ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчёт по лабораторной работе № 5

«Алгоритмы сортировки»

Выполнил работу

Московкин Александр Николаевич

Академическая группа № J3111

Принято

Вершинин Владислав Константинович

Санкт-Петербург

2024

**Структура отчёта:**

1. Введение

Цель данной лабораторной работы — сравнить три различных алгоритма сортировки: сортировка расчёской (*comb sort*), пирамидальная сортировка (*pyramid heap sort*) и сортировка с помощью двоичного дерева (*tree sort*). Для этого необходимо протестировать каждый алгоритм на массиве данных и измерить время выполнения сортировки.

Для достижения указанной цели лабораторной работы необходимо решить несколько задач:

1. Изучить каждый из алгоритмов сортировки, их принципы работы и применимость в зависимости от характеристик данных.
2. Реализовать функции сортировки на языке C++ для каждого из выбранных алгоритмов.
3. Провести тестирование разработанных алгоритмов на массивах разных размеров (от 1000 до 10000 элементов) и различными распределениями значений.
4. Записать время выполнения каждого алгоритма в файлы, чтобы можно было проанализировать производительность.
5. Построить графические зависимости времени выполнения алгоритмов от длины массива, чтобы визуально оценить их эффективность.
6. Проанализировать временные и пространственные характеристики каждого алгоритма, проведя сравнение их сложности.
7. Проанализировать соответствие теоретической асимптотики с практическими результатами, выяснить наличие выбросов в данных и их причины, а также определить, в каких случаях целесообразно использовать каждый из рассматриваемых алгоритмов.
8. Теоретическая подготовка

2.1. Основные понятия и определения

**Сортировка** — процесс упорядочивания элементов в массиве или векторе по определённому критерию. Существуют различные алгоритмы сортировки, каждый из которых имеет свои преимущества и недостатки в зависимости от типа данных и их распределения.

2.2. Сортировка расчёской (*Comb sort*)

**Comb sort** — это довольно упрощённый алгоритм сортировки, Сортировка расчёской улучшает сортировку пузырьком, и конкурирует с алгоритмами, подобными быстрой сортировке. Основная идея — устранить черепах, или маленькие значения в конце списка, которые крайне замедляют сортировку пузырьком.

2.3. Бисерная сортировка (*Bead sort*)

**Bead Sort** — этот алгоритм, также известный как сортировка по гравитации, был вдохновлен природными явлениями и был разработан с учетом падения объектов (или бусин) под действием гравитации.

Идея: Положительные числа представлены набором бусин, как на счетах.

2.4. Сортировка с помощью двоичного дерева (*tree sort*)

**Tree Sort** — универсальный алгоритм сортировки, заключающийся в построении двоичного дерева поиска по ключам массива (списка), с последующей сборкой результирующего массива путём обхода узлов построенного дерева в необходимом порядке следования ключей.

2.5. Временная и пространственная сложность

Comb Sort:

* Временная сложность:
* Худший случай: *O(n²)* — Происходит в сильно несортированных массивах с неудачной последовательностью шагов.
* Лучший случай: *O(n\*logn)* — Это достигается, если массив уже почти отсортирован, так как большая часть работы выполняется на последних итерациях с маленькими значениями *gap*, что схоже с сортировкой вставками.
* Пространственная сложность: *O(1)* — Алгоритм не использует дополнительной памяти, кроме нескольких переменных для хранения текущего *gap* и индексов.

Bead Sort:

* Временная сложность:
* Худший случай: *O(S)* — где *S* — сумма всех значений массива. Так как алгоритм использует "гравитационную" модель, он зависит от того, сколько "бусинок" нужно обработать. Если значения в массиве велики, S может быть значительно больше n.
* Лучший случай: *O(n)*, где *n* — количество элементов. Это достигается, если массив уже отсортирован.
* Пространственная сложность: *O(S)*, где *S* — сумма всех значений элементов массива. Для представления каждой "бусинки" требуется память, поэтому сложность масштабируется с увеличением максимального значения в массиве.

Tree Sort:

* Временная сложность:
* Худший случай: *O(n²)* — Если элементы поступают в отсортированном порядке (или в обратном порядке), дерево становится вырожденным (линейным списком). В этом случае каждая вставка занимает *O(n)*, и суммарное время для *n* элементов будет *O(n²)*.
* Лучший случай: *O(n\*logn)* — Если дерево сбалансировано (примерно одинаковое количество элементов слева и справа), то операции вставки и обхода дерева занимают *O(logn)* для каждой вставки. Всего *n* элементов, следовательно, общая сложность: *O(n\*logn)*.
* Пространственная сложность:
* Худший случай: Для обхода дерева используется стек вызовов. В худшем случае (вырожденное дерево) стек вызовов может потребовать *O(n)* памяти.
* Лучший случай: В лучшем случае (сбалансированное дерево) глубина стека вызовов будет *O(logn)*.

