СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Для человека информация всегда была практически тем, от чего зависит его жизнь. Именно на основании того, какую информацию и насколько быстро человек её получает, принимаются те или иные решения. А со временем количество информации становится только больше. Отсюда и возникает острая необходимость анализировать и отбирать только уникальные данные, которые нужны человеку.

Алгоритмы поиска наборов данных используются повсеместно: при устранении нарушений целостности структуры различных баз данных (интернет-магазины, финансовый организации, БД клиентов); при компиляции программы на языке Си (таблица идентификаторов). Особенно часто проблема дубликации возникает при работе с данными, за которые отвечает человек. Здесь присутствует банальный человеческий фактор (работник может отвлечься, забыть о том, что внёс одни и те же данные несколько раз или даже умышленно допустить ошибку).

Вообще, задача поиска дубликатов в некоем информационном массиве относится к области анализа структур данных и алгоритмов. И важно, чтобы исследуемый алгоритм был эффективен как по времени, так и по памяти; в противном случае, пользователь не будет готов столкнуться с медленной или «прожорливой» программой.

Целью данной работы является обзор существующих методов поиска дубликатов в наборе данных, определение оптимального среди них. Для достижения поставленной цели необходимо решить ряд задач:

1) рассмотреть проблему дубликации данных;
2) описать существующие алгоритмы поиска дубликатов;
3) провести всесторонний анализ таких алгоритмов;
4) определить оптимальный среди рассмотренных.

1 Анализ предметной области

1.1 Определение понятия дубликата

Дубликат, копия — объект, созданный по образцу другого, повторяющий, воспроизводящий его внешний вид и другие свойства [1].

Применительно к области информатики, дубликатом является некоторый объект, который содержится в наборе данных два и более раз. Например, в массиве, состоящем из элементов: «100», «10», «200», «20», «0.5», «11», «300», «30», «0.5», «400», «1», «31», «0.5» — значение «0.5» является дубликатом, так как встречается в наборе данных больше одного раза.

В таблице 1 приведён пример базы данных, в которой находится две одинаковых записи. В ней атрибут «ID» является первичным ключом. Идентификатор со одинаковым значением «2» встречается в таблице больше одного раза, таким образом появляется дубликат.

Таблица 1 — Пример базы данных с дубликатами

ID	Департамент	ФИО	Дата	Статус
1	ИТ	Иванов Иван	2020-01-15	Больничный
2	ИТ	Иванов Иван	2020-01-16	На работе
3	ИТ	Иванов Иван	2020-01-18	На работе
4	ИТ	Иванов Иван	2020-01-19	Оплачиваемый отпуск
5	ИТ	Иванов Иван	2020-01-20	Оплачиваемый отпуск
6	Бухгалтерия	Петрова Ирина	2020-01-15	Оплачиваемый отпуск
7	Бухгалтерия	Петрова Ирина	2020-01-16	На работе
8	Бухгалтерия	Петрова Ирина	2020-01-17	На работе
9	Бухгалтерия	Петрова Ивановна	2020-01-18	На работе
10	Бухгалтерия	Петрова Ирина	2020-01-19	Оплачиваемый отпуск
11	Бухгалтерия	Петрова Ирина	2020-01-30	Оплачиваемый отпуск
2	ИТ	Иванов Иван	2020-01-16	На работе

Стоит отметить, что для баз данных важно содержать в таблицах исключительно уникальные строки, так как иначе нарушается условие целостности сущностей [2].

1.2 Методы поиска дубликатов в наборах данных и их оценка

К поиску дубликатов могут быть применены различные алгоритмы. Дубликаты могут быть как простыми объектами (число, слово, строка), так и сложными (фрагмент текста, запись в базе данных). Поэтому применимость конкретного алгоритма зависит от представленных данных (хотя подход к выявлению целой группы объектов, образующей дубликат, не сильно отличается от подхода к выявлению единичного объекта).

Что касается оценки алгоритмов поиска дубликатов, то рассматривается два параметра нотации «О»: временная и пространственная сложности алгоритма. Временная сложность алгоритма (в худшем случае) — это функция от размера входных данных, равная максимальному количеству элементарных операций, проделываемых алгоритмом для решения экземпляра задачи указанного размера. По аналогии с временной сложностью, определяют пространственную сложность алгоритма, только в этом случае говорится не о количестве элементарных операций, а об объёме используемой памяти [3].

Сложность алгоритмов обычно оценивают по времени выполнения или по используемой памяти [4]. В обоих случаях сложность зависит от размеров входных данных: массив из 100 элементов будет обработан быстрее, чем аналогичный из 1000. При этом точное время зависит от процессора, типа данных, языка программирования и множества других параметров. Важна лишь асимптотическая сложность, т. е. сложность при стремлении размера входных данных к бесконечности.

Допустим, некоторому алгоритму нужно выполнить $4n^3 + 7n$ условных операций, чтобы обработать n элементов входных данных. При увеличении n

на итоговое время работы будет значительно больше влиять возведение n в куб, чем умножение его на 4 или же прибавление $7n$. Тогда говорят, что временная сложность этого алгоритма равна $O(n^3)$, т. е. зависит от размера входных данных кубически.

1.3 Критерии оптимальности алгоритма

Понятие оптимальности алгоритма тесно зависит от типа входного набора данных (списки, массивы и т. д.) и даже от модели ЭВМ. Строго говоря, алгоритм является оптимальным, если любой другой алгоритм, решающий данную задачу, работает не быстрее данного [6].

Алгоритм поиска дубликатов в наборе данных не должен каким-либо образом изменять входные данные, так как их сохранность критически важна для программы.

Для нахождения оптимального алгоритма необходимо рассмотреть и проанализировать существующие методы решения задачи поиска дубликатов.

2 Анализ существующих алгоритмов

2.1 Brute Force

Называемый «грубой силой», или же «решением в лоб», такой алгоритм решает задачу поиска дубликатов путём прямого сравнения пар рядом стоящих элементов в наборе a [4] (схема алгоритма на рисунке 1).

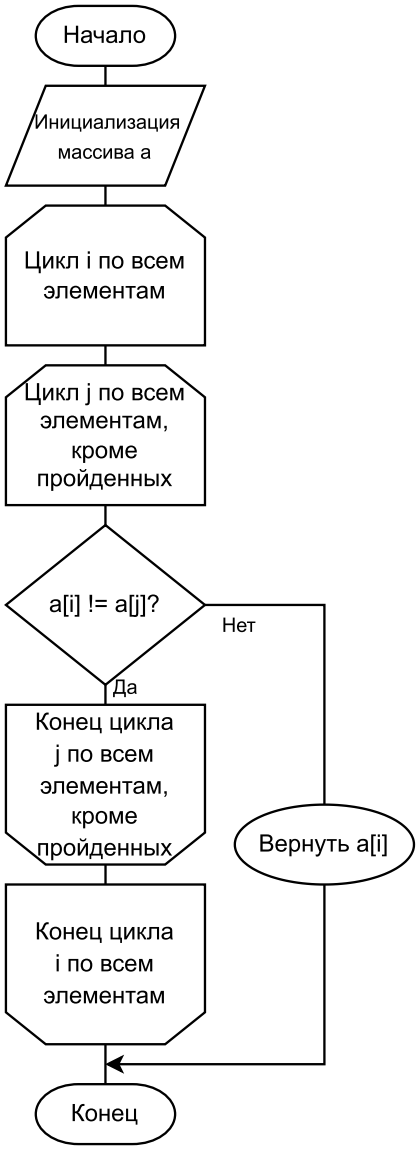

Рисунок 1 — Схема алгоритма «Brute Force»

Алгоритм «Brute force» имеет вычислительную сложность $O(n^2)$ [3]. Это доказывает рисунок 2, на котором отображена зависимость времени выполнения алгоритма от размера набора входящих данных.

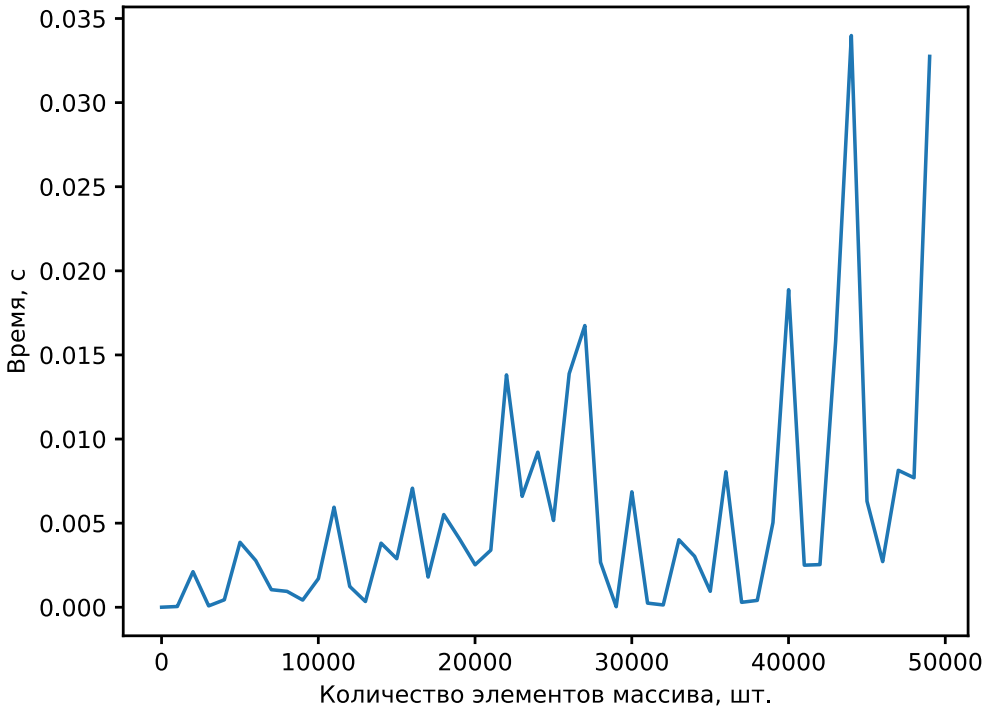

Рисунок 2 — Зависимость времени выполнения алгоритма от размера набора входных данных

Пространственная сложность алгоритма — $O(1)$, т. к. из-за того, что количество операций сравнения двух элементов не меняется в процессе прохода по набору данных [3].

Достоинством этого алгоритма является простота, однако из-за квадратичной вычислительной сложности он становится малоприменимым на практике.

2.2 Метод подсчёта итераций

Этот подход заключается в том, чтобы иметь структуру данных, в которой можно пересчитать количество итераций каждого целочисленного элемента (схема на рисунке 3). Такой метод подходит как для массивов, так и для хэш-таблиц [6].

Рисунок 3 — Схема алгоритма

Алгоритм имеет вычислительную и пространственную сложности $O(n)$ [3] (рисунок 4), что не является приемлемым показателем, хоть он и не требует сложных вычислений. Кроме того, метод подсчёта итераций не подходит для поиска дубликатов в наборе данных, который не состоит из элементов целочисленного типа.

