Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Отчет по лабораторной работе №3 по дисциплине "Анализ алгоритмов"

Тема Алгоритмы сортировки

Студент Варламова Е. А.

Группа ИУ7-51Б

Оценка (баллы) _____

Преподаватель: Волкова Л.Л.

Москва — 2021 г.

Оглавление

Введение

Сортировка является очень важной задачей обработки информации. При этом под сортировкой понимают процесс упорядочивания элементов по какому-либо признаку в заданном массиве объектов. Например, текстовые данные можно отсортировать в лексикографическом порядке, а числовые в порядке неубывания или невозрастания.

В настоящее время многие программные системы работают с большим объёмом данных, поэтому возникает задача ускорения процесса обработки. Именно эту задачу и решают алгоритмы сортировки. Так, время поиска в отсортированном массиве пропорционально логарифму количества элементов, а в неотсортированном - пропорционально количеству элементов, что значительно медленнее.

Важной характеристикой любого алгоритма сортировки является скорость его работы, то есть время, за которое данные будут отсортированы. Время сортировки будет зависеть от количества сравнений и перестановок элементов, а также от длины массива данных.

Поэтому **целью** данной работы является получить навыки сравнительного анализа на примере 3 алгоритмов сортировки: сортировки пузырьком, сортировки выбором и быстрой сортировки.

Для достижения поставленной цели необходимо решить следующие **задачи**:

- изучить алгоритмы сортировки;
- провести сравнительный анализ алгоритмов на основе теоретических расчётов;
- реализовать алгоритмы сортировки;
- протестировать реализованные алгоритмы;
- провести сравнительный анализ реализаций алгоритмов по затраченному процессорному времени.

1 Аналитическая часть

1.1 Сортировка пузырьком

Идея алгоритма состоит в последовательных проходах по массиву. За один проход элементы всех пар стоящих рядом элементов сравнивают друг с другом и, если они стоят в неправильном порядке, меняют их местами. Проходы повторяются N-1 раз, где N - длина массива. Доказывается, что такое количество проходов достаточно для получения отсортированного массива. Действительно, при сортировке чисел по возрастанию за первый проход "выплывает" на последнее место наибольший элемент. Так, за один проход, как минимум, один элемент встаёт на правильное место. Поэтому после N-1 проходов все элементы будут на своих местах.

Из описания алгоритма очевидна модификация, ускоряющая его: если за проход не было обменов, то все элементы стоят на своих местах, следовательно, можно закончить выполнение сортировки.

При этом важно заметить, что алгоритм является устойчивым, то есть одинаковые по заданному признаку элементы останутся в том же порядке, что и до сортировки.

1.2 Сортировка выбором

Идея алгоритма состоит в том, что массив делят на 2 части: отсортированную и неотсортированную. Части отделяет текущий элемент. Текущий элемент сравнивают с каждым элементом из неотсортированной части. Если элементы стоят в неправильном порядке, то запоминают элемент из неотсортированной части и его позицию. Далее сравнивают уже запомненный элемент и элементы из неотсортированной части. Так продолжается до конца массива. Эти действия напоминают поиск минимума (максимума) в числовом массиве. В результате одного такого прохода получают позицию некоторого элемента. Если эта позиция не совпадает с позицией текущего элемента, то элементы меняют местами, иначе ничего не происходит. Такущий элемент помечается как граница отсортированной части, а текущим назначается следующий элемент. Алгоритм заканчивается тогда, когда отсортированной частью массива будет являться весь массив.

Важно заметить, что алгоритм не является устойчивым.

1.3 Быстрая сортировка

Идея алгоритма состоит в следующем:

- выбрать из массива элемент, называемый опорным. Это может быть любой из элементов массива. От выбора опорного элемента не зависит корректность алгоритма, но может зависеть его эффективность. Обычно выбирают средний элемент.
- сравнить все остальные элементы с опорным и переставить их в массиве так, чтобы разбить массив на два отрезка: «меньшие опорного» и «равные и большие».
- для двух отрезков рекурсивно применяем ту же последовательность действий, пока длина отрезка больше единицы.

1.4 Вывод

В данном разделе были описаны идеи трёх алгоритмов сортировки: сортировки пузырьком, сортировки выбором и быстрой сортировки.

2 | Конструкторская часть

2.1 Схемы алгоритмов

На рисунках 2.1, 2.2 и 2.3 показаны схемы алгоритмов сортировки пузырьком, выбором и быстрой сортировки соответственно.

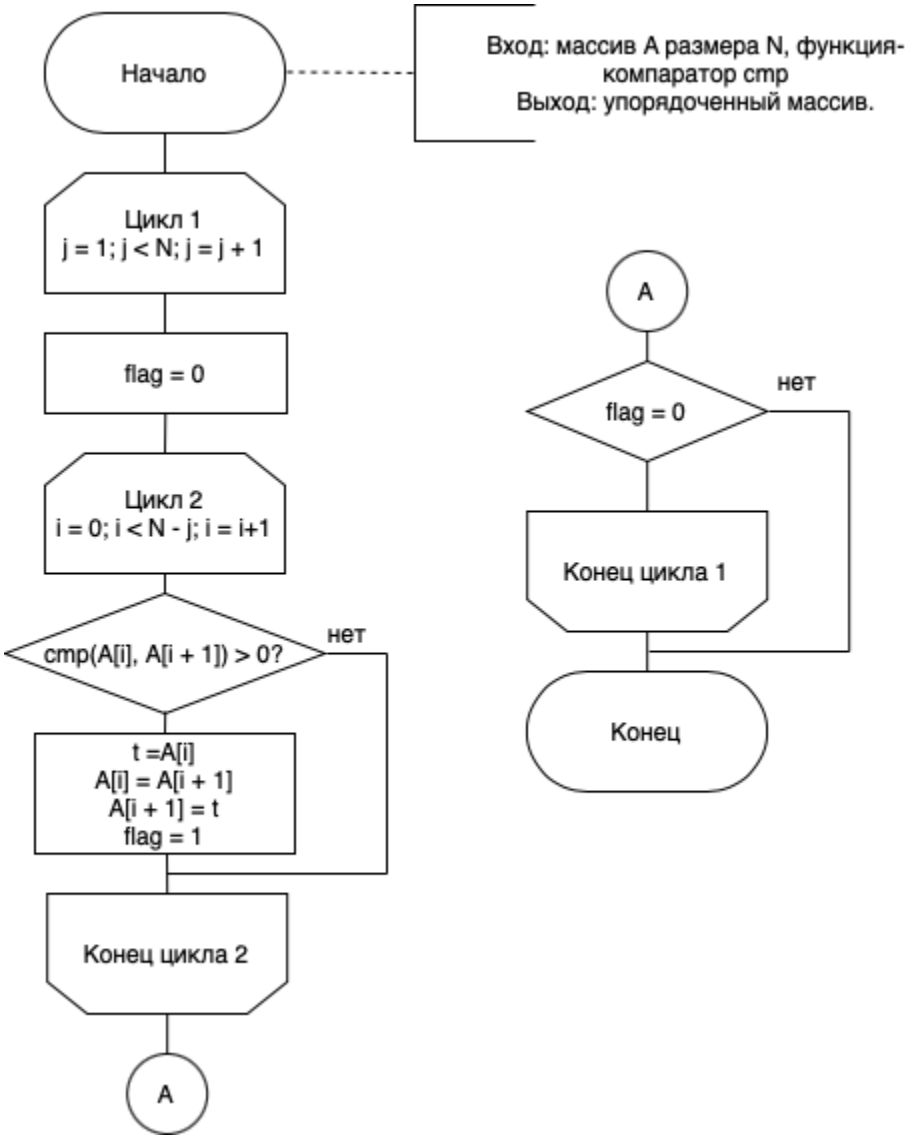

Рис. 2.1: Схема сортировки пузырьком

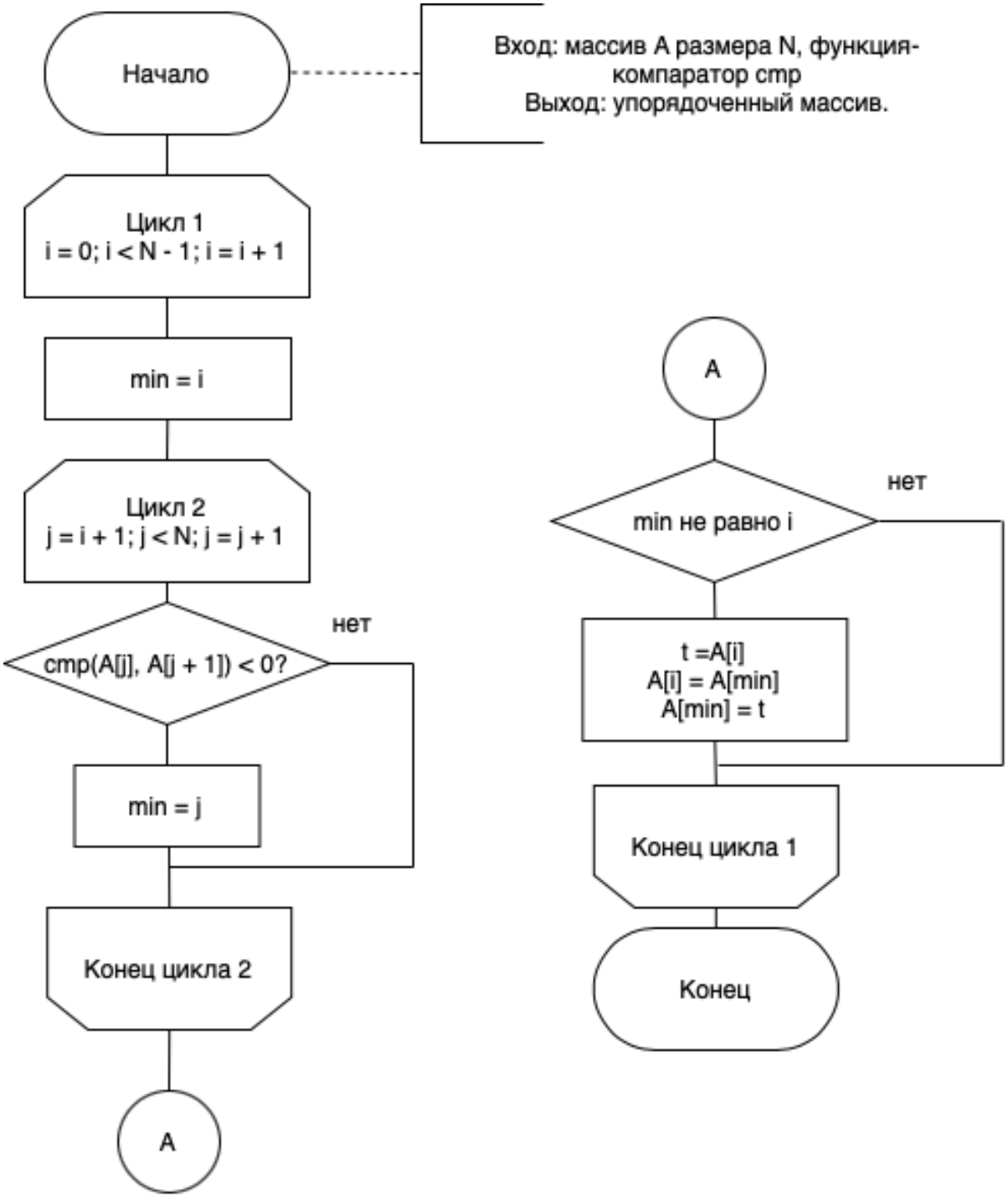

Рис. 2.2: Схема сортировки выбором

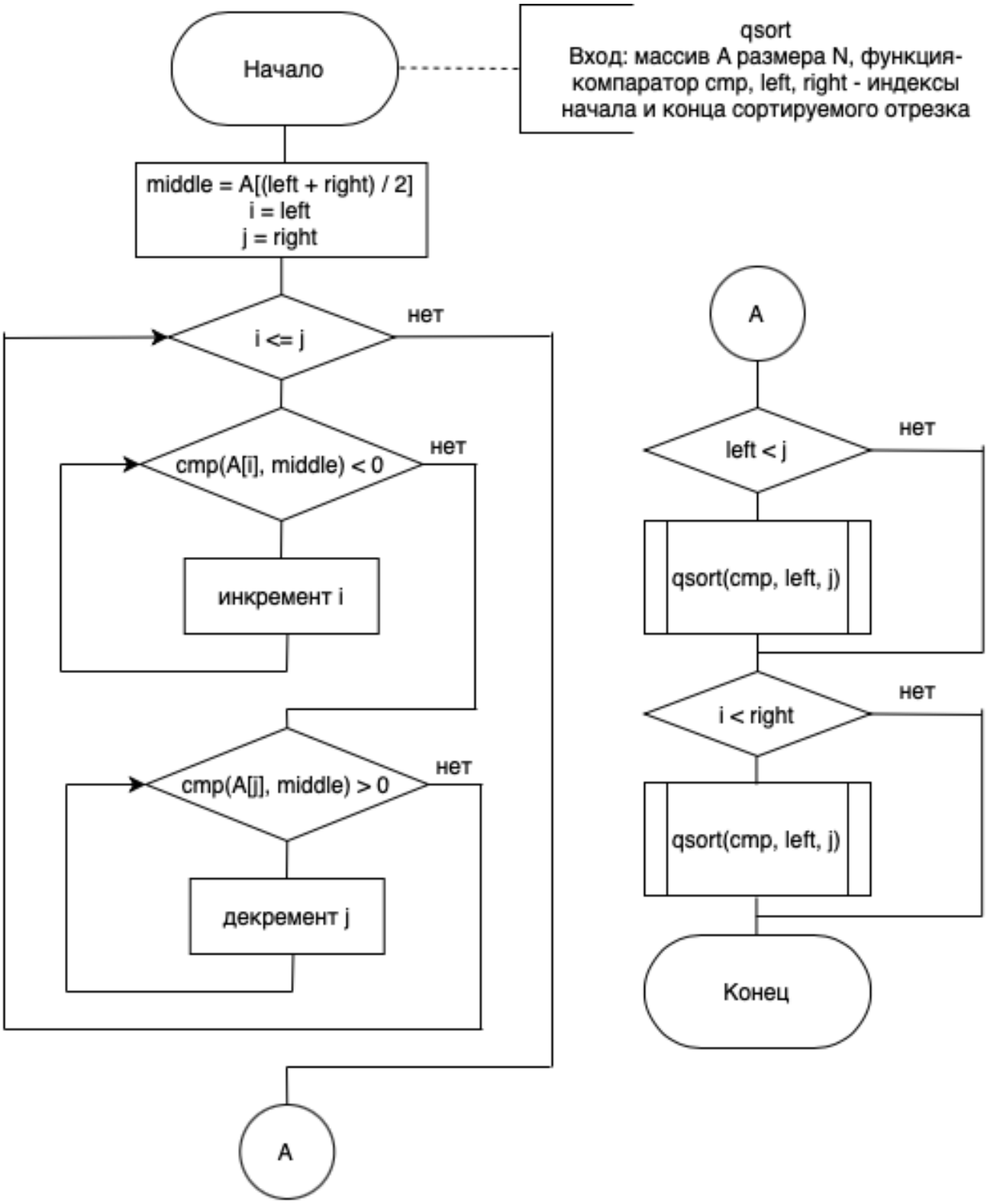

Рис. 2.3: Схема быстрой сортировки

2.2 Трудоемкость алгоритмов

Введем модель трудоемкости для оценки алгоритмов:

1. базовые операции стоимостью 1 — +, -, *, /, =, ==, <=, >=, !=, +=, [], получение полей класса
2. оценка трудоемкости цикла: Fц = a + N*(a + Fтела), где a - условие цикла
3. стоимость условного перехода возьмем за 0, стоимость вычисления условия остаётся.

Далее будет приведены оценки трудоемкости алгоритмов. Построчная оценка трудоемкости сортировки пузырькм с флагом (Табл. 2.1) и построчная оценка трудоемкости сортировки выбором (Табл. 2.2).

