Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Отчет по лабораторной работе №3
по курсу «Анализ алгоритмов»

Тема Алгоритмы поиска в массиве

Студент Талышева О.Н.

Группа ИУ7-55Б

Преподаватели Волкова Л.Л., Строганов Ю.В.

Москва — 2024 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цель работы: исследовать различные алгоритмы поска элемента в массиве (линейный и бинарный алгоритмы).

Для достижения этой цели были поставлены следующие задачи:

1. рассмотреть линейный алгоритм поиска элемента в массиве;
2. рассмотреть бинарный алгоритм поиска элемента в массиве;
3. реализовать оба алгоритма поиска элемента в массиве на выбранном языке программирования;
4. сравнить эффективность этих алгоритмов по числу сравнений до нахождения искомого элемента на практике.

1 Аналитическая часть

1.1 Алгоритм линейного поиска

Линейный поиск — это метод поиска элемента в коллекции элементов. При линейном поиске каждый элемент коллекции просматривается последовательно один за другим, чтобы найти нужный элемент. Линейный поиск также известен как последовательный поиск.

Алгоритм линейного поиска можно разбить на следующие этапы:

1. Начать: Начните с первого элемента коллекции элементов.
2. Сравнить: Сравните текущий элемент с желаемым элементом.
3. Найдено: Если текущий элемент равен искомому элементу, верните значение true или индексируйте текущий элемент.
4. Переместить: В противном случае перейдите к следующему элементу в коллекции.
5. Повтор: Повторяйте шаги 2-4, пока мы не дойдем до конца подборки.
6. Не найдено: если в конце коллекции не найден нужный элемент, верните, что нужного элемента нет в массиве.

В алгоритме линейного поиска:

1. Каждый элемент рассматривается как потенциальное соответствие ключу и проверяется на то же самое.
2. Если найден какой-либо элемент, равный ключу, поиск завершен успешно и возвращается индекс этого элемента.
3. Если не найдено элемента, равного ключу, поиск выдаёт “Совпадение не найдено”.

При линейном поиске мы перебираем все элементы массива и проверяем, равен ли текущий элемент целевому элементу. Если мы обнаружим, что какой-либо элемент равен целевому элементу, то вернем индекс текущего элемента. В противном случае, если ни один элемент не равен целевому элементу, то вернем -1, поскольку элемент не найден.

Временная сложность:

1. Лучший вариант: В лучшем случае ключ может присутствовать в первом индексе. Таким образом, сложность наилучшего варианта равна O(1)
2. Наихудший случай: В наихудшем случае ключ может присутствовать в последнем индексе, т. е. напротив конца, с которого начался поиск в списке. В то же время такую же сложность даёт и отсутствие искомого элемента в массиве. Итак, сложность наихудшего случая равна O(N), где N - размер списка.

3. Средний случай: O (N / 2)

Вспомогательное пространство: O(1) поскольку, кроме переменной для перебора по списку, никакая другая переменная не используется.

Приложения алгоритма линейного поиска:

1. Несортированные списки: Когда у нас есть несортированный массив или список, линейный поиск чаще всего используется для поиска любого элемента в коллекции.

2. Небольшие наборы данных: Линейный поиск предпочтительнее бинарного, когда у нас есть небольшие наборы данных с

3. Поиск по связанным спискам: В реализациях связанных списков линейный поиск обычно используется для поиска элементов в списке. Каждый узел проверяется последовательно, пока не будет найден нужный элемент.

4. Простая реализация: Линейный поиск намного проще понять и реализовать по сравнению с бинарным или троичным поиском.

Преимущества алгоритма:

1. Линейный поиск может использоваться независимо от того, отсортирован массив или нет. Его можно использовать для массивов любого типа данных.

2. Не требует дополнительной памяти. Это хорошо подходящий алгоритм для небольших наборов данных.

Недостатки алгоритма:

1. Линейный поиск имеет временную сложность O (N), что, в свою очередь, замедляет его выполнение для больших наборов данных.

2. Не подходит для больших массивов.

