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

Студент гр. 3354 _____ Кротов И. С.

Преподаватель _____ Пестерев Д. О.

Санкт-Петербург
2024

Задание на курсовую работу

Студент Кротов И. С.

Группа 3354

Тема работы: Сравнение различных алгоритмов поиска в массиве

Исходные данные:

Язык программирования: Python

ПО для разработки: PyCharm

Содержание пояснительной записки: «Содержание», «Аннотация», «Введение», «Теоретическая часть», «Практическая часть», «Раздел с 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:

Студент гр. 3354 _____ Кротов И. С.

Преподаватель _____ Пестерев Д. О.

Аннотация

В курсовой работе будут рассмотрены несколько алгоритмов поиска элемента в массиве с целью их сравнения друг с другом, чтобы узнать, какой из них в каком случае эффективнее использовать, почему именно их лучше выбирать для той или иной задачи.

Список рассматриваемых алгоритмов: линейный поиск, бинарный поиск, интерполяционный поиск.

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 the course work, several array search algorithms will be considered to compare them with each other to 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 list of algorithms under consideration: linear search, binary search, interpolation search.

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

Содержание

Введение

Эффективный поиск данных – одна из наиболее важных задач в информатике и программировании, ведь от скорости поиска данных зависит производительность.

Умение выбрать подходящий алгоритм для конкретной задачи является важным навыком для разработчика. Именно правильно подобранный алгоритм отличает быстрое, надежное и стабильное приложение от приложения, которое может упасть от любого простого запроса.

Цель работы: изучение теоретической основы алгоритмов поиска в массиве, реализация различных алгоритмов поиска в массиве, исследование их временной и пространственной сложностей, сравнение друг с другом и анализ их производительности на примере неотсортированного, отсортированного и отсортированного упорядоченного массивов.

Для работы были выбраны 3 алгоритма поиска данных в массиве:

1. Линейный поиск
2. Бинарный поиск
3. Интерполяционный поиск

Теоретическая часть

Линейный поиск

Линейный поиск – это алгоритм поиска в произвольном массиве, основанный на последовательном сравнении всех элементов массива с заданным ключом. Является простейшим алгоритмом поиска в массиве.

Алгоритм работает по принципу перебора: каждый элемент массива сравнивается с искомым значением. Если элемент найден, возвращается его индекс, иначе - -1.

Пространственная сложность:

$S(n) = 4 = \theta(1)$

Временная сложность:

$T_{best}(n) = 1 + 1 + 1 + 2 = 5 = \theta(1)$

$T_{worst}(n) = 1 + 2n + n + 2 = 3n + 3 = \theta(n)$

$T_{average}(n) = 1 + 2\frac{n}{2} + \frac{n}{2} + 2 = 1 + n + \frac{n}{2} + 2 = 3 + \frac{3}{2}n = \theta(n)$

