МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение высшего образования
«Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского» (ННГУ)

Институт информационных технологий, математики и механики

Курс: Прикладная математика и информатика

ОТЧЕТ

по лабораторной практике

Тема:

«Изучение алгоритмов сортировок»

Выполнил:
студент группы
3824Б1ПМ4
Каргин Н. А.

_____
подпись

Руководитель практикой:
Преподаватель предмета «языки и методы программирования»
Мееров И. Б.

_____
подпись

Нижний Новгород
2024

Содержание

Введение

Сортировка массивов - одна из фундаментальных задач в программировании, которая заключается в упорядочении элементов массива в соответствии с определенным критерием. Эта задача имеет важное значение во многих областях, таких как базы данных, алгоритмы поиска, машинное обучение и другие.

Сортировка массивов может быть выполнена различными способами, каждый из которых имеет свои преимущества и недостатки. Выбор оптимального алгоритма сортировки зависит от размера массива, типа данных, требований к памяти и другим факторам.

Целью данной лабораторной работы является исследование и сравнение различных алгоритмов сортировки массивов, таких как пузырьковая сортировка, сортировка подстановкой и сортировка выбором. Мы будем анализировать сложность, эффективность и стабильность этих алгоритмов, а также сравнивать их производительность на различных типах данных и размерах массивов.

Лабораторная работа будет включать в себя следующие этапы:

- Изучение теоретических основ алгоритмов сортировки массивов.
- Реализация алгоритмов сортировки на языке программирования C.
- Тестирование и сравнение производительности алгоритмов на различных типах данных и размерах массивов.
- Анализ результатов и выводы о наиболее эффективных алгоритмах для различных случаев.
- Результаты данного проекта будут полезны для программистов, разработчиков и исследователей, которые работают с массивами данных и стремятся оптимизировать производительность своих алгоритмов.

Существует большое количество различных сортировок, которые применяются повсеместно в программах. Алгоритмы сортировок помогают сэкономить такие ресурсы, как время работы какой-либо части кода и,

соответственно, время человека и память, используемую для выполнения вашей программы. Например:

При редактировании файла нет необходимости держать весь файл в оперативной памяти. Каждой строке присваивается порядковый номер, и после внесения изменений достаточно обновленные строки отсортировать и вставить в сам файл.

Группировка элементов по признакам, которые характеризуются определенным числом или текстовой величиной (символы также поддаются упорядочиванию, например, по расположению их в алфавите или по номеру в таблице символов Юникода)

При сравнении двух и более таблиц или файлов для экономии времени достаточно отсортировать их. Таким образом, не надо "бегать" от одной строки к другой, а анализ будет более удобным и структурированным.

Алгоритмы сортировок применяются в различных областях, причина их использования состоит и в упорядочивании тех или иных структур для простоты понимания человеком, и в оптимизации работы программы по отношению к ресурсам компьютера.

Цель работы

Цель работы проанализировать самые простые варианты сортировок данных и найти среди них самый эффективный.

Сортировка «Пузырьком» (Bubble sort)

Сортировка пузырьком - это один из самых простых и интуитивно понятных алгоритмов сортировки, который часто используется в качестве введения в тему алгоритмов сортировки. Несмотря на свою простоту, этот алгоритм является эффективным способом упорядочивания данных, особенно для небольших наборов данных. В этой статье мы детально рассмотрим принцип работы сортировки пузырьком, проанализируем её временную и пространственную сложность, а также обсудим преимущества и ограничения этого алгоритма.

Принцип работы сортировки пузырьком

Сортировка пузырьком основана на сравнении соседних элементов и обмене их местами, если они стоят в неправильном порядке. Алгоритм работает следующим образом:

1) Начиная с первого элемента, сравниваем его со следующим элементом.

2) Если текущий элемент больше следующего, меняем их местами.

3) Переходим к следующей паре элементов и повторяем шаги 1-2.

4) Продолжаем этот процесс, пока не пройдем весь массив.

5) Повторяем шаги 1-4, пока массив не будет полностью отсортирован.

Таким образом, на каждой итерации цикла самый большой элемент "всплывает" в конец массива, словно пузырь. Отсюда и название "сортировка пузырьком".

В сортировке методом пузырька количество итераций внешнего цикла определяется длинной списка минус единица, так как когда второй элемент становится на свое место, то первый уже однозначно минимальный и не требует сортировки.

Количество итераций внутреннего цикла зависит от номера итерации внешнего цикла, так как конец списка уже отсортирован, и выполнять проход по этим элементам смысла нет.

Реализация на языке программирования C:

```
void BubbleSort(int mas[], int size) {
    for (int i = 0; i < size ; ++i) {
        bool flag = true;
        for (int j = 0; j < size - i - 1; ++j) {
            if (mas[j] > mas[j + 1]) {
                int temp = mas[j];
                mas[j] = mas[j+1];
                mas[j+1] = temp;
                flag = false;

            }
        }
        if (flag) {
            break;
        }
    }
}
```

Асимптотическая сложность

Асимптотическая сложность сортировки пузырьком зависит от того, насколько упорядочен исходный массив. В худшем случае, когда массив отсортирован в обратном порядке, асимптотическая сложность алгоритма составляет O(n^2), где n - размер массива. Это связано с тем, что на каждой итерации цикла нам нужно выполнять n-i сравнений, где i - номер текущей итерации.

Однако, если массив уже почти отсортирован, сортировка пузырьком может работать гораздо быстрее. В этом случае, асимптотическая сложность будет близка к O(n), поскольку нам потребуется всего несколько проходов по массиву, чтобы завершить сортировку.

Колличество элементов массива	**Время(мс)**
1000	0.001
10000	0.077
50000	1.921
100000	9.278

На скорость сортировки влияет:

Размер входных данных: Скорость работы сортировки пузырьком сильно зависит от количества элементов. При удвоении размера массива время выполнения увеличивается примерно в 4 раза.

Начальный порядок элементов: Чем больше беспорядок в исходном массиве, тем дольше будет работать алгоритм.

Преимущества:

- Простота реализации и понимания
- Не требует дополнительной памяти
- Стабильность (сохраняет относительный порядок равных элементов)

Недостатки:

- Неэффективность для больших массивов
- Квадратичная временная сложность в среднем и худшем случаях

Сортировка «Выбором» (SearchSort)

Алгоритм сортировки выбором основан на поиске минимального (или максимального, в зависимости от необходимости) элемента в неотсортированной части массива и его перемещении в начало (или конец) этой части.

Принцип работы сортировки выбором

1)Разделение массива: Алгоритм логически разделяет входной массив на две части:

- Отсортированная часть (изначально пустая)
- Неотсортированная часть (изначально весь массив)

2) Поиск минимального элемента: В неотсортированной части массива находится минимальный элемент.

3)Обмен элементов: Найденный минимальный элемент меняется местами с первым элементом неотсортированной части.

4)Расширение отсортированной части: Граница между отсортированной и неотсортированной частями сдвигается вправо на один элемент.

5)Повторение: Шаги 2-4 повторяются, пока неотсортированная часть не станет пустой.

Алгоритм сортировки выбором можно использовать для сортировки небольших массивов или для образовательных целей. Однако для больших данных рекомендуется использовать более эффективные алгоритмы, такие как быстрая сортировка или сортировка слиянием.

Реализация на языке программирования C:

```
void SearchSort(int mas[], int size) {
    for (int i = 0; i < size; ++i) {
        int a = mas[i];
        int index = i;
        for (int j = i + 1; j < size; ++j) {
            if (mas[j] < mas[index]) {
                index = j;
            }
        }
        int tmp = mas[i];
```

```
            mas[i] = mas[index];
            mas[index] = tmp;

        }
}
```

Асимптотическая сложность

Асимптотическая сложность сортирровки выбором не сильно отличается от сортировки пузырьком. В худшем случае, когда массив отсортирован в обратном порядке, асимптотическая сложность алгоритма составляет O(n^2), где n - размер массива. Алгоритм всегда выполняет одно и то же количество сравнений и обменов, независимо от начального порядка элементов.

Внешний цикл выполняется (n-1) раз.

Внутренний цикл в первой итерации выполняется (n-1) раз, затем (n-2) раз и так далее.

Общее количество сравнений: (n-1) + (n-2) + ... + 2 + 1 = n(n-1)/2, что дает квадратичную сложность O(n^2).

В лучшем случае, даже если массив уже отсортирован, алгоритм все равно должен пройтись по всем элементам для нахождения минимального элемента. Поэтому временная сложность в лучшем случае также составляет O(n2)

Колличество элементов массива	**Время(мс)**
1000	0.001
10000	0.044
50000	1.095
100000	4.377

Факторы, которые не влияют на скорость сортировки выбором

Порядок элементов в массиве: Сортировка выбором всегда производит полную сортировку массива, независимо от порядка элементов.

Количество повторений: Сортировка выбором не зависит от количества повторений в массиве.

Преимущества:

- Простота реализации: Алгоритм сортировки выбором является одним из самых простых алгоритмов сортировки. Он легко понимается и реализуется даже начинающими программистами.
- Эффективность для небольших массивов: Для небольших массивов (например, менее 100 элементов) сортировка выбором может быть достаточно эффективной и быстрой.
- Минимальное использование дополнительной памяти: Сортировка выбором не требует выделения дополнительной памяти, кроме нескольких переменных для хранения индексов и временных значений. Это делает ее подходящей для систем с ограниченными ресурсами.

Недостатки:

- Низкая эффективность для больших массивов: Для больших массивов (например, более 1000 элементов) сортировка выбором становится неэффективной, так как ее временная сложность составляет O(n2)O(n2).
- Медленная сортировка почти отсортированных массивов: Если массив уже почти отсортирован, сортировка выбором все равно будет выполнять полный проход по массиву, что может быть неэффективно.

Сортировка «Подстановкой» (Insertion Sort)

Сортировка вставками является стабильной, алгоритм сортировки на месте который строит окончательный отсортированный массив по одному элементу за раз. Он не самый лучший с точки зрения производительности, но традиционно более эффективен, чем большинство других простых O(n2) алгоритмов, такие как сортировка выбором или же пузырьковая сортировка.

Это также хорошо известный онлайн-алгоритм, поскольку он может сортировать список по мере его получения. Во всех других алгоритмах нам нужно, чтобы все элементы были предоставлены алгоритму сортировки перед его применением. Но сортировка вставками позволяет нам начать с частичного набора элементов, сортирует его (так называемый частично отсортированный набор). Если мы хотим, мы можем вставить больше элементов (это новый набор элементов, которых не было в памяти, когда началась сортировка) и отсортировать их. В реальном мире данные для сортировки обычно не статичны, а скорее динамичны. Если в процессе сортировки вставляется хотя бы один дополнительный элемент, другие алгоритмы не реагируют легко. Но только алгоритм сортировки вставками не прерывается и может хорошо реагировать на дополнительный элемент.

Принцип работы сортировки подстановкой:

1) Начиная со второго элемента массива (первый элемент уже отсортирован, поскольку он состоит из одного элемента), берем текущий элемент и сравниваем его с предыдущими элементами.
2) Если текущий элемент меньше предыдущего элемента, то мы перемещаем предыдущий элемент на одну позицию вправо, чтобы освободить место для текущего элемента.
3) Мы продолжаем сравнивать текущий элемент с предыдущими элементами и перемещать их, пока не найдем правильную позицию для текущего элемента.

4) Как только мы нашли правильную позицию для текущего элемента, мы вставляем его в эту позицию.
5) Мы повторяем шаги 1-4 для каждого элемента массива, начиная со второго элемента.

Реализация на языке программирования C:

```
void InsertionSort(int mas[], int size) {
    for (int i = 1; i < size; ++i) {
        for (int j = i - 1; j >= 0 && j < size; --j) {
            if (mas[j] > mas[j + 1]) {
                int tmp = mas[j + 1];
                mas[j + 1] = mas[j];
                mas[j] = tmp;
            }
        }
    }
}
```

Асимптотическая сложность

Худший случай возникает, когда массив отсортирован в обратном порядке. Для каждого элемента нужно сравнивать и сдвигать все предыдущие элементы. Количество операций пропорционально n * (n-1) / 2, что дает квадратичную сложность O(n^2).

Когда массив уже отсортирован алгоритм просто проходит по массиву один раз, выполняя только сравнения. Количество операций линейно зависит от размера массива O(n)

Колличество элементов массива	Время(мс)
1000	0.001
10000	0.078
50000	1.908
100000	7.692

Преимущества:

- Простой в реализации.
- Эффективен для небольших массивов.

