**Отчет по лабораторной работе 1**

Выполнила: Щавровская П.К., АТ-05

Преподаватель: Спиричева Н.Р.

Оглавление

Введение

I. Алгоритм сортировки пузырьком

II. Алгоритм быстрой сортировки

III. Алгоритм сортировки с помощью дерева

IV. Алгоритм сортировки вставками

V. Алгоритм сортировки слиянием

VI. Алгоритм сортировки красно-черным деревом

VII. Алгоритм поразрядной сортировки

VII. Алгоритм пирамидальной сортировки

Общий вывод

**Введение**

В данной работе рассматриваются различные алгоритмы сортировки с целью сопоставления их программной сложности с теоретической. Для упрощения оценки сложности алгоритмов был разработан отдельный класс, который генерирует случайные массивы, а также массивы, отсортированные по возрастанию и убыванию.

Настройки для проведения сортировок указаны в файле config.xml. Эти настройки содержат название алгоритма, длину массивов и количество повторений для каждой длины массива. Для каждой сортировки выполняются три типа тестов: лучший случай (массив, отсортированный по возрастанию), худший случай (массив, отсортированный по убыванию) и несколько случаев со случайными данными.

**I Алгоритм сортировки пузырьком**

**Bubble Sort**

Сравнивает соседние элементы и меняет их местами, если это необходимо. Алгоритм выполняется следующим образом:

1. Начинаем с первого элемента списка и сравниваем его со следующим.

2. Если первый элемент больше второго, меняем их местами.

3. Продолжаем этот процесс до конца списка.

4. После завершения первого прохода самый большой элемент окажется в конце списка.

5. Затем начинаем второй проход, исключая последний элемент.

6. Повторяем эти шаги, пока весь список не будет отсортирован.

Сложность алгоритма:

• В лучшем случае: O(n)

• В среднем и худшем случаях: O(n²)

Программная сложность:

Посмотрим на график (Рис. 1), который мы создали на основе работы данного алгоритма.

Изображение выглядит как текст, диаграмма, линия, График

Контент, сгенерированный ИИ, может содержать ошибки.

Рис. 1 Результаты сортировки пузырьком

Основываясь на график, можно сказать, что в среднем сортировка пузырьком имеет сложность O(n2), т.е. соответствует квадратичному росту. Лучший и худший случай значительно отличаются от среднего.

**Вывод:**

На основе программного теста, можно утверждать, что реальная и теоретическая сложности практически совпадают.

**II Алгоритм быстрой сортировки**

**Quick Sort**

Алгоритм быстрой сортировки:

1. Выбор опорного элемента: Обычно выбирается один элемент массива (например, последний, первый или средний).

2. Разделение массива:

• Устанавливаются два указателя: i (начало массива) и j (конец массива).

• Указатель i двигается вправо, пока не найдет элемент, больший или равный опорному.

• Указатель j двигается влево, пока не найдет элемент, меньший или равный опорному.

• Если i меньше j, элементы на позициях i и j меняются местами.

3. Разделение на подмассивы: Когда указатели пересекаются, массив делится на три части:

• Элементы меньше опорного.

• Сам опорный элемент.

• Элементы больше опорного.

4. Рекурсивная сортировка: Рекурсивно применяем тот же процесс к подмассивам слева и справа от опорного элемента.

5. Сборка результата: В конце все части объединяются в один отсортированный массив.

Сложность алгоритма:

• Лучший случай: O(n log n) — когда массив уже отсортирован или почти отсортирован.

• Средний случай: O(n log n) — для случайных массивов.

• Худший случай: O(n²) — когда массив отсортирован в обратном порядке или все элементы одинаковы (в этом случае выбор опорного элемента неэффективен).

Программная сложность:

Быстрая сортировка часто реализуется с использованием рекурсии, что может привести к большому потреблению памяти в случае глубоких рекурсий. Однако при правильной реализации (например, с использованием хвостовой рекурсии или оптимизации выбора опорного элемента) можно достичь хорошей производительности.

