, 16000):

    current\_array = numbers[:i]

    sorted\_array = sorted\_numbers[:i]

    start\_time = time.perf\_counter\_ns()

    binary\_search(sorted\_array, search\_value)

    sorted\_time = time.perf\_counter\_ns() - start\_time

    start\_time = time.perf\_counter\_ns()

    binary\_search(current\_array, search\_value)

    unsorted\_time = time.perf\_counter\_ns() - start\_time

    data.append((i, sorted\_time, unsorted\_time))

sizes, sorted\_times, unsorted\_times = zip(\*data)

coeffs\_sorted = np.polyfit(np.log(sizes), sorted\_times, 1)

coeffs\_unsorted = np.polyfit(np.log(sizes), unsorted\_times, 1)

plt.figure(figsize=(15, 5))

plt.subplot(1, 2, 1)

plt.scatter(sizes, sorted\_times, s=10, label='Время (отсортированный)')

plt.plot(sizes, coeffs\_sorted[0] \* np.log(sizes) + coeffs\_sorted[1], color='red',

         label=f"Аппроксимация: {coeffs\_sorted[0]:.2f}log(x) + {coeffs\_sorted[1]:.2f}")

plt.xlabel("Количество элементов")

plt.ylabel("Время поиска (нс)")

plt.title("Зависимость времени поиска (отсортированный массив)")

plt.legend()

plt.grid(True)

plt.subplot(1, 2, 2)

plt.scatter(sizes, unsorted\_times, s=10, label='Время (неотсортированный)')

plt.plot(sizes, coeffs\_unsorted[0] \* np.log(sizes) + coeffs\_unsorted[1], color='red',

         label=f"Аппроксимация: {coeffs\_unsorted[0]:.2f}log(x) + {coeffs\_unsorted[1]:.2f}")

plt.xlabel("Количество элементов")

plt.ylabel("Время поиска (нс)")

plt.title("Зависимость времени поиска (неотсортированный массив)")

plt.legend()

plt.grid(True)

plt.tight\_layout()

plt.show()

# Заключение

Было рассмотрено пять алгоритмов поиска элемента в массиве и вот общие выводы, которые можно сделать:

1. Если нужно найти элемент в несортированном массиве или найти первое вхождение искомой переменной, то лучше всего использовать линейный поиск.
2. Если же нужно выполнить поиск в отсортированном массиве, то самый простой и быстрый для этого - бинарный поиск.
3. Если отсортированный массив равномерно распределён, то самым быстрым и эффективным будет интерполяционный поиск.

Код и отчёт загружены в GitHub: