Отчёт: лабораторная работа №1

Студент: Золин И. М.
Группа: М4245
GitHub: https://github.com/IMZolin/gen-algs-lab1

Алгоритм: бинарный поиск
Задание: получить навыки вычисления сложности алгоритмов и их оптимизации различными методами.

1. Вычисление сложности алгоритма

Входные данные – отсортированный массив.

Рекурсивный подход

На каждом шаге вычисляется середина интервала и сравнивается с искомым значением. Если элемент найден — возвращается индекс; иначе поиск продолжается в левой или правой части массива.

Временная сложность: $O(\log_2 N)$

- $T(N) = c + T(\frac{N}{2})$
- $T\left(\frac{N}{2}\right) = c + T(\frac{N}{4}) \Rightarrow T(N) = T\left(\frac{N}{4}\right) + 2c$
- $T\left(\frac{N}{4}\right) = c + T\left(\frac{N}{8}\right) \Rightarrow T(N) = T\left(\frac{N}{8}\right) + 3c$
- $T(N) = T\left(\frac{N}{2^k}\right) + k \cdot c$
- $T\left(\frac{N}{2^i}\right) = T(1) \Rightarrow \frac{N}{2^k} = 1; \Rightarrow N = 2^k \Rightarrow \log_2 N = k$
- $T(N) = T\left(\frac{N}{2^{\log_2 N}}\right) + c \cdot \log_2 N = T\left(\frac{N}{N}\right) + c \cdot \log_2 N$
- $T(N) = T(1) + c \cdot \log_2 N \rightarrow O(\log_2 N)$

Память: $O(\log_2 N)$

Каждый рекурсивный вызов добавляет один фрейм в стек вызовов (пара чисел + локальные переменные)

Итеративный подход

Временная сложность: $O(\log_2 N)$

На каждой итерации интервал сужается вдвое. Цикл завершается при размере интервала ≤ 1.

- $\frac{N}{2^k} \leq 1 \Rightarrow 2^{\mathrm{k}} \geq N \Rightarrow \mathrm{k} \geq \log_2 N$
- $T(N) = T(1) + c \cdot \log_2 N \rightarrow O(\log_2 N)$

Память: $O(1)$

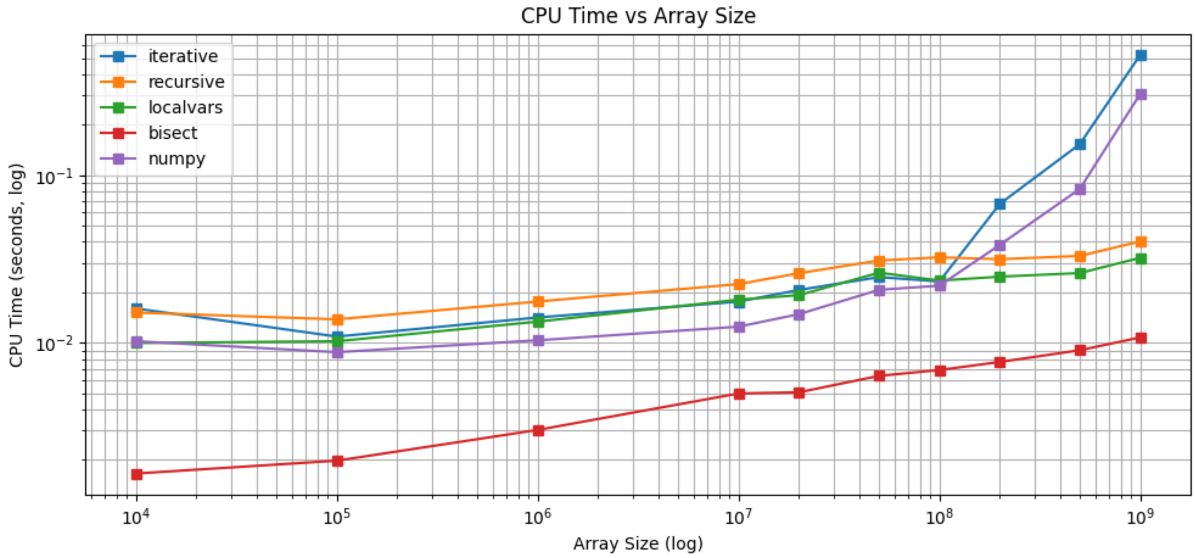
2. Оптимизация алгоритма

- **Оптимизация локальных переменных (Local Vars, CPython).** В CPython локальные переменные хранятся в компактных C-массивах, доступ к ним быстрее, чем к глобальным (которые хранятся в словаре). Плюс отсутствуют операции поиска в глобальной области.
- **NumPy.** Векторизованная реализация на C. Проблема: для обычного списка требуется преобразование в np.array, что добавляет накладные расходы при больших размерах.
- **Bisect.** Модуль bisect реализован на C и выполняет бинарный поиск за один вызов C-функции, без циклов Python. Константа времени существенно ниже (порядка ×10 быстрее обычной версии).

3. Результаты нагрузочных тестов

Метод	10k	100k	1M	10M	20M	50M	100M	200M	500M
iterative	0.0155	0.0111	0.0142	0.0176	0.0207	0.0246	0.0232	0.0678	0.3192
recursive	0.0151	0.0139	0.0176	0.0223	0.0263	0.0312	0.0330	0.0316	0.0336
local vars	0.0099	0.0102	0.0134	0.0183	0.0193	0.0264	0.0235	0.0249	0.0261
NumPy	0.0102	0.0089	0.0104	0.0125	0.0149	0.0208	0.0219	0.0386	0.0386
bisect	**0.0016**	**0.0019**	**0.0030**	**0.0051**	**0.0051**	**0.0064**	**0.0069**	**0.0077**	**0.0096**

- Итеративная версия масштабируется хорошо, но скорость ограничена накладными расходами Python-цикла. На 200M и 500M видно резкое увеличение времени.
- Рекурсивная версия медленнее итеративной на 10–30% из-за стоимости вызовов функций.
- Local vars стабильно быстрее, чем итеративный на 30-40%. Стабильная производительность даже при больших размерах.
- NumPy эффективен только при работе с существующими NumPy-массивами. При преобразовании из Python-списков возникают накладные расходы.
- Bisect – наиболее быстрая реализация. Работа происходит полностью в C, Python выполняет только единичный вызов функции.
- Память остается постоянной, за исключением небольших колебаний, т. к. для списков Python память уже выделена. При использовании NumPy наблюдаются случайные всплески, т. к. при создании массива NumPy используются большие непрерывные буферы.



4. Выводы
    - Все вариации алгоритма бинарного поиска имеют сложность. Но константы сильно отличаются от подхода.
    - Ни один алгоритм не выделяет дополнительную память, кроме рекурсии.
    - NumPy полезен только тогда, когда массив в формате NumPy, в противном случае увеличиваются затраты на преобразование.
    - Лучшая вариация – bisect. Самый быстрый по всем входным данным.