Рассмотрим график (Рис. 2), который мы построили на основе работы алгоритма:

Изображение выглядит как текст, диаграмма, линия, График

Контент, сгенерированный ИИ, может содержать ошибки.

Рис. 2 Результаты быстрой сортировки

**Вывод:**

После тестов, получив статистические данные, имеем сложность для быстрой сортировки схожую с теоретической для всех случаев.

**III Алгоритм сортировки с помощью дерева**

**Tree Sort**

Алгоритм сортировки с использованием бинарного дерева поиска:

1. Создание пустого двоичного дерева поиска.

2. Добавление первого элемента: Помещаем его в корень дерева.

3. Добавление последующих элементов:

• Начинаем с корня дерева.

• Если элемент меньше текущего узла, идём в левое поддерево.

• Если элемент больше или равен текущему узлу, идём в правое поддерево.

• Продолжаем движение до тех пор, пока не найдем пустое место для нового узла.

4. Обход дерева:

• После добавления всех элементов производим симметричный (инфиксный) обход дерева: сначала левое поддерево, затем корень, затем правое поддерево.

• В результате этого обхода элементы будут отсортированы.

5. Получение результата: Порядок элементов, полученный при обходе, является отсортированным массивом.

Сложность алгоритма:

• Средний случай: O(n log n) — если дерево сбалансировано, вставка и поиск занимают O(log n).

• Худший случай: O(n²) — если элементы вставляются в уже отсортированном порядке (например, по возрастанию), дерево вырождается в связный список.

• Лучший случай: O(n log n) — если дерево сбалансировано.

Программная сложность:

Рассмотрим график (Рис. 3), который мы построили на основе работы алгоритма:

Изображение выглядит как текст, диаграмма, линия, График

Контент, сгенерированный ИИ, может содержать ошибки.

Рис. 3 Алгоритм сортировки с помощью дерева

**Вывод:**

Алгоритм сортировки с помощью деревья на практике и на теории имеет схожие сложности во всех случаях.

**IV Алгоритм сортировки вставками**

**Insertion Sort**

Алгоритм:

1. Начало с первого элемента:

• Первый элемент массива считается уже отсортированным. Это начало "отсортированной" части массива.

2. Обработка второго элемента:

• Берём второй элемент и сравниваем его с элементами перед ним.

• Если он меньше, перемещаем все большие элементы вправо, чтобы освободить место для вставки.

• Вставляем элемент на найденную позицию.

3. Повторение для всех последующих элементов:

• Для каждого следующего элемента (начиная с третьего) повторяем процесс: сравниваем с элементами в отсортированной части, сдвигаем элементы большего размера вправо и вставляем текущий элемент на правильное место.

4. Продолжение процесса:

• Продолжаем до тех пор, пока весь массив не будет отсортирован.

5. Завершение сортировки:

• Сортировка завершена, когда все элементы обработаны и массив полностью отсортирован.

Сложность алгоритма:

• Лучший случай: O(n) — если массив уже отсортирован, то каждый новый элемент просто сравнивается с последним элементом отсортированной части.

• Средний случай: O(n²) — при случайных данных, так как в среднем потребуется сдвигать половину элементов для каждой вставки.

• Худший случай: O(n²) — если массив отсортирован в обратном порядке, каждый новый элемент будет вставляться в начало отсортированной части.**Программная сложность:**

Изображение выглядит как текст, диаграмма, линия, График

Контент, сгенерированный ИИ, может содержать ошибки.Рассмотрим график (Рис. 4), который мы построили на основе работы алгоритма:

Рис. 4 Алгоритм сортировки вставкой

**Вывод:**

После программных тестов видно, что сложности практическая и теоретическая совпадают.

**V Алгоритм сортировки слиянием**

**Merge Sort**

Алгоритм сортировки слиянием:

1. Начальное состояние:

• Берем исходный массив элементов.

• Если длина массива меньше или равна 1, он уже отсортирован.

2. Процесс разделения:

• Делим массив пополам на две равные части.