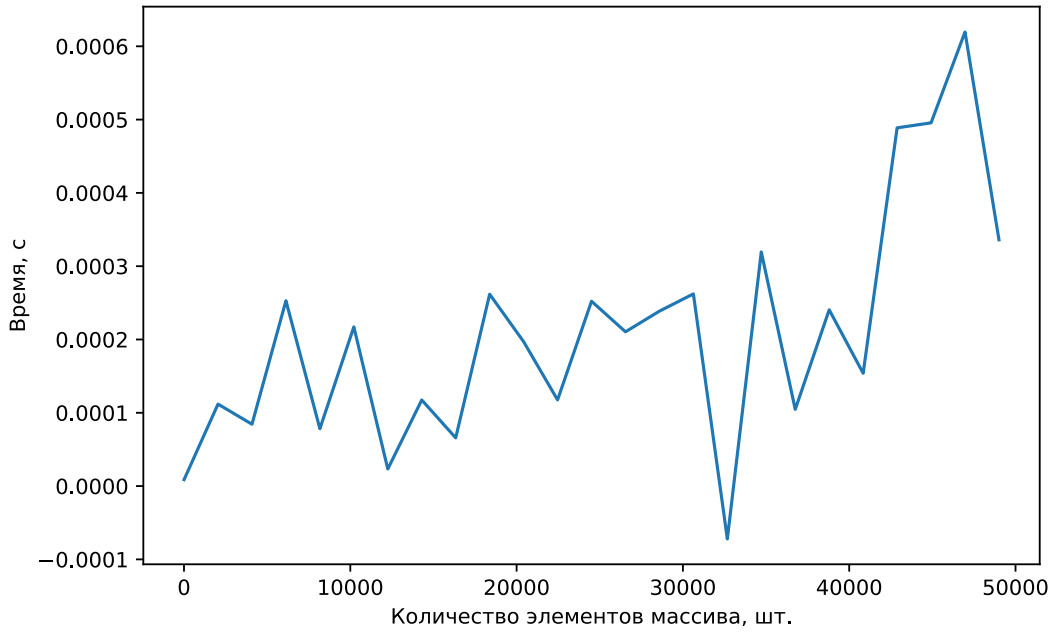

Рисунок 4 — Зависимость времени выполнения алгоритма от размера набора входных данных

2.3 Метод с предварительной сортировкой

Во многих современных языках программирования (таких как Java, Python, C++ и т. д.) используются встроенные функции, реализующие, по сути, один и тот же подход к сортировке — сортировка «Timsort» (названа в честь Тима Петерса [7]).

Timsort — гибридный алгоритм сортировки, сочетающий сортировку вставками и сортировку слиянием: по специальному алгоритму входной массив разделяется на подмассивы, затем каждый из подмассивов сортируется сортировкой вставками и собираются в единый массив сортировкой слиянием.

Алгоритм разбиения исходного массива на подмассивы накладывает определённые ограничения как на сортируемый массив, так и на подмассивы. Во-первых, подмассив должен быть упорядочен либо строго по убыванию, либо нестрого по возрастанию. Во-вторых, размер подмассива не должен быть слишком большим, так как к подмассиву размера будет в дальнейшем

применена сортировка вставками, а она эффективна только на небольших массивах. Оптимальная величина — это степень числа 2 (или близким к нему). Это требование обусловлено тем, что алгоритм слияния подмассивов наиболее эффективно работает на подмассивах примерно равного размера [8].

Для изучаемого метода пространственная сложность — $O(1)$, вычислительная — $O(n * \log n)$. Схема и график алгоритма Тима Петерса представлены соответственно на рисунках 5 и 6.