2.6. Понимание алгоритмов сортировки

Для успешного анализа и выбора алгоритма сортировки важно учитывать:

* Структуру данных и их характер (например, случайные, уже отсортированные, или сильно упорядоченные).
* Ожидаемый размер и диапазон значений данных, особенно для алгоритмов таких как Bead Sort, где отрицательные значения в массиве могут повлиять на результат.

1. Реализация

3.1. Выбор инструментов

Для реализации задач по сортировке были использованы стандартные библиотеки C++:

* *<vector>* для хранения массивов данных и результатов сортировки.
* *<algorithm>*для использования функций, если конкретно, то *fill*, который заполняет нам двумерный массив в *Bead Sort*.

3.2. Разработка алгоритма

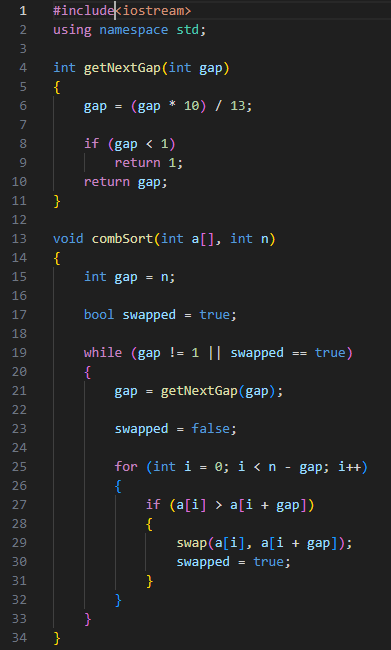
Процесс разработки алгоритмов состоял из нескольких ключевых этапов:

1. **Анализ задачи**: важно было понять, как каждый алгоритм работает и какие ситуации могут повлиять на их производительность. Были изучены основные принципы и временные характеристики каждого алгоритма.
2. **Определение структуры данных**: Для хранения промежуточных данных и считываемых значений были выбраны векторы (*vector*), которые позволяют динамически изменять размер и обеспечивают эффективное управление памятью.
3. **Написание основного кода**:

**3.2.1. Сортировка расчёской (Comb Sort)**

**Алгоритм**: Сортировка расчёской улучшает пузырьковую сортировку за счёт устранения "черепах" — небольших значений в конце массива, которые замедляют сортировку. Алгоритм использует переменную *gap*, которая на каждой итерации уменьшается до тех пор, пока не станет равной 1.

**Код**:



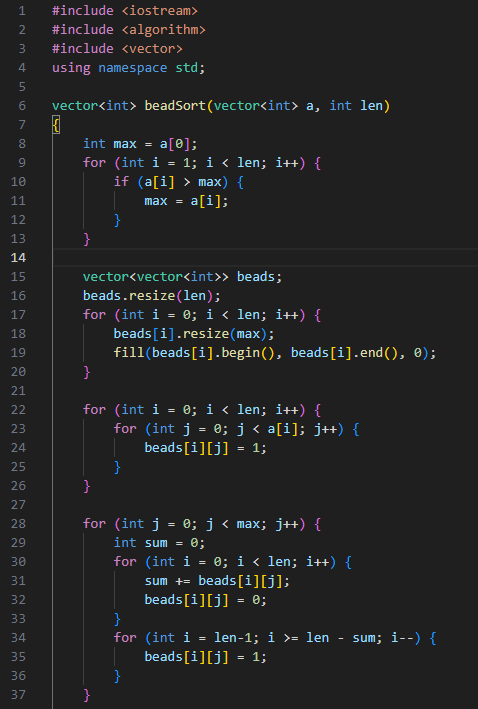
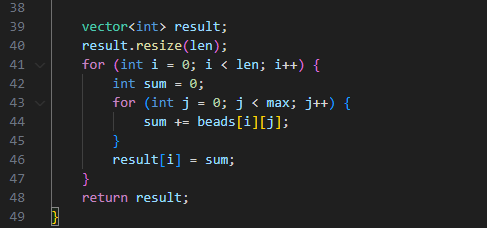
**Анализ**:

* **Временная сложность**: от *O(n²)* в худшем случае до *O(n\*logn)* в лучшем.
* **Пространственная сложность:** *O(1)*, так как не требуется дополнительная память.
* **Применение:** Подходит для массивов, близких к отсортированным, за счёт более быстрого устранения "черепах".

**3.2.2. Бисерная сортировка (Bead sort)**

**Алгоритм**: Бисерная сортировка моделирует "падение бусинок" под действием гравитации, представляя числа в виде вертикальных столбцов.