Когда использовать алгоритм?

1. Когда мы имеем дело с небольшим набором данных.

2. При поиске набора данных, хранящегося в непрерывной памяти.

1.2 Алгоритм бинарного поиска

Двоичный (бинарный) поиск — это алгоритм поиска, используемый для нахождения позиции целевого значения в отсортированном массиве. Он работает путем многократного деления интервала поиска пополам до тех пор, пока не будет найдено целевое значение или интервал не станет пустым. Интервал поиска делится пополам путем сравнения целевого элемента со средним значением пространства поиска.

Условия применения алгоритма бинарного поиска в структуре данных:

1. Структура данных должна быть отсортирована.
2. Доступ к любому элементу структуры данных должен занимать постоянное время.

Ниже представлен пошаговый алгоритм бинарного поиска:

1. Разделите пространство поиска на две половины, найдя средний индекс «mid» .
2. Сравните средний элемент пространства поиска с ключом .
   (a) Если ключ найден в среднем элементе, процесс завершается.
   (b) Если ключ не найден в среднем элементе, выберите, какая половина будет использоваться в качестве следующего пространства поиска.
   (c) Если ключ меньше среднего элемента, то для следующего поиска используется левая часть.
   (d) Если ключ больше среднего элемента, то для следующего поиска используется правая часть.
3. Этот процесс продолжается до тех пор, пока не будет найден ключ или не будет исчерпан весь объем пространства поиска.

Алгоритм двоичного поиска может быть реализован следующими двумя способами:

1. Алгоритм итерационного бинарного поиска
2. Алгоритм рекурсивного бинарного поиска

Анализ сложности алгоритма бинарного поиска:

1. Лучший случай: O(1)
2. Средний случай: O(log N)
3. Худший случай: O(log N)

Вспомогательное пространство: O(1). Если рассматривать стек рекурсивных вызовов, то вспомогательное пространство будет O(logN).

Применение алгоритма бинарного поиска:

1. Двоичный поиск можно использовать в качестве строительного блока для более сложных алгоритмов, используемых в машинном обучении, таких как алгоритмы обучения нейронных сетей или поиска оптимальных гиперпараметров для модели.
2. Его можно использовать для поиска в компьютерной графике, например, в алгоритмах трассировки лучей или наложения текстур.

3. Его можно использовать для поиска в базе данных.

Преимущества бинарного поиска:

1. Двоичный поиск быстрее линейного поиска, особенно для больших массивов.
2. Более эффективен, чем другие алгоритмы поиска с аналогичной временной сложностью, такие как интерполяционный поиск или экспоненциальный поиск.
3. Двоичный поиск хорошо подходит для поиска в больших наборах данных, хранящихся во внешней памяти, например на жестком диске или в облаке.

Недостатки бинарного поиска:

1. Массив должен быть отсортирован.
2. Двоичный поиск требует, чтобы искомая структура данных хранилась в смежных ячейках памяти.
3. Двоичный поиск требует, чтобы элементы массива были сопоставимы, то есть их нужно упорядочить.

Оценка трудоёмкости для "холодного"старта алгоритма:

$$f_{\text{бин. поиск}} = f_{\text{сорт}} + f_{\text{поиск}}$$

Оценка трудоёмкости алгоритма на отсортированном массиве из N элементов:

$$f_{\text{бин. поиск}} = f_{\text{сорт}} + \begin{cases} O(1) & \text{лучший случай (элемент на 1-ом месте);} \\ O(N) & \text{худший случай (элемент на N-ом месте или отсутствует).} \end{cases} \tag{1}$$

1.3 Сортировка

Виды сортировок

Так как бинарный поиск работает только с отсортированными массивами, необходимо подробнее рассмотреть сортировки.

Сортировка — это процесс упорядочивания элементов в списке по определенному критерию. Существует множество алгоритмов сортировки, и они могут быть классифицированы по различным критериям, таким как производительность, дополнительные по памяти и устойчивость.

Рассмотрим некоторые распространенные виды сортировок:

1. Сортировка пузырьком (Bubble Sort):

   Простой алгоритм, который многократно проходит по списку, сравнивая соседние элементы и меняя их местами, если они находятся в неверном порядке.

   Время выполнения:

(a) O(n^2) в худшем и среднем случаях;

(b) O(n) в лучшем случае (если список уже отсортирован).

2. Сортировка вставками (Insertion Sort):

   Элементы сортируются по одному, вставляя их в правильное положение в уже отсортированной части списка.

   Время выполнения:

   (a) O(n^2) в худшем и среднем случаях;

   (b) O(n) в лучшем случае (при помощи уже отсортированного списка).

3. Сортировка выбором (Selection Sort):

   На каждом проходе выбирается наименьший (или наибольший) элемент и помещается в начало (или конец) списка.

   Время выполнения: O(n^2) для всех случаев.

4. Сортировка слиянием (Merge Sort):

   Алгоритм типа "разделяй и властвуй" , который разбивает список на две части, сортирует их и затем объединяет.

   Время выполнения: $O(n * \log n)$ во всех случаях.

5. Быстрая сортировка (Quick Sort):

   Также алгоритм "разделяй и властвуй" , который выбирает опорный элемент, распределяет элементы по сравнению с опорным и рекурсивно сортирует подсписки.

   Время выполнения:

   (a) $O(n * \log n)$ в среднем случае;

   (b) O(n^2) в худшем случае (например, если массив уже отсортирован).

6. Пирамидальная сортировка (Heap Sort):

   Основан на структуре данных "куча". Алгоритм строит кучу из массива и затем извлекает максимальные элементы, перестраивая кучу.

   Время выполнения: $O(n * \log n)$ во всех случаях.

Сортировка выбором

В связи с простотой реализации, стабильной сложностью и отсутствием необходимости дополнительной памяти для подготовки массива к бинарному поиску решено было реализовать сортировку выбором.

Алгоритм сортировки выбором:

1. Вычислить длину переданного массива;

2. Пройтись по всем элементам массива;

   (a) Запомнить индекс текущего элемента как индекс минимального для текущего шага;

   (b) Пройтись по оставшимся элементам массива;

   Сравнить запомненный минимальный с текущим элементом и запомнить текущий, если он меньше запомненного.

   (c) Обменять элемент с запомненным как минимальный индексом с текущим;

Таким образом трудоёмкость сортировки выбором для массива длиной n будет следующей:

$$f_{\text{сорт}} = 1+2+n\cdot\left(2+1+5+2+(n-i-1)\cdot\left(2+3+\left\{\begin{array}{ll}0, & \text{лучший случай}\\ 1, & \text{худший случай}\end{array}\right.\right)\right) \qquad (2)$$

2 Конструкторская часть

2.1 Описания алгоритмов

На основании теоретических измышлений были разработаны алгоритмы, реализующие линейный и бинарный поиск в массиве. Блок-схемы этих алгоритмов приведены на рисунках 1 (линейный) и 2 (бинарный).