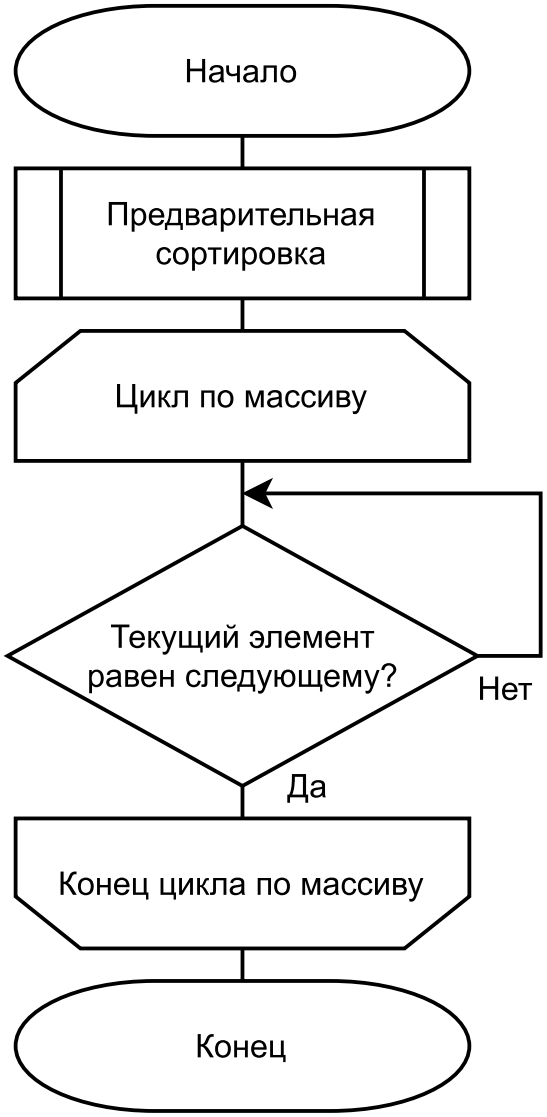

Рисунок 5 — Схема алгоритма

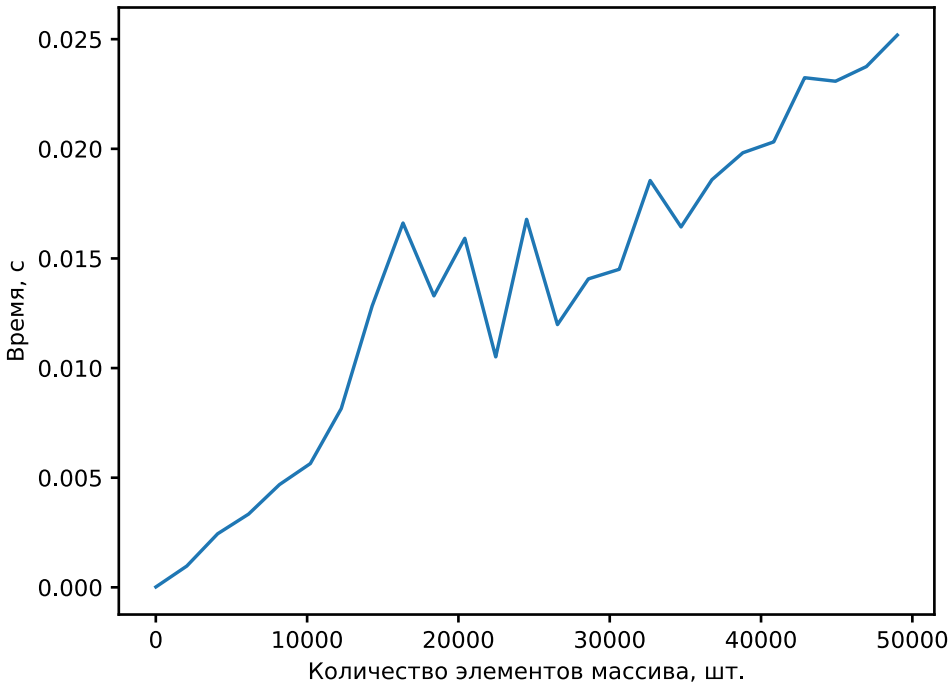

Рисунок 6 — Зависимость времени выполнения алгоритма от размера набора входных данных

Недостатком этого алгоритма поиска дубликата является использование встроенной функции, реализация которой может разниться от языка к языку.

2.4 Метод подсчёта суммы элементов

Этот метод не требует знаний какого-либо языка программирования (что бесспорно является одним из его плюсов). Его схема изображена на рисунке 7.

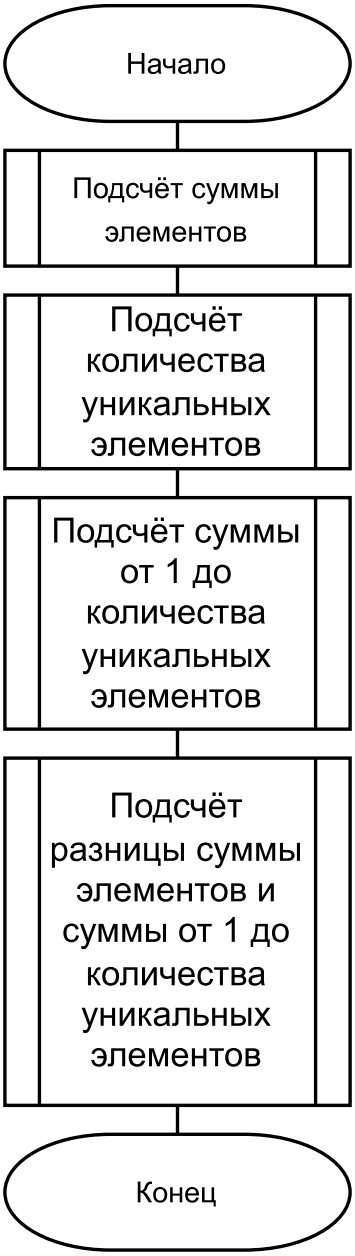

Рисунок 7 — Схема алгоритма

В этом примере можно добиться результата временной сложности $O(n)$ (показано на рисунке 8) [4] и пространственной — $O(1)$.