2.2.1 Сортировка пузырьком с флагом

Табл. 2.1 Построчная оценка веса

Код	Вес
bool flag = false;	1
int j = 1;	1
while (j < length)	1
{	0
flag = false;	1
int i = 0;	1
while (i < length - j)	2
{	0
if(cmp(arr[i], arr[i + 1]) > 0)	4
{	0
tmp = arr[i];	2
arr[i] = arr[i + 1];	3
arr[i + 1] = tmp;	2
flag = true;	1
}	0
i++;	1
}	0
if (flag == false) break;	1
j++;	1
}	0

Лучший случай: Массив отсортирован; не произошло ни одного обмена за 1 проход -> выходим из цикла
Трудоемкость: $1 + 1 + 1 + 1 * (1 + 1 + 2 + (n - 1) * (2 + 4 + 1) + 1) = 8 * n = O(n)$

Худший случай: Массив отсортирован в обратном порядке; в каждом случае происходил обмен, все внутренние циклы будут состоять из N - j итераций.
Трудоемкость:
Внутренний цикл: $(n - 1) * (n - 2) * (2 + 4 + 2 + 3 + 2 + 1 + 1)/2 = (n^2 - 3n + 3) * 15/2 = 7.5n^2 - 22.5n + 22.5$
Внешний цикл: $1 + 1 + 1 + (n - 1) * (1 + 1 + 1 + 2 + 1 + 1) = 7n - 4$
Итог: $7.5n^2 - 22.5n + 22.5 + 7n - 4 = 7.5n^2 - 15.5n + 18.5 = O(n^2)$

2.2.2 Сортировка выбором

Табл. 2.2 Построчная оценка веса

Код	Вес
int i = 0;	1
while (i < length - 1)	2
{	0
min = i;	1
int j = i + 1;	2
while (j < length)	1
{	0
if(cmp(arr[j], arr[min]) < 0)	4
{	0
min = j;	1
}	0
j++;	1
}	0
if (min != i)	1
{	0
tmp = arr[i];	2
arr[i] = arr[min];	2
arr[min] = tmp;	2
}	0
i++;	1
}	0

Лучший случай: отсортированный массив, обмены не производятся.
Трудоемкость:
Внешний цикл: $1 + 2 + (n-1) * (2 + 1 + 2 + 1 + 1) = 7n - 4$
Внутренний цикл: $(n-1) * (n-2) * (1 + 4 + 1)/2 = 3n^2 - 9n - 9$
Итог: $3n^2 - 9n - 9 + 7n - 4 = 3n^2 - 2n - 11 = O(n^2)$

Худший случай: массив отсортирован в обратном порядке, все обмены будут произведены.
Трудоемкость:
Внешний цикл: $1 + 2 + (n-1) * (2 + 1 + 2 + 1 + 2 + 2 + 2 + 1) = 13n - 10$
Внутренний цикл: $(n-1) * (n-2) * (1 + 4 + 1 + 1)/2 = 3.5n^2 - 10.5n - 10.5$
Итог: $3.5n^2 - 10.5n - 10.5 + 13n - 10 = 3.5n^2 - 2.5n - 20.5 = O(n^2)$

2.2.3 Быстрая сортировка

Лучший случай: сбалансированное дерево вызовов $O(n * log(n))$. В сбалансированном варианте при каждой операции разделения массив делится на две одинаковые части, следовательно, максимальная глубина рекурсии, при которой размеры обрабатываемых подмассивов достигнут 1, составит log2n. В результате количество сравнений, совершаемых быстрой сортировкой, равно значению рекурсивного выражения $C_n = 2 * C_{n/2} + n$, что дает общую сложность O(nlogn) [1].

Худший случай: несбалансированное дерево $O(n^2)$. В несбалансированном варианте каждое разделение даёт два подмассива размерами 1 и n 1, то есть при каждом рекурсивном вызове больший массив будет на 1 короче, чем в предыдущий раз. В этом случае потребуется $\sum_{i=0}^{n}(n-i) = O(n^2)$ операций, то есть сортировка будет выполняться за квадратичное время [1].