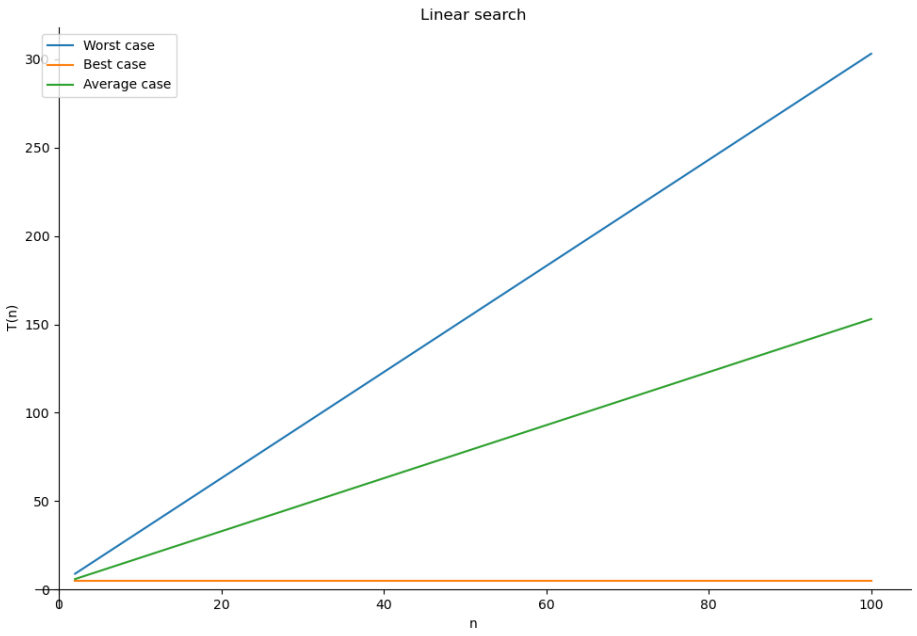

Рисунок 1 – График временной сложности линейного поиска

Бинарный поиск

Бинарный поиск — это более эффективный алгоритм, который работает только с отсортированными списками. Он последовательно делит массив пополам и ищет ключ в одной из частей, в зависимости от значения (если ключ больше, чем средний элемент, то в правой части, иначе – в левой, и так до тех пор, пока либо средний элемент не будет равен ключу, либо не переберется весь массив).

Пространственная сложность:

$S(n) = 4 + 4 + 4 = 12 = \theta(1)$

Временная сложность:

$T_{best}(n) = 1 + 2 + 1 + 1 + 1 = 6 = \theta(1)$

Для нахождения функции временной сложности для худшего случая используем мастер-теорему:

$T_{worst}(n) = T_{worst}(n/2) + f(n)$, где $f(n)$ – время, необходимое для разбиения массива на 2 части.

$f(n) = 1 + 1 + 1 + 1 = 4$, тогда $T_{worst}(n) = T_{worst}(n/2) + 4$

$T_{worst}(n) = T_{worst}\left(\frac{n}{4}\right) + 4 + 4 = T_{worst}\left(\frac{n}{8}\right) + 4 + 4 + 4 =$

$T_{worst}(n/2^k) + k * 4$, где k – количество делений массива на 2 части.

Продолжаем до тех пор, пока не дойдем до одного элемента

$T_{worst}(n/2^k) = T_{worst}(1)$:

$T_{worst}(n) = T_{worst}(1) + k * 4$;

$T_{worst}(1) = 1 + 1 + 1 = 3$

$\frac{n}{2^k} = 1$, тогда $k = \log_2 n$

Подставим результаты:

$T_{worst}(n) = 4 + 4 * \log_2 n = \theta(\log n)$

Чтобы рассчитать средний случай бинарного поиска, воспользуемся формулой математического ожидания:

$$T_{average}(n) = = \frac{1}{n}\sum_{i=1}^{\log_2 n} i * 2^{i-1} = \frac{1}{n}\left(1 + 4 + 12 + 32 + \cdots + \log_2 n * 2^{\log_2 n}\right)$$