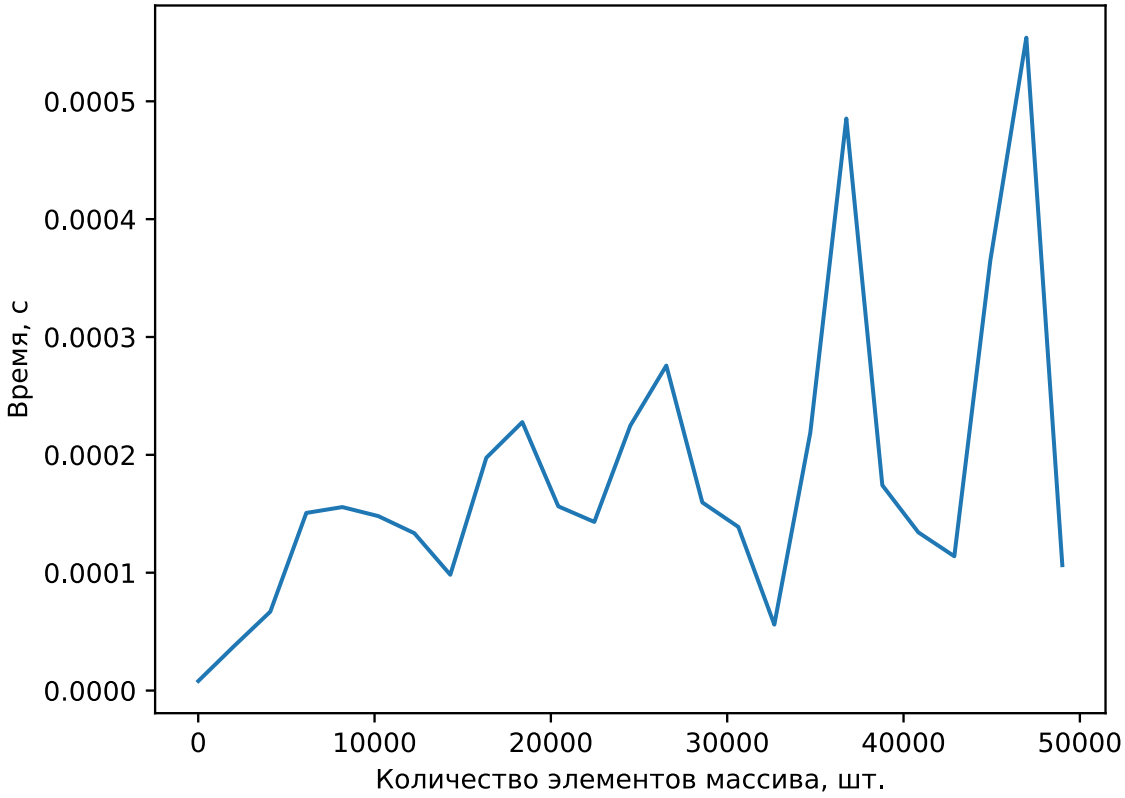

Рисунок 8 — Зависимость времени выполнения алгоритма от размера набора входных данных

Тем не менее, это решение работает только в случае, когда имеется один дубликат в наборе данных, и это является серьёзным недостатком и ставит под сомнение его использование вне учебных целей. Важной оговоркой к методу подсчёта суммы элементов является тот факт, что для того, чтобы подсчитать количество уникальных элементов, нужно их сначала найти. Эта рекурсия препятствует нормальной реализации алгоритма [9].

2.5 Метод маркера

Если все предыдущие методы поиска дубликата в наборе данных предполагали, что элементы имели собственный индекс (как в обычном массиве), то метод маркера оперирует тем, что входные данные — это многосвязный список, каждый элемент которого ссылается на следующий,

который в свою очередь имеет индекс, равный значению предыдущего, и так далее (пример на рисунке 9). При проходе через каждый элемент помечается соответствующий индекс, прибавляя к нему знак минус. Элемент является дубликатом, если его индекс уже помечен минусом [10].

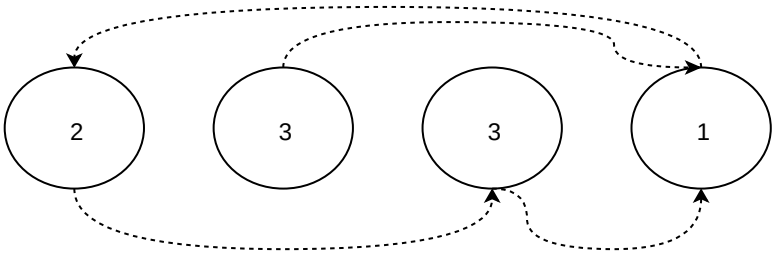

Рисунок 9 — Пример многосвязного списка

Таким образом, можно искать несколько дубликатов одного и того же значения элемента или несколько дубликатов разных значений.

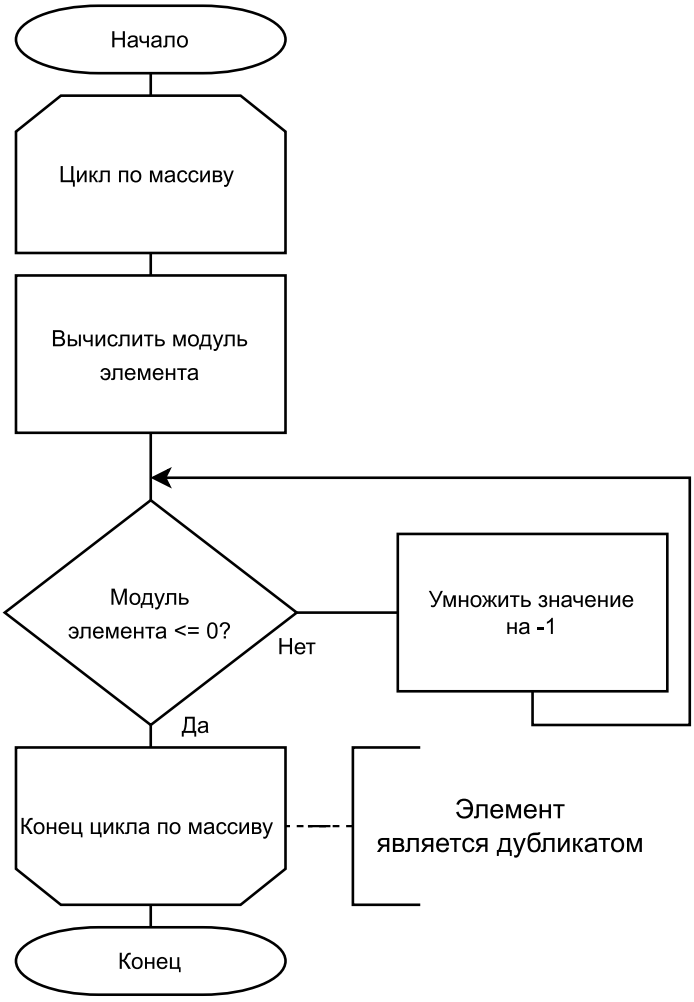

Рисунок 10 — Схема работы алгоритма

Временная сложность этого алгоритма — $O(n)$, а пространственная — $O(1)$ (график на рисунке 11) [7].