**Код**:

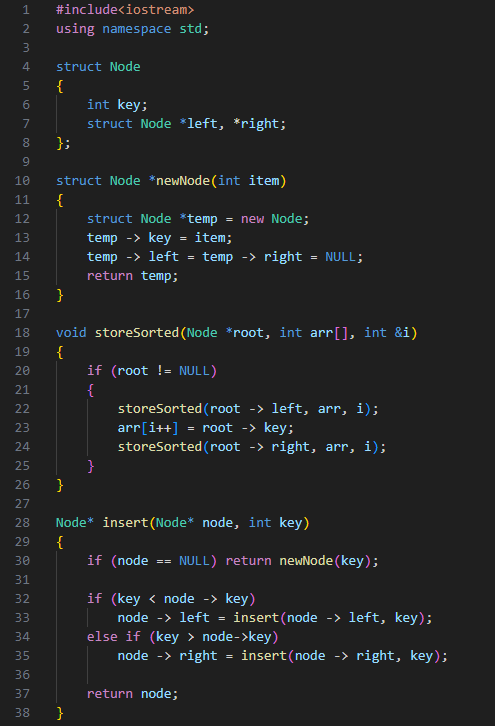
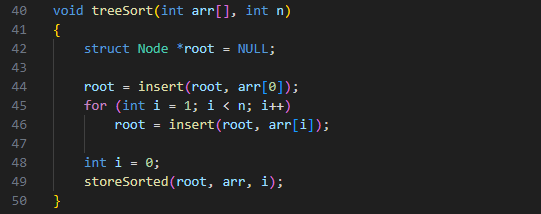
**Анализ**:

* **Временная сложность**: *O(n)* в лучшем случае и *O(S)* в худшем, где *S* — сумма значений элементов.
* **Пространственная сложность:** *O(S)*, из-за использования памяти для представления бусинок.
* **Применение:** Используется для небольших массивов с положительными целыми числами.

**3.2.3. Сортировка с помощью двоичного дерева (Tree Sort)**

**Алгоритм**: Строит бинарное дерево поиска из элементов массива, затем выполняет обход дерева, возвращая элементы в отсортированном порядке.

**Код**:

**Анализ**:

* **Временная сложность**: от *O(n\*logn)* в лучшем случае до *O(n²)* в худшем.
* **Пространственная сложность:** от *O(logn)* до *O(n)* из-за глубины дерева.
* **Применение:** Эффективен при использовании сбалансированных деревьев.

1. Экспериментальная часть

Для проверки производительности алгоритмов ***comb sort***, ***bead sort*** и ***tree sort*** были созданы тестовые массивы различной длины (1000, 5000, 10000 элементов) с разными распределениями значений (равномерное, случайное и отсортированное в обратном порядке). Время выполнения измерялось с использованием стандартной библиотеки C++.

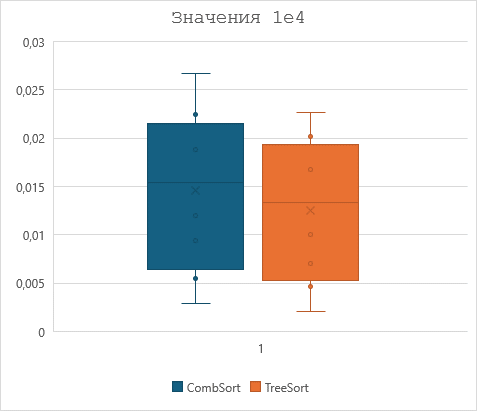
**Замеры для данных до 1е4**

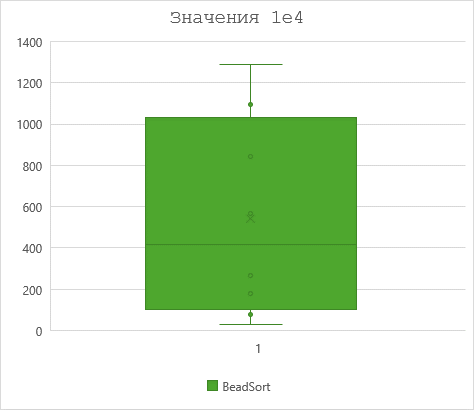
|  |  |  |  |
| --- | --- | --- | --- |
| Размер | Comb Sort | Bead Sort | Tree Sort |
| 1000 | 0.0001877 | 2.37046 | 0.00016 |
| 2000 | 0.0003991 | 5.94674 | 0.0003328 |
| 3000 | 0.0006734 | 9.01615 | 0.0005128 |
| 4000 | 0.0009239 | 12.4237 | 0.0006957 |
| 5000 | 0.0012248 | 15.6041 | 0.0009341 |
| 6000 | 0.0014747 | 18.3879 | 0.001119 |
| 7000 | 0.0016949 | 22.1968 | 0.0013309 |
| 8000 | 0.0020734 | 25.5299 | 0.0015577 |