$$\leq \frac{1}{n}\log_2 n = \theta(\log n)$$

$$T_{average}(n) = \theta(\log n)$$

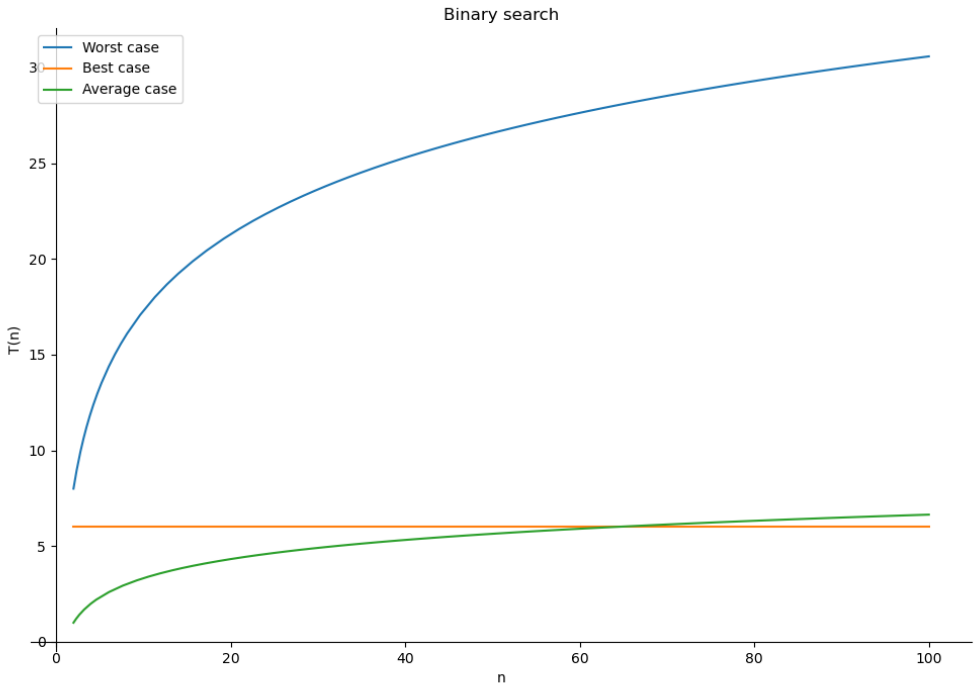

Рисунок 2 – График временной сложности бинарного поиска

Интерполяционный поиск

Интерполяционный поиск — это алгоритм поиска в отсортированном массиве, который основан на предсказании индекса искомого элемента, по интерполирующей элементы массива функции y = f(i), где i – индекс элемента. Является ещё одним алгоритмом типа «разделяй и властвуй», аналогичным бинарному поиску.

Пространственная сложность:

$S(n) = 4 * 2 = 8 = \ \theta(1)$

Временная сложность:

$T_{best}(n) = 1 \ + \ 2 \ + \ 3 \ + \ 1 \ + \ 1 \ + \ 2 \ = \ 10 = \ \theta(1)$

При использовании линейной функции для интерполяции элементов худшим случаем является неравномерно распределенные элементы, например, если элементы описывают экспоненциальную функцию: $y = f(i) = ae^{bi} + c$ и искомый ключ находится в точке выпуклости графика $f(i)$, в таком случае мы будем последовательно сдвигать назад крайний индекс, пока не дойдем до искомого элемента, следовательно:

$T_{worst}(n) = \theta(n)$

В среднем случае на каждой итерации цикла количество элементов будет уменьшаться до корня количества элементов на предыдущей итерации, тогда, используя мастер теорему, можно составить рекуррентное соотношение:

$T_{average}(n) = T_{average}\left(\sqrt{n}\right) + f(n), f(n) = 3 + 5 = 8$

$T_{average}(n) = T_{average}\left(\sqrt{n}\right) + 8 = T_{average}\left(n^{1/2^2}\right) + 2 * 8$

$= T_{average}\left(n^{\frac{1}{2^3}}\right) + 3 * 8 = \cdots = T_{average}\left(n^{\frac{1}{2^k}}\right) + k * 8$

Поиск заканчивается, когда остается 2 элемента, т. е.

$n^{\frac{1}{2^k}} = 2; \frac{1}{2^k} = \ \log_n 2 = \frac{1}{\log_2 n}; k = \ \log_2(\log_2 n); T_{average}(2) = 7$

$T_{average}(n) = \ T_{average}(2) + \ 8 * \log_2(\log_2 n) = \ \theta(\log(\log n))$