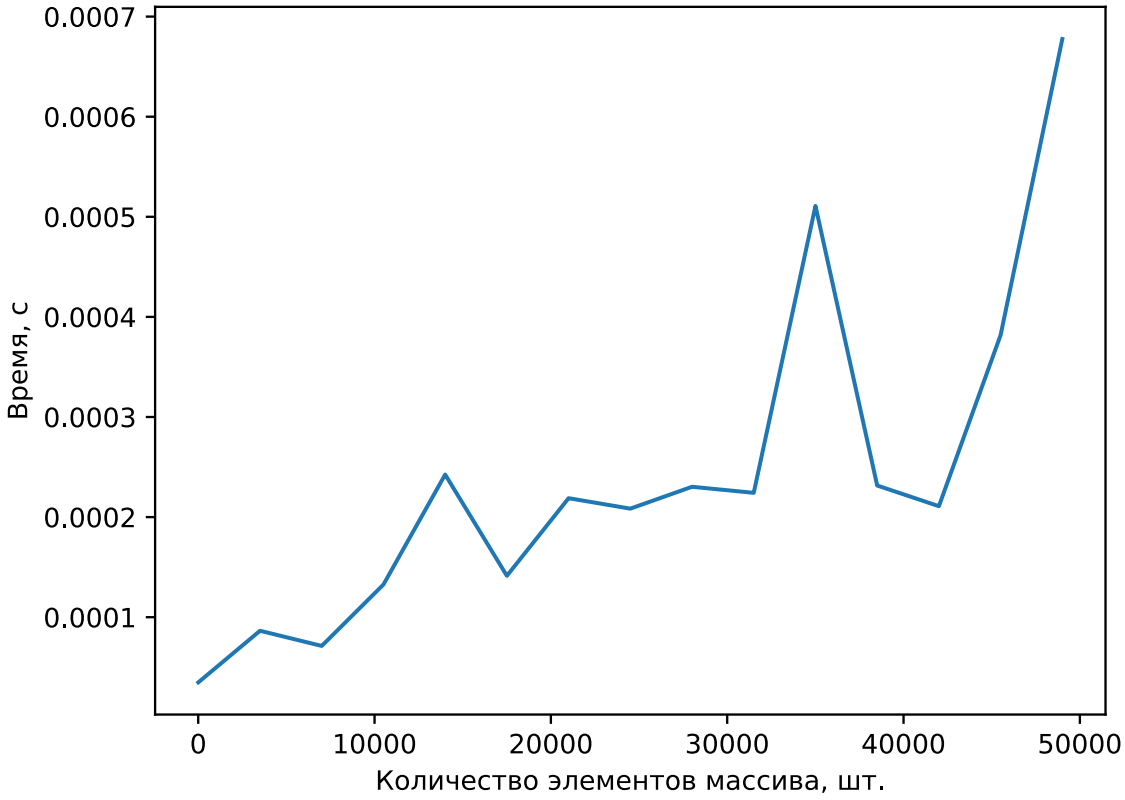

Рисунок 11 — Зависимость времени выполнения алгоритма от размера набора входных данных

Одним из недостатков этого алгоритма являются возможность работать только на данных целочисленного типа. Ещё одной его особенностью является то, что метод маркера изменяет входной набор данных (в частности, помечает элементы знаком минуса). Последнее можно исправить, скопировав входной массив, однако это существенно замедлит алгоритм и увеличит потребляемую им память [11].

2.6 Метод бегуна

Метод бегуна также предлагает рассматривать массив как список связей между элементами (что возможно достичь ограничением диапазона значений элементов) [11].

Пусть дан список из элементов «1», «2», «3», «4», «2» (изображён на рисунке 12). Очевидно, что дубликат существует, когда в списке есть циклическая связь. Более того, дубликат проявляется на точке входа цикла (в этом случае, второй элемент) [12].

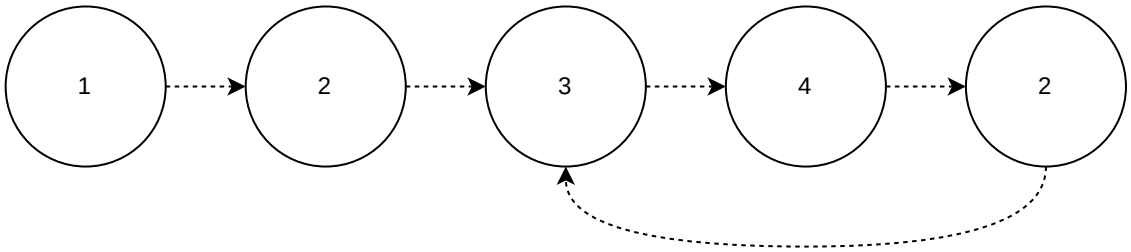

Рисунок 12 — Пример многосвязного списка

Удобно взять за основу алгоритм нахождения цикла по Флойду [13] (схема на рисунке 13).