• Для каждой половины массива рекурсивно применяем ту же процедуру.

• Продолжаем деление, пока не получим подмассивы длиной 1.

3. Процесс слияния:

• Берем две соседние отсортированные части массива.

• Сравниваем элементы этих частей.

• Берем меньший элемент и помещаем его в результат.

• Повторяем процесс, пока не объединим все элементы.

4. Повторение процесса:

• Для каждой пары соседних частей выполняем слияние.

• Полученные большие отсортированные части снова объединяем.

• Продолжаем, пока весь массив не будет отсортирован.

5. Финальный результат:

• Получаем полностью отсортированный массив.

• Все элементы расположены в порядке возрастания.

Сложность алгоритма:

• Лучший случай: O(n log n) — даже если массив уже отсортирован, алгоритм все равно будет делить его и сливать.

• Средний случай: O(n log n) — при случайных данных.

• Худший случай: O(n log n) — даже в самом неблагоприятном случае алгоритм будет работать с такой же сложностью.

**Программная сложность:**

Изображение выглядит как текст, диаграмма, снимок экрана, линия

Контент, сгенерированный ИИ, может содержать ошибки.Рассмотрим график (Рис. 5), который мы построили на основе работы алгоритма:

Рис. 5 Алгоритм сортировки слиянием

**Введение:**

В результате тестов можно сделать вывод, что практическая и теоретическая сложности совпадают.

**VI Алгоритм сортировки красно-черным деревом**

**Red-Black Tree Sort**

Алгоритм сортировки с использованием красно-черного дерева

1. Начальное состояние:

• Создаем пустое красно-черное дерево.

• Первый добавленный элемент становится корнем и окрашивается в черный цвет.

2. Процесс добавления элемента:

• Новый элемент добавляется как лист дерева.

• Изначально новый узел окрашивается в красный цвет.

• Если родительский узел черный, добавление завершено.

• Если родительский узел красный, требуется балансировка.

3. Балансировка дерева:

• Проверяем, нарушены ли свойства красно-черного дерева.

• Выполняем повороты узлов для восстановления баланса:

• Левый поворот: используется, когда правый ребенок красного узла становится родителем.

• Правый поворот: используется, когда левый ребенок красного узла становится родителем.

• Меняем цвета узлов согласно правилам.

• Обеспечиваем, чтобы все пути от узла до листьев содержали одинаковое количество черных узлов.

4. Поддержание свойств:

• Корень всегда черный.

• Все листья (NULL-узлы) черные.

• Красный узел не может иметь красных детей (т.е., два красных узла не могут стоять друг под другом).

• Любой путь от узла до листа содержит одинаковое количество черных узлов.

5. Получение результата:

• После добавления всех элементов дерево автоматически сбалансировано.

• Обход дерева симметричным способом (ин-ордер) дает отсортированный массив.

Сложность алгоритма:

• Лучший случай: O(n log n)

• Средний случай: O(n log n)

• Худший случай: O(n log n)

За счет само балансировки красно-черного дерева поддерживается глубина O(log n), что обеспечивает O(log n) для операций вставки и O(n) для обхода.

**Программная сложность:**

Рассмотрим график (Рис. 6), который мы построили на основе работы алгоритма:

Изображение выглядит как текст, диаграмма, линия, График

Контент, сгенерированный ИИ, может содержать ошибки.

Рис. 6 Алгоритм сортировки красно-черным деревом

**Вывод:**

По графику видно, что практическая и теоретические сложности схожи, но данный алгоритм стабильнее, чем сортировка бинарным деревом.

**VII Алгоритм Поразрядной сортировки**

Алгоритм поразрядной сортировки

1. Определение максимального числа:

• Находим максимальное число в массиве, чтобы определить количество разрядов.

2. Сортировка по каждому разряду:

• Для каждого разряда (от младшего к старшему) выполняем сортировку с помощью стабильной сортировки (например, сортировка подсчетом).

3. Стабильная сортировка:

• Сортируем элементы по текущему разряду, используя вспомогательный массив, чтобы сохранить порядок элементов с одинаковыми значениями.