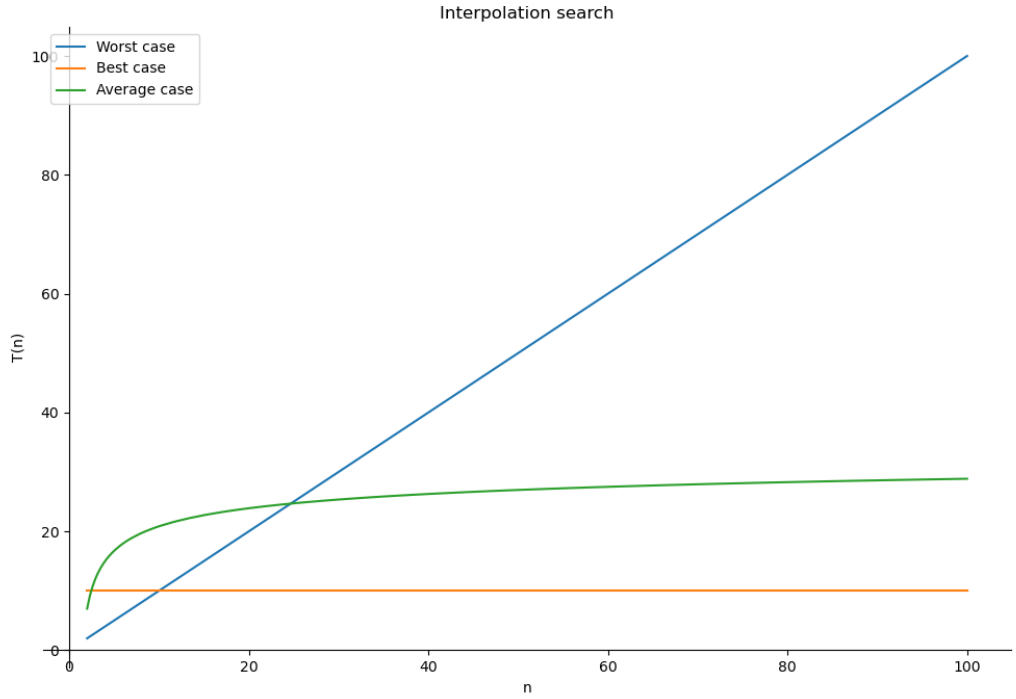

Рисунок 3 – График временной сложности интерполяционного поиска

Итоговая таблица асимптотик алгоритмов поиска

Алгоритм	Асимптотическая временная сложность			Асимптотическая пространственная сложность
	T_{best}	$T_{average}$	T_{worst}	
Линейный поиск	$\theta(1)$	$\theta(n)$	$\theta(n)$	$\theta(1)$
Бинарный поиск	$\theta(1)$	$\theta(\log n)$	$\theta(\log n)$	$\theta(1)$
Интерполяционный поиск	$\theta(1)$	$\theta(\log(\log n))$	$\theta(n)$	$\theta(1)$

Практическая часть

В практической части работы будет произведено сравнение алгоритмов поиска, чтобы выявить, какие из них эффективнее и в какой ситуации.

Для проверок будем брать массив, состоящий из 1000000 чисел, перемешаем его, создадим второй, который будет отсортированной копией первого, а затем с шагом в 10000 чисел будем увеличивать часть массива, в которой будет происходить поиск, запуская секундомер, который зафиксирует время поиска элемента (который будет выбран из чисел массива). Для каждого полученного набора данных составим аппроксимированные графики зависимости времени от размера. Оси абсцисс соответствуют количеству элементов в массиве, оси ординат — времени выполнения в наносекундах.