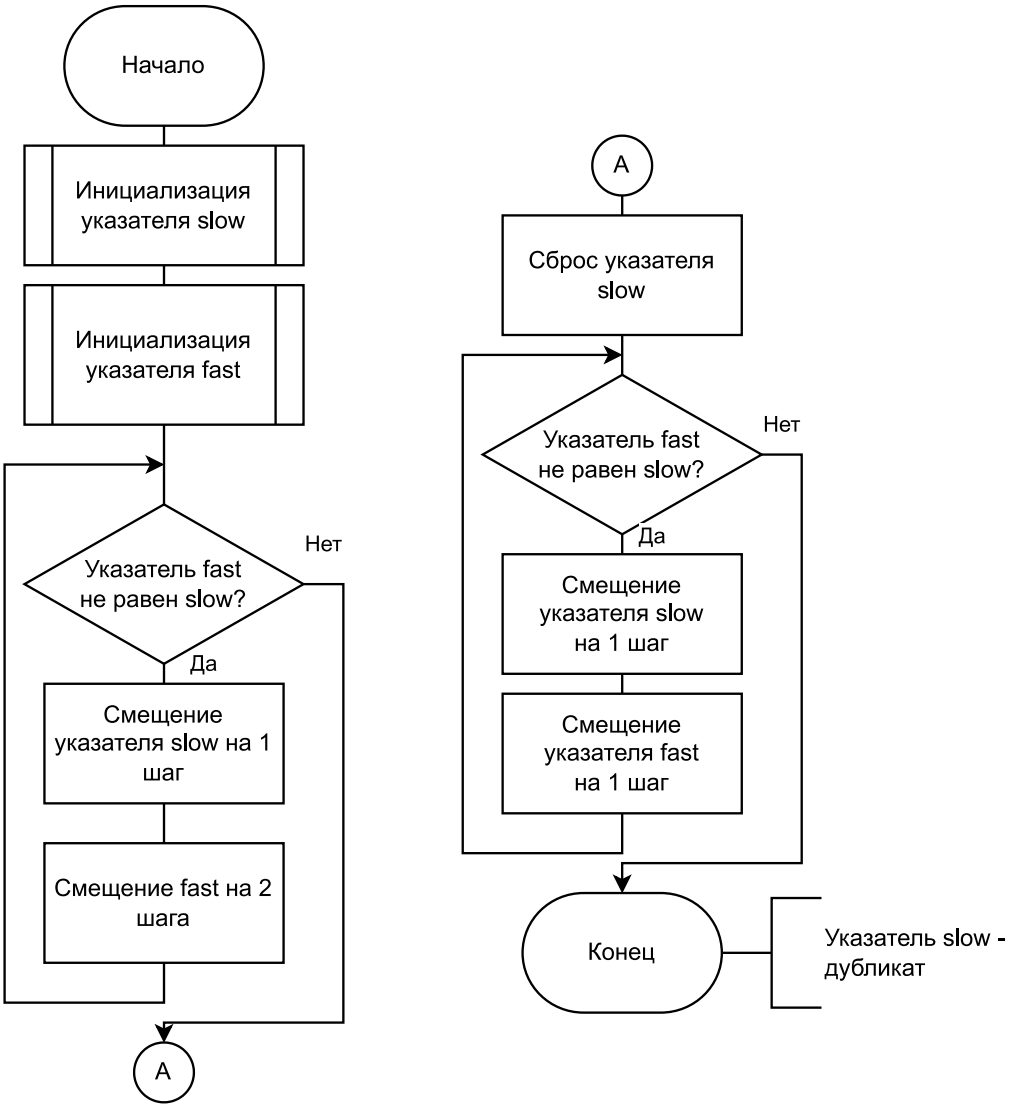

Рисунок 13 — Схема работы алгоритма

Это решение даёт результат временной сложности $O(n)$ и пространственной $O(1)$; не требует изменения входящего списка (график на рисунке 14) [7].