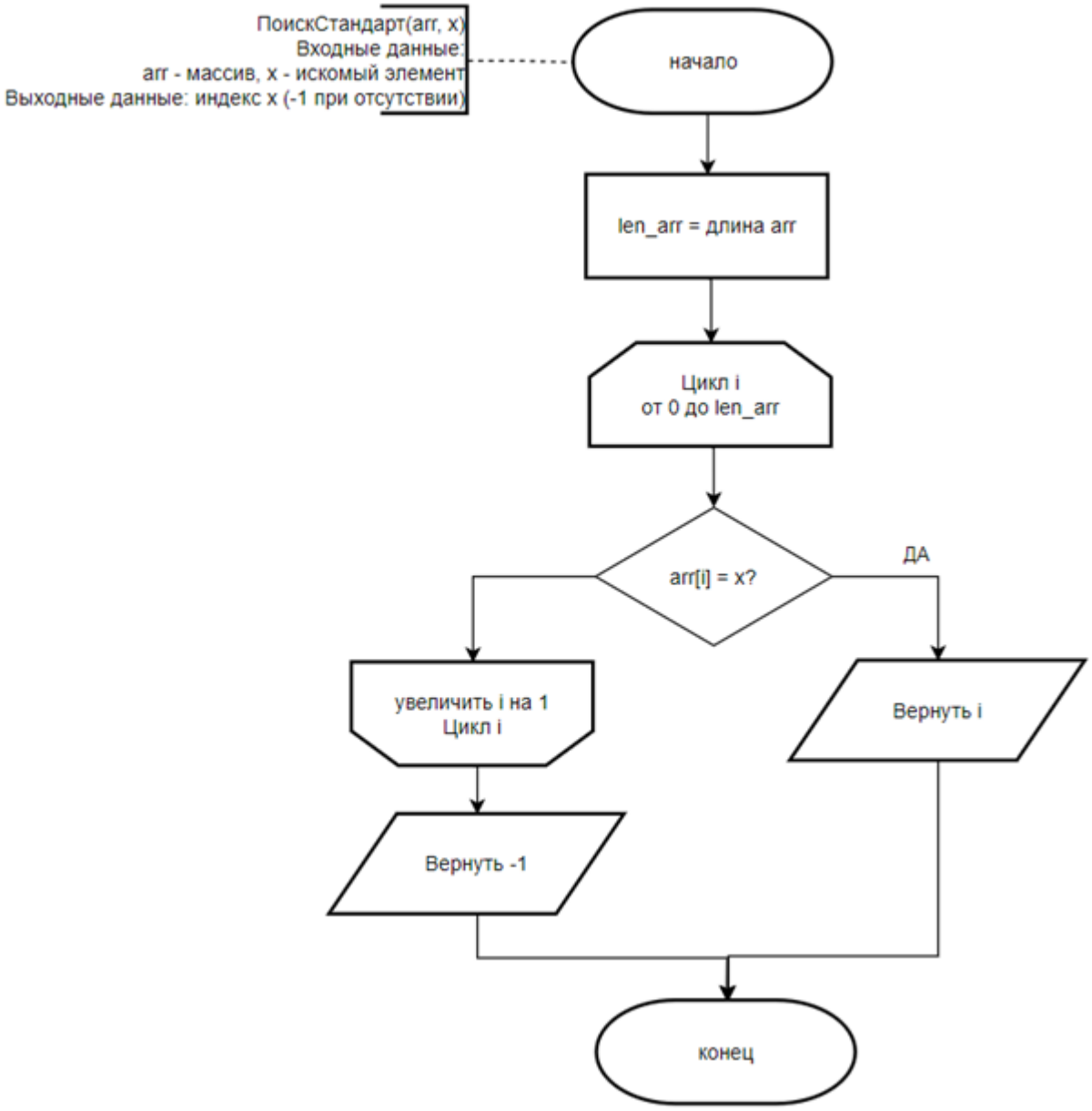

Рисунок 1 – Блок-схема линейного алгоритма поиска в массиве

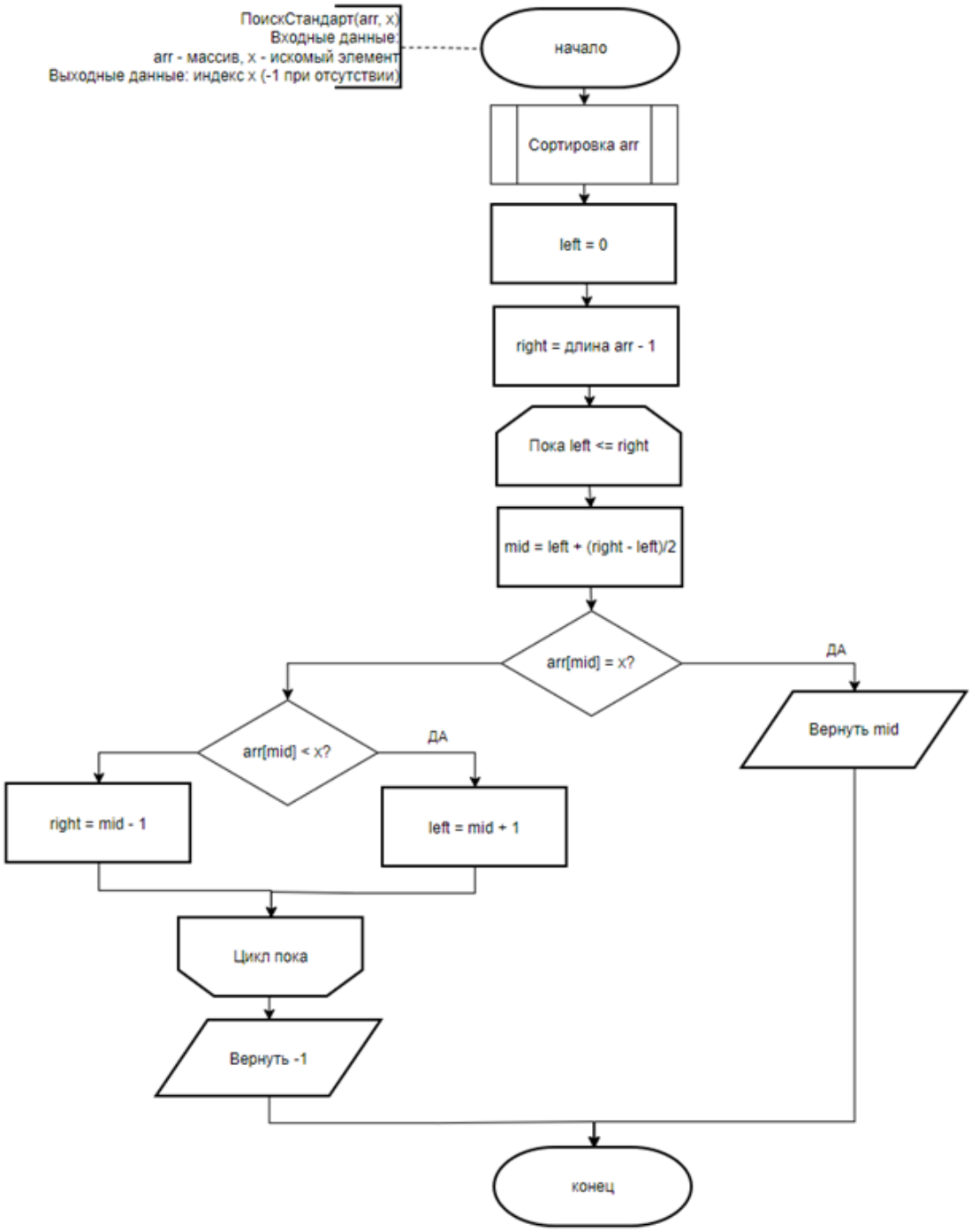

Рисунок 2 – Блок-схема бинарного алгоритма поиска в массиве

3 Технологическая часть

3.1 Требования к программному обеспечению

На вход программе подаются массив из целых чисел и целое число, которое будет искаться в нём.

На выход программа выдаёт индекс найденного в массиве элемента или -1 (на случай отсутствия искомого числа в массиве). Также в зависимости от выбранного пункта меню программа замеряет время работы алгоритмов и рисует получившиеся графики.

3.2 Средства реализации

В связи с опытом реализации подобных задач на Python, программа была реализована на этом языке программирования. Для замеров времени была использована функция `process_time()` из библиотеки `time`, вычисляющая процессорное время [1].