Уравнение регрессионной кривой имеет вид $ax\log(x) + bx$

Сравнение отсортированного/неотсортированного массивов

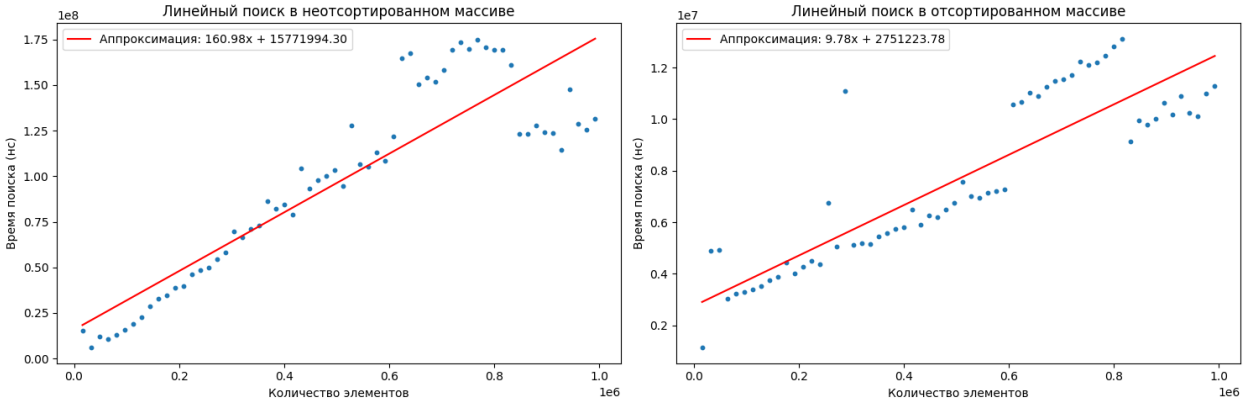

Рисунок 4 – Поиск элемента в отсортированном/неотсортированном массивах с помощью линейного поиска

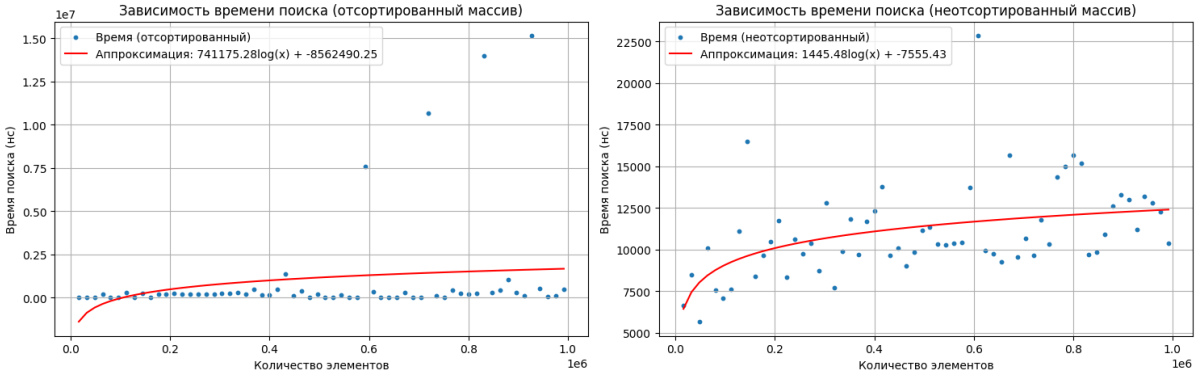

Рисунок 5 – Поиск элемента в отсортированном/неотсортированном массивах с помощью бинарного поиска

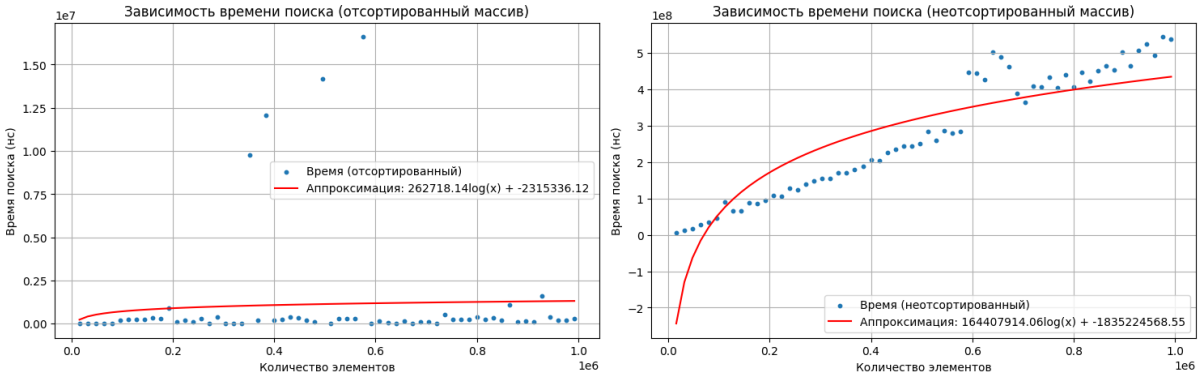

Рисунок 6 – Поиск элемента в отсортированном/неотсортированном массивах с помощью интерполяционного поиска