4. Повторяем:

• Повторяем процесс для всех разрядов.

Сложность алгоритма

• Лучший случай: O(nk)

• Средний случай: O(nk)

• Худший случай: O(nk)

где n — количество элементов, а k — количество разрядов в максимальном числе.

**Программная сложность:**

Изображение выглядит как текст, диаграмма, линия, График

Контент, сгенерированный ИИ, может содержать ошибки.Рассмотрим график (Рис. 7), который мы построили на основе работы алгоритма:

Рис. 7 Алгоритм поразрядной сортировки

**Вывод:**

По графику видно, что практическая и теоретические сложности схожи.

**VIII Алгоритм Поразрядной сортировки**

Алгоритм:

1. Построение кучи:

• Преобразуем массив в бинарную кучу. Это можно сделать за O(n) времени.

2. Сортировка:

• Извлекаем максимальный элемент (для максимальной кучи) из кучи и помещаем его в конец массива.

• Уменьшаем размер кучи на 1 и восстанавливаем свойства кучи для оставшихся элементов.

• Повторяем процесс, пока не отсортируем весь массив.

Сложность алгоритма

• Лучший случай: O(n log n)

• Средний случай: O(n log n)

• Худший случай: O(n log n)**Программная сложность:**

Рассмотрим график (Рис. 8), который мы построили на основе работы алгоритма:

Изображение выглядит как текст, диаграмма, снимок экрана, линия

Контент, сгенерированный ИИ, может содержать ошибки.Рис. 8 Алгоритм пирамидальной сортировки

**Вывод:**

По графику видно, что практическая и теоретические сложности схожи

**Общий вывод**

• Поразрядная сортировка (Radix Sort). Поразрядная сортировка эффективна для сортировки целых чисел и строк фиксированной длины. В отличие от многих других алгоритмов, она не использует сравнения для определения порядка элементов. Вместо этого она обрабатывает элементы по разрядам, что позволяет ей достигать линейной временной сложности O(n) для определённых наборов данных.

• Пирамидальная сортировка (Heap Sort). Пирамидальная сортировка эффективна для сортировки больших массивов и использует структуру данных кучи для достижения своей цели. Она гарантирует время выполнения O(n log n) в худшем случае, что делает её надежным выбором.

• Сортировка пузырьком: Этот простой алгоритм имеет сложность O(n²) и не рекомендуется для использования на больших массивах из-за своей низкой эффективности. Он подходит только для учебных целей или очень небольших наборов данных.

• Быстрая сортировка (Quick Sort): Один из самых быстрых алгоритмов для обычных массивов, он имеет среднее время выполнения O(n log n) и работает на месте. Однако его производительность может ухудшиться для почти отсортированных массивов, если не использовать стратегии выбора опорного элемента.

• Сортировка на основе деревьев (TreeSort): Этот алгоритм использует бинарное дерево для сортировки. Он хорошо подходит для обработки данных в режиме потока, но требует дополнительной памяти и может страдать от ухудшения производительности при несбалансированных деревьях.

• Сортировка вставками (Insertion Sort): Эффективна для небольших массивов или почти отсортированных данных, но её эффективность значительно снижается для больших и сильно перемешанных массивов, так как имеет сложность O(n²).

• Сортировка слиянием (Merge Sort): Эффективно работает с большими массивами и гарантирует O(n log n) в любом случае, но требует дополнительной памяти для временных массивов, что может быть проблемой при работе с ограниченной памятью.

• Сортировка с использованием красно-черного дерева (Red-Black Tree Sort): Этот алгоритм подходит для случаев, когда данные часто изменяются или необходимо поддерживать отсортированный список. Однако его производительность может быть ниже, чем у быстрой сортировки или сортировки слиянием из-за постоянных операций по балансировке.

Каждый из этих алгоритмов имеет свои уникальные характеристики, и выбор подходящего метода зависит от конкретных условий задачи, таких как размер данных, необходимость в стабильности, доступная память и особенности структуры данных.