3.3 Реализации алгоритмов

Ниже приведены реализации алгоритмов поиска элементов в массиве линейным способом (листинг 1) и бинарным (листинг 2) на Python.

```
def algo_search_order(arr, x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i
    return -1
```

Листинг 1 – реализация линейного алгоритма поиска в массиве

```
def algo_search_bin(arr, x):
    arr = sorted(arr)
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = left + (right - left) // 2
        if arr[mid] == x:
            return mid
        elif arr[mid] < x:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

Листинг 2 – реализация бинарного алгоритма поиска в массиве

3.4 Тесты

Для тестирования линейного и бинарного алгоритмов поиска элементов в массиве были составлены таблицы с входными данными (массив и искомый элемент), ожидаемым результатом (индексом) и полученным результатом от обоих способов.

Таблица 1 – Таблица тестов для алгоритмов поиска элементов в массиве

Массив	Искомый элемент	Ожидание линейный	Ожидание бинарный	Результат линейный	Результат бинарный
[]	1	-1	-1	-1	-1
[1, 2, 3]	4	-1	-1	-1	-1
[1, 2, 3]	1	0	0	0	0
[1, 2, 3]	2	1	1	1	1
[1, 2, 3]	3	2	2	2	2
[3, 2, 1]	1	2	0	2	0

В ходе проведённого тестирования (с помощью pytest) ошибок в алгоритмах не выявлено (см листинг 3.4).

```
pytest
======================== test session starts ========================
platform win32 -- Python 3.11.9, pytest-8.2.2, pluggy-1.5.0
rootdir: L:\sem_5\\Algorithm_analysis\lab_03
plugins: Faker-28.4.1
collected 12 items

test_algo.py ....................................                [100%]