По полученным графикам видно, что все алгоритмы, кроме линейного, действительно работают лучше на отсортированных массивах. Оно и неудивительно, ведь бинарный и интерполяционный поиски изначально рассчитаны на работу с отсортированными массивами. Поэтому переходим в следующий раздел, в котором произведём сравнение алгоритмов, эффективных только на отсортированных массивах.

Сравнение алгоритмов, эффективных только на отсортированных массивах

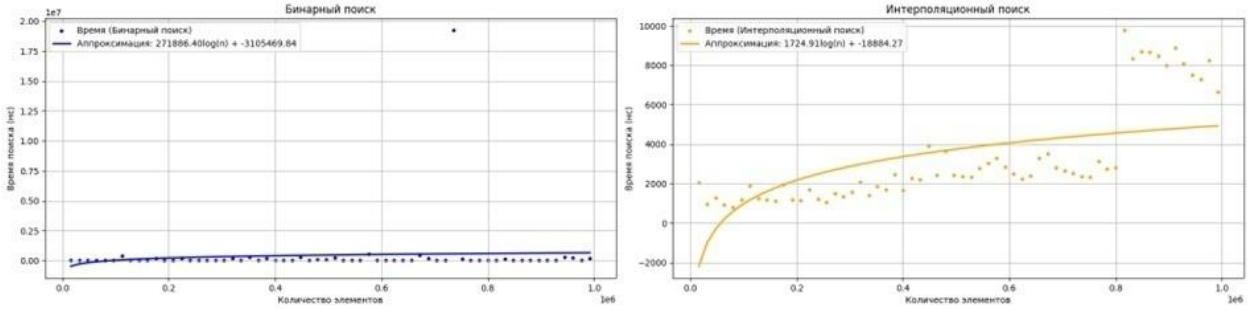

Рисунок 7 – Поиск элемента в отсортированном массиве с помощью бинарного и интерполяционного поисков

По полученным графикам видно, что в одинаковых условиях бинарный поиск всё же работает быстрее интерполяционного. Таким образом, можно сделать вывод, что для поиска элемента в отсортированных массивах бинарный поиск чаще является наиболее эффективным выбором.

Сравнение алгоритмов на упорядоченном отсортированном массиве

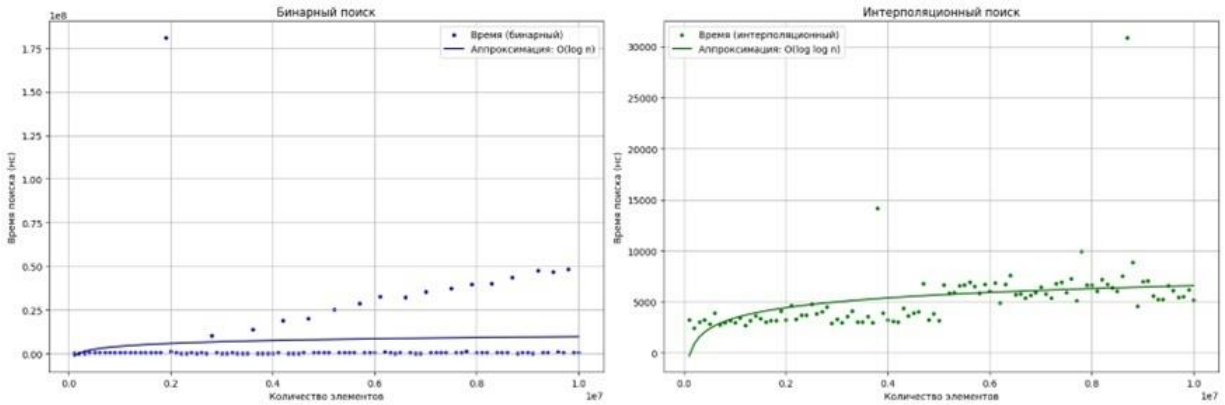

Рисунок 7 – Поиск элемента в упорядоченном отсортированном массиве с помощью бинарного и интерполяционного поисков

По полученным графикам видно, что если данные будут равномерно распределены, то выгоднее будет интерполяционный поиск, но это требует особых условий.