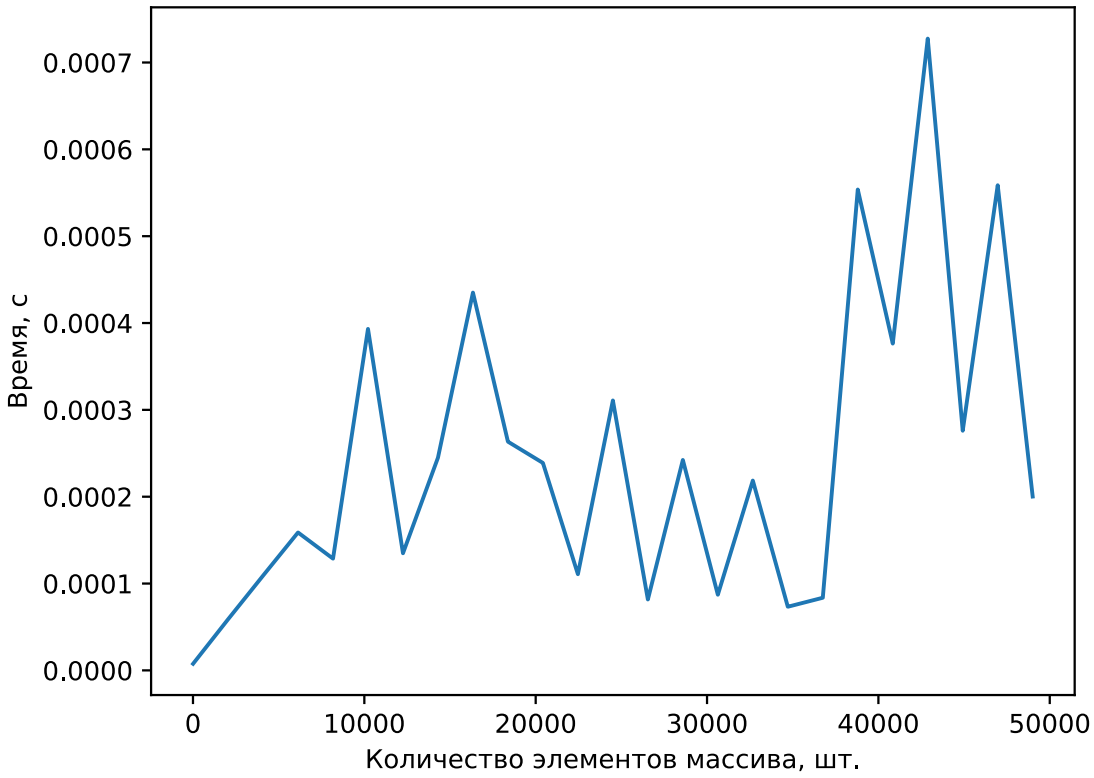

Рисунок 14 — Зависимость времени выполнения алгоритма от размера набора входных данных

3 Применимость алгоритмов поиска дубликатов на практике

Из рассмотренных методов не все являются достаточно оптимальными по времени и (или) по памяти, а значит не каждый из них можно применить на практике с уверенностью, что вычислительные ресурсы не будут использоваться в чрезмерном количестве [15].

Пространственные и временные характеристики рассмотренных выше алгоритмов представлены в таблице 2.

Таблица 2 — Сравнение алгоритмов поиска

Алгоритм	Пространственная сложность	Временная сложность
Brute Force	$O(1)$	$O(n^2)$
Метод подсчёта итераций	$O(n)$	$O(n)$
Метод с предварительной сортировкой	$O(1)$	$O(n * \log n)$
Метод подсчёта суммы элементов	$O(1)$	$O(n)$
Метод маркера	$O(1)$	$O(n)$
Метод бегуна	$O(1)$	$O(n)$

Если же пространственная сложность почти у всех алгоритмов одинаковая — $O(1)$, то по времени 4 из 6 алгоритмов имеет линейное время, у остальных либо квадратичное, либо линейно-логарифмическое.

Отсюда можно сделать промежуточный вывод — алгоритмы подсчёта суммы элементов, маркера и бегуна имеют довольно неплохие характеристики сложности по сравнению с другими алгоритмами.

Однако метод подсчёта суммы элементов работает только в случае одного дубликата, а метод маркера в процессе работы изменяет входной набор данных (в частности, список). Поэтому оптимальным алгоритмом поиска дубликатов в наборе данных является алгоритм бегуна.

ЗАКЛЮЧЕНИЕ

Проведён анализ предметной области. Рассмотрена проблема дубликации данных. Описаны существующие алгоритмы поиска дубликатов. Проведён всесторонний анализ таких алгоритмов. Определён оптимальный алгоритм среди представленных — алгоритм бегуна. Он обладает $O(1)$ пространственной сложностью и $O(n)$ временной сложностью, не изменяет входные данные. Более того, с помощью этого алгоритма можно найти сразу несколько дубликатов.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Маслова М.А. Алгоритм поиска и устранения дубликатов объектов в информационных системах обработки персональных данных / М.А. Маслова // Перспективы развития информационных технологий. — 2016. — №29. — С. 204–211.

2. Чуканов К.В. Целостность баз данных / Г.Я. Чичикин, К.В. Чуканов // Наука, техника и образование. — 2018. — №11 (52).

3. Maheshwari A. Introduction to Theory of Computation / M. Smid, A. Maheshwari // School of Computer Science. Carleton University. — 2019. — 45–46 p.

4. Introduction to Algorithms / T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein. — Boston: MIT Press and McGraw-Hill. 2017.

5. McIlroy P. Optimistic Sorting and Information Theoretic Complexity / P. McIlroy. // Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms. — Austin, 2015.

6. Борзяк А.А. Способ конвейерной обработки данных / Л.Ю. Исмаилова, А.А. Борзяк // ИТНОУ: информационные технологии в науке, образовании и управлении. — 2018. — №3 (7).

7. Aditya Y.B. Grokking Algorithms. An illustrated guide for programmers and other curious people. / Y.B. Aditya — London: Manning, 2017.

8. Allender E. Improved lower bounds for the cycle detection problem / M.M Klawe, E. Allender — Atlanta: Elsevier, 1985.

9. Гардашова З.Ш. Методика применения программирования поиск элемента в упорядоченном массиве / Г.Н. Тагиев, З.Ш. Гардашова // Colloquium-journal. — 2019. — №2–1 (26).

10. Айткулов П.Г. Обработка символьных массивов / П.Г. Айткулов // УБС. — 2010. — №28.

11. Левкин А.С. Алгоритм поиска нечетких дубликатов текста / В.О. Хилько, А.С. Левкин // Актуальные проблемы инфотелекоммуникаций в науке и

образовании: сборник научных статей: в 4-х томах. — СПб.: 2017. — С. 314–317.

12. Sedgewick R. Algorithms / K. Wayne, R. Sedgewick. — London: Addison Wesley, 2018.

13. Амиргалиев Е. Поиск дубликатов в тексте с использованием алгоритма Шингла / С. Бегадил, Е. Амиргалиев // Университет Сулейман Демиреля. — 2018. — С. 203.

14. Экспериментальное исследование метода идентификации массивов бинарных данных / Е.В. Лебеденко, В.В. Рябоконь, А.Н. Лапко, М.А. Куцакин. // Известия ТулГУ. Технические науки. — 2017. — №5.

15. Берсенева Е.А. Автоматизированная система лексического анализа: основные методы и подходы / А.А. Седов, Е.А. Берсенева // Вестник современной клинической медицины. — 2017. — №2.