========================= 12 passed in 0.24s =========================
```

Листинг 3 – тестирование алгоритмов с помощью pytest

4 Исследовательская часть

Для рассмотрения числа сравнений в зависимости от индекса на котором находится элемент для линейного и бинарного поисков элементов в массиве были собранны данные и составлены следующие гистограммы: для линейного поиска (см. рисунок 3), для бинарного (см. рисунок 4) и отсортированный по возрастанию числа сравнений для бинарного (см. рисунок 5). Алгоритмы запускались на массиве длиной 50, элементы которого соответствуют индексам и ищутся один за другим с шагом 1.

Замеры были проведены на процессоре 13-го поколения Intel(R) Core(TM) i5-13500H с тактовой частотой 2.60 ГГц, оперативная память 16,0 ГБ, тип системы 64-разрядная операционная система, процессор x64, версия Python 3.11.9.

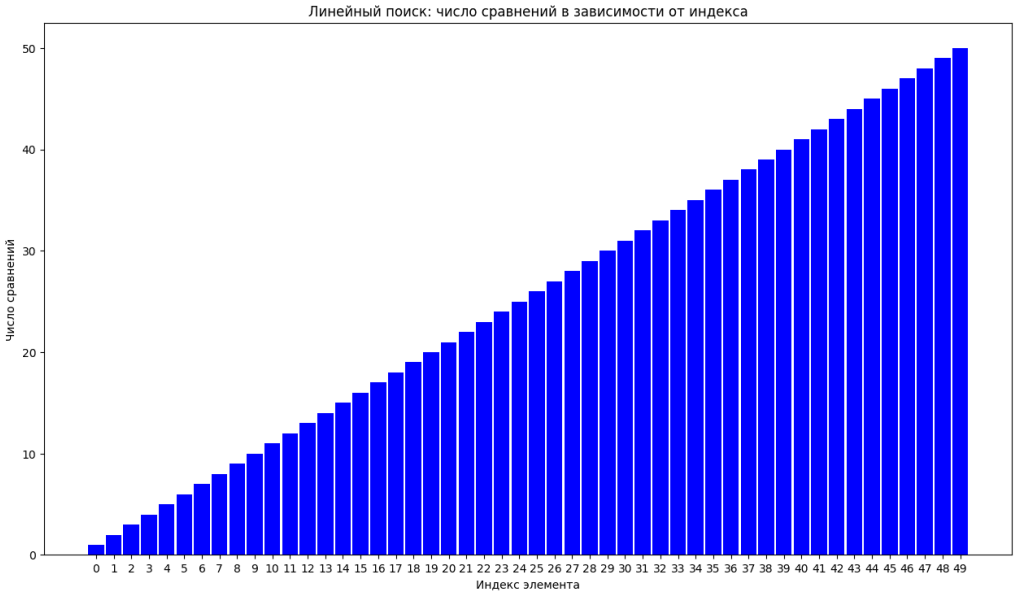

Рисунок 3 – Гистограмма числа сравнений в зависимости от индекса на котором находится элемент для линейного поиска

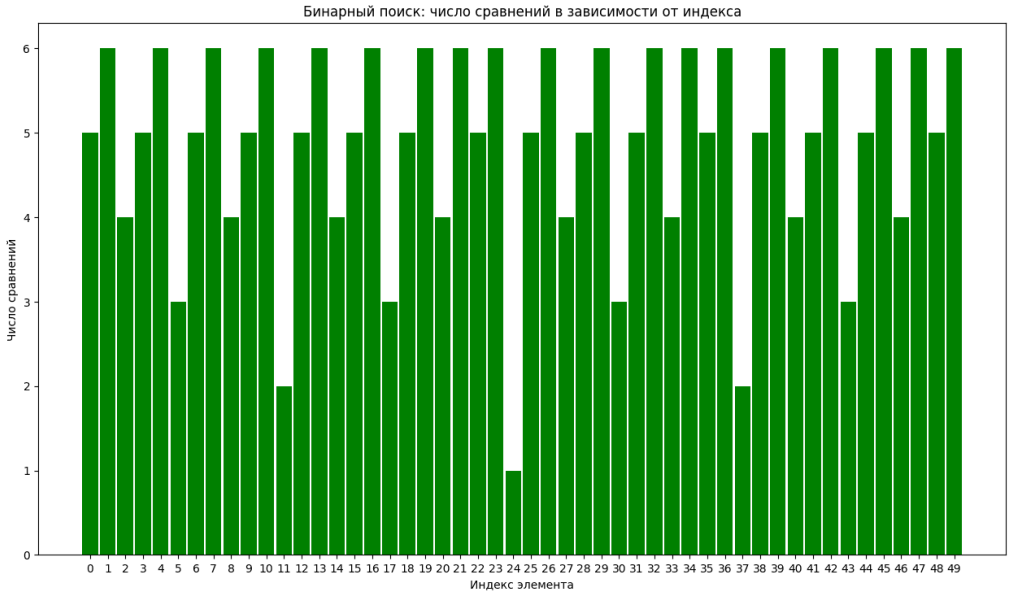

Рисунок 4 – Гистограмма числа сравнений в зависимости от индекса на котором находится элемент для бинарного поиска

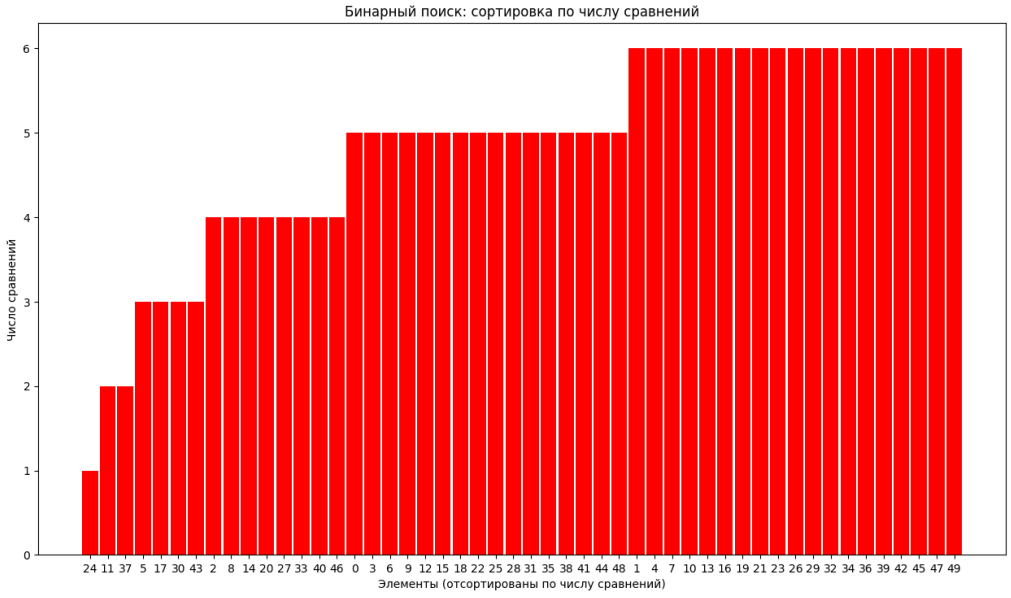

Рисунок 5 – Гистограмма числа сравнений в зависимости от индекса на котором находится элемент для бинарного поиска (отсортированная по возрастанию числа сравнений)

Вывод

Линейный поиск

При линейном поиске сравнение производится последовательно от первого элемента до того, который совпадет с искомым значением. Число сравнений зависит от того, где находится искомый элемент:

1. Если элемент находится на первом индексе, будет сделано 1 сравнение.
2. Если на втором, то 2 сравнения и так далее.
3. Если элемент отсутствует, требуется сравнить все 50 элементов.

Таким образом, для элемента на индексе i число сравнений равно $i+1$.

Бинарный поиск

При бинарном поиске массив делится на две части, и продолжается поиск в соответствующей половине, в зависимости от того, меньше или больше искомое значение среднего элемента. Число сравнений можно оценить как логарифм числа элементов массива:

$$\text{Число сравнений} \approx \log_2(n) \tag{3}$$

где n — текущее количество элементов в части массива, где продолжается поиск.

Для массива из 50 элементов число сравнений для поиска элемента на разных индексах будет следующим:

число шагов при бинарном поиске для массива из 50 элементов — это округлённое значение $log_2(50)$, что примерно равно 6. Таким образом, бинарный поиск сделает около 6 сравнений для нахождения элемента.

Сравнение по индексам:

Линейный поиск:

1. Элемент на индексе 0: 1 сравнение.
2. Элемент на индексе 1: 2 сравнения.
3. Элемент на индексе 25: 26 сравнений.
4. Элемент на индексе 49: 50 сравнений.

Бинарный поиск: Независимо от того, где находится элемент, максимальное количество сравнений не превысит 6 для массива из 50 элементов.

Вывод

Линейный поиск имеет линейную сложность $O(n)$, и число сравнений прямо зависит от позиции элемента. Бинарный поиск имеет логарифмическую сложность $O(\log n)$ и обеспечивает гораздо меньшее число сравнений, особенно для элементов, находящихся ближе к концу массива, однако он работает только на отсортированных массивах, а сортировка может компенсировать выигрыш от более быстрого поиска.

ЗАКЛЮЧЕНИЕ

В результате выполнения лабораторной работы были исследованы различные алгоритмы поска элемента в массиве (линейный и бинарный алгоритмы).

В частности:

1. рассмотрен линейный алгоритм поиска элемента в массиве;
2. рассмотрен бинарный алгоритм поиска элемента в массиве;
3. реализованы оба алгоритма поиска элемента в массиве на выбранном языке программирования;
4. сравнена эффективность этих алгоритмов на практике.

В ходе лабораторной работы были рассмотрены, спроетированны и запрограммированы линейный и бинарный алгоритмы поиска элемента в массиве.

Были разработаны тесты для всех алгоритмов, учитывающие крайние случаи, ожидаемых результатов которых достигли все реализации.

Сравнения полученных программ показали, что бинарный алгоритм работает быстрее линейного, так как в большинстве случаев делает меньше сравнений, но, в связи с тем, что для бинарного поиска нужен отсортированный массив, такого выигрыша по времени не удастся добиться на несортированных данных.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Python Documentation. time.process_time() - Документация по стандартной библиотеке Python. Дата обращения: 03 сентября 2024 г. [Электронный ресурс]. Доступно по адресу: https://docs-python.ru/standart-library/modul-time-python/funktsija-process-time-modulja-time/

[2] Г. Лорин. Сортировка и системы сортировки. Перевод с английского. Ред. Я. Я. Васина, В. Ю. Королев. Москва: Наука, 1983.