- Стабильный (сохраняет относительный порядок элементов с одинаковыми значениями).
- Адаптивный (эффективен для частично отсортированных массивов).

Недостатки:

- Неэффективен для больших массивов из-за квадратичной сложности.

Применение:

- Часто используется в комбинации с другими алгоритмами сортировки.
- Полезен, когда новые элементы добавляются в уже отсортированный массив.

Сравнение алгоритмов. Вывод

Вид сортировки	**Кол-во элементов**	**Кол-во замен**	**Время (мс)**
Bubble sort	1000	245590	0.001
	5000	6121184	0.023
	10000	24612848	0.085
SearchSort	1000	1000	0.001
	5000	5000	0.011
	10000	10000	0.046
Insertion Sort	1000	247486	0.001
	5000	6075299	0.02
	10000	24655714	0.083

В этом примере мы используем один и тот же код для сортировки списка из 1000, 5000 и 10000 случайных целых чисел с помощью пузырьковой сортировки, сортировки выбором и сортировки вставками. Затем мы измеряем время выполнения каждого алгоритма с помощью модуля time.h и с помощью простого счетчика считаем колличество замен.

В случае относительно небольших массивов скорость сортировки у всех трех алгоритмов правктически равна и не требует большого колличества времени. Инетерснее становится при колличесвтве элементов начиная от 5000, когда уже наглядно видно, что сортировка выбором становится значительно быстрее конкурентов.

Как видно из результатов, для этого набора данных сортировка поиском выполняется намного быстрее, чем сортировка пузырьком и сортировка вставками. Однако важно отметить, что производительность каждого

алгоритма может варьироваться в зависимости от конкретных характеристик набора данных.

Используемый код:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <malloc.h>
void SearchSort(int mas[], int size) {
    for (int i = 0; i < size; ++i) {
        int a = mas[i];
        int index = i;
        for (int j = i + 1; j < size; ++j) {
            if (mas[j] < mas[index]) {
                index = j;
            }
        }
        int tmp = mas[i];
        mas[i] = mas[index];
        mas[index] = tmp;
    }
}
void InsertionSort(int mas[], int size) {
    for (int i = 1; i < size; ++i) {
        for (int j = i - 1; j >= 0 && j < size; --j) {
            if (mas[j] > mas[j + 1]) {
                int tmp = mas[j + 1];
                mas[j + 1] = mas[j];
                mas[j] = tmp;
            }
        }
    }
}

void BubbleSort(int mas[], int size) {
    for (int i = 0; i < size ; ++i) {
        bool flag = true;
        for (int j = 0; j < size - i - 1; ++j) {
            if (mas[j] > mas[j + 1]) {
                int temp = mas[j];
                mas[j] = mas[j+1];
                mas[j+1] = temp;
                flag = false;

            }
        }
        if (flag) {
            break;
        }
    }
}
int main() {
        int* mas;
        int* cloneMas;
        float seconds = 0;
        srand(time(NULL));
        int size = 0;
        int sort = 0;
        int cnt = 0;
        printf("Enter lentgth of namber ");
        scanf_s("%d", &size);
        mas = (int*)malloc(size * sizeof(int));
```

```
        cloneMas = (int*)malloc(size * sizeof(int));
        for (int i = 0; i < size; ++i) {
            mas[i] = rand() % 100;
        }
        while (true) {
            cnt = 0;
            for (int i = 0; i < size; ++i) {
                cloneMas[i] = mas[i];
            }
            printf("Choose name sort: \n 1 - Bubble Sort\n 2 - Search Sort \n 3 -
Insertion Sort\n");
            scanf_s("%d", &sort);
            clock_t start = clock();
            switch (sort) {
            case 1:
                BubbleSort(cloneMas, size);
                break;
            case 2:
                SearchSort(cloneMas, size);
                break;
            case 3:
                InsertionSort(cloneMas, size);
                break;
            }
            clock_t end = clock();
            seconds = (float)(end - start) / CLOCKS_PER_SEC;
            printf("\n");
            printf("time:%f\n", seconds);
        }
        free(mas);
        free(cloneMas);
        return 0;
}
```