Раздел с кодом

Searches.py

```
def linear_search(A, target):
    for i, num in enumerate(A):
        if num == target:
            return i
    return -1

def binary_search(A, target):
    first = 0
    last = len(A) - 1
    index = -1
    while (first <= last) and (index == -1):
        mid = (first + last) // 2
        if A[mid] == target:
            index = mid
        else:
            if target < A[mid]:
                last = mid - 1
            else:
                first = mid + 1
    return index

def interpolation_search(A, target):
    left, right = 0, len(A) - 1
    while left <= right and A[left] <= target <= A[right]:
        if left == right:
            if A[left] == target:
                return left
            return -1
```

```
    estimated_pos = left + ((target - A[left]) * (right - left)) // (A[right] - A[left])

    if A[estimated_pos] == target:
        return estimated_pos
    elif A[estimated_pos] < target:
        left = estimated_pos + 1
    else:
        right = estimated_pos - 1
return -1
```

Main.py

```
from searches import *
import matplotlib.pyplot as plt
import numpy as np
import random
import time

Numbers_list = list(range(1000000))
random.shuffle(Numbers_list)
Data = []
target_value = random.choice(Numbers_list)
Sorted_numbers_list = sorted(Numbers_list)

for i in range(10000, 1000001, 10000):
    testing_array = Numbers_list[:i]
    sorted_testing_array = Sorted_numbers_list[:i]

    start_time = time.perf_counter_ns()
    binary_search(sorted_testing_array, target_value)
    sorted_time = time.perf_counter_ns() - start_time

    start_time = time.perf_counter_ns()
    binary_search(testing_array, target_value)
    unsorted_time = time.perf_counter_ns() - start_time

    Data.append((i, sorted_time, unsorted_time))

sizes, sorted_times, unsorted_times = zip(*Data)
coefficients_sorted = np.polyfit(np.log(sizes), sorted_times, 1)
coefficients_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, coefficients_sorted[0] * np.log(sizes) + coefficients_sorted[1],
color='red',
        label=f"Аппроксимация: {coefficients_sorted[0]:.2f}log(x) +
{coefficients_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, coefficients_unsorted[0] * np.log(sizes) + coefficients_unsorted[1],
color='red',
        label=f"Аппроксимация: {coefficients_unsorted[0]:.2f}log(x) +
{coefficients_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

В ходе выполнения курсовой работы были рассмотрены 3 алгоритма поиска элемента в массиве: линейный поиск, бинарный поиск и интерполяционный поиск. Сравнивая практические графики с теорией можно сделать вывод, что графики согласуются с теорией и демонстрируют ожидаемую асимптотик. Также можно сказать, что каждый поиск в зависимости от заполнения массива и в зависимости от местоположения искомого ключа, имеет свои особенности и однозначно сказать какой из них лучше для произвольного случая нельзя.

Общие выводы, которые можно сделать:

1. Если необходимо найти элемент в несортированном массиве или первое вхождение искомой переменной, то для выполнения задачи лучше всего подойдёт линейный поиск.
2. Если необходимо выполнить поиск в отсортированном массиве, то лучшим вариантом для достижения результата будет бинарный поиск.
3. Если отсортированный массив равномерно распределён, то самым быстрым и эффективным будет уже не бинарный поиск, а интерполяционный поиск.

Ссылка на репозиторий: https://github.com/Kr-Van/AISD_CourseWork/