*графики 1*

**Замеры для 1е4**

|  |  |  |  |
| --- | --- | --- | --- |
| Замер № | Comb Sort | Bead Sort | Tree Sort |
| 1 | 0.0029147 | 31.5035 | 0.0020456 |
| 2 | 0.0054287 | 77.6619 | 0.0046388 |
| 3 | 0.0094103 | 178.389 | 0.0070626 |
| 4 | 0.011975 | 265.126 | 0.0099711 |
| 5 | 0.0188336 | 567.2345 | 0.0170448 |
| 6 | 0.0187856 | 843.0835 | 0.0167287 |
| 7 | 0.0224573 | 1095.348 | 0.0201604 |
| 8 | 0.0267239 | 1287.402 | 0.0226205 |





*графики 2*

### **Анализ графиков**

**График 1** визуализирует время работы трех алгоритмов сортировки (**Comb Sort**, **Bead Sort** и **Tree Sort**) для массивов размером до 8000 элементов:

**Comb Sort:**

* График показывает, что время выполнения растет с увеличением размера массива, но рост более плавный, чем у алгоритмов с квадратичной сложностью. Это связано с улучшением пузырьковой сортировки за счет использования переменной *gap*.
* Алгоритм демонстрирует временную сложность на массивах, близких к отсортированным, и в худшем случае.
* *Comb Sort* оказался эффективным для массивов с равномерным или случайным распределением данных.

**Bead Sort:**

* Для малых массивов *Bead Sort* показал неплохие результаты, однако с увеличением размера массива время выполнения резко увеличивается.
* Это связано с тем, что временная сложность алгоритма зависит от суммы всех значений массива, а пространственная сложность делает алгоритм ресурсоемким для больших данных с высокими значениями элементов.
* График демонстрирует стабильную производительность на массивах с небольшим диапазоном значений, что делает алгоритм подходящим для узкоспециализированных задач.

**Tree Sort:**

* *Tree Sort* показывает линейно-логарифмическую зависимость времени выполнения на случайных массивах, если дерево сбалансировано. Однако в худшем случае (если массив был отсортирован заранее) сложность возрастает до *O(n²)*.
* Алгоритм демонстрирует высокую эффективность на массивах среднего размера с равномерным распределением данных.
* График показывает, что *Tree Sort* является надежным выбором для сортировки массивов, где значения распределены случайным образом.

**График 2** отображает время выполнения тех же алгоритмов для массивов размером до 80000 элементов:

**Comb Sort:**

* На больших массивах *Comb Sort* демонстрирует стабильный рост времени выполнения, оставаясь в пределах в большинстве случаев.
* Алгоритм сохраняет свою эффективность, однако его производительность может ухудшиться при наличии значительного количества "черепах" в массиве (маленьких значений, затрудняющих сортировку).

**Bead Sort:**

* *Bead Sort* показывает резкий рост времени выполнения, что делает его практически непригодным для больших массивов. Его эффективность значительно снижается из-за высокой временной и пространственной сложности при увеличении диапазона значений.
* График демонстрирует, что *Bead Sort* лучше использовать для массивов с малым количеством элементов и небольшим диапазоном значений.

**Tree Sort:**

* *Tree Sort* продолжает показывать линейно-логарифмическую зависимость времени выполнения для массивов со случайным распределением данных.
* На больших массивах алгоритм сохраняет свою эффективность, особенно при использовании сбалансированных деревьев. Однако при работе с массивами, заранее отсортированными в обратном порядке, производительность значительно ухудшается.

1. Заключение

В ходе выполнения лабораторной работы были исследованы три алгоритма сортировки: *Comb Sort*, *Bead Sort* и *Tree Sort*. Все алгоритмы были реализованы на языке *C++* и протестированы на массивах различной длины и распределений данных.

**Comb Sort** продемонстрировал хорошую производительность на массивах среднего размера и оказался наиболее подходящим для частично отсортированных данных.

**Bead Sort**, несмотря на свою уникальность, оказался малоэффективным для больших массивов из-за высокой временной и пространственной сложности. Этот алгоритм подходит для небольших массивов с узким диапазоном значений.

**Tree Sort** показал стабильную и надежную производительность на случайных массивах благодаря своей логарифмической зависимости. Однако при работе с несбалансированными деревьями производительность ухудшается.

Полученные результаты подтверждают теоретические оценки временной и пространственной сложности для каждого из алгоритмов. Выбор подходящего метода сортировки зависит от характеристик данных и требований к производительности.

1. Приложения