2.3 Вывод

На основе описания алгоритмов, данного в аналитическом разделе, были построены схемы трёх алгоритмов сортировки, оценены их трудёмкости в лучшем и худшем случаях. Таким образом:

- сортировка пузырьком: лучший - $O(n)$, худший - $O(n^2)$
- сортировка выбором: лучший - $O(n^2)$, худший - $O(n^2)$

- быстрая сортировка: лучший - $O(nlogn)$, худший - $O(n^2)$

3 | Технологическая часть

3.1 Средства реализации

В качестве языка программирования был выбран C++, поскольку он является достаточно быстрым, а среды разработки – CLion. Время работы алгоритмов было замерено с помощью clock() [2].

3.2 Входные и выходные данные

На вход ПО получает массив сравнимых элементов и функцию сравнения двух элементов. На выходе – тот же массив, но отсортированный в заданном порядке.

3.3 Реализация алгоритмов

В листингах 3.1 - 3.3 приведена реализация трёх алгоритмов сортировки.

Листинг 3.1: Функция быстрой сортировки

```
template <typename Type>
void Array<Type>::_qsort(int (*cmp)(Type, Type), int left, int right)
{
    Type middle = data[(right + left) / 2];

    int i = left;
    int j = right;

    while (i <= j)
    {
        while (cmp(data[i], middle) < 0)
            i++;
        while (cmp(data[j], middle) > 0)
            j--;

        if (i <= j)
        {
            Type tmp = data[i];
            data[i] = data[j];
            data[j] = tmp;
            i++;
            j--;
        }
    }
    if (left < j)
        _qsort(cmp, left, j);
    if (i < right)
        _qsort(cmp, i, right);
}
```

Листинг 3.2: Функция сортировки массива пузырьком

```
template <typename Type>
void Array<Type>::bubble(int (*cmp)(Type, Type))
{
    bool fl;
    for (size_t j = 1; j < length; j++) {
        fl = false;
        for (size_t i = 0; i < length - j; i++) {
            if (cmp(data[i], data[i + 1]) > 0) {
                Type tmp = data[i];
                data[i] = data[i + 1];
                data[i + 1] = tmp;
                fl = true;
            }
        }
        if (!fl)
            break;
    }
}
```

Листинг 3.3: Функция сортировки массива выбором

```
template <typename Type>
void Array<Type>::selection(int (*cmp)(Type, Type))
{
    for (size_t i = 0; i < length - 1; i++)
    {
        size_t min = i;
        for (size_t j = i + 1; j < length; j++)
        {
            if (cmp(data[j], data[min]) < 0)
            {
                min = j;
            }
        }
        if (min != i)
        {
            Type tmp = data[i];
            data[i] = data[min];
            data[min] = tmp;
        }
    }
}
```

3.4 Тестирование

В таблице 3.1 приведены тесты для функций, реализующих алгоритмы сортировки. Все тесты пройдены успешно.

Входной массив	Результат	Ожидаемый результат
$[-4, -4, -9, 9, 2]$	$[-9, -4, -4, 2, 9]$	$[-9, -4, -4, 2, 9]$
$[-5, 6, -1, 5, 8]$	$[-5, -1, 5, 6, 8]$	$[-5, -1, 5, 6, 8]$
$[7]$	$[7]$	$[7]$
$[7, 1, 5, 2, 10]$	$[1, 2, 5, 7, 10]$	$[1, 2, 5, 7, 10]$
$[1, 0]$	$[0, 1]$	$[0, 1]$

Таблица 3.1: Тестирование реализованных алгоритмов

3.5 Вывод

В данном разделе были выбраны средства разработки и с помощью них реализованы три алгоритма сортировки: пузырьком, выбором и быстрая сортировка. Кроме того, было проведено тестирование реализованных алгоритмов.

4 Исследовательская часть

4.1 Технические характеристики

Ниже приведены технические характеристики устройства, на котором было проведено тестирование ПО:

- Операционная система: macOS Big Sur 64-bit.
- Оперативная память: 8 GB.
- Процессор: 1,4 GHz 4-ядерный процессор Intel Core i5.

4.2 Время выполнения алгоритмов

Процессорное время выполнения алгоритма замерялось с помощью clock()[2]: для более точного замера каждая сортировка на каждой длине массива считалась несколько раз, а затем время усреднялось. На рисунках 4.1, 4.2 и 4.3 показаны графики зависимости процессорного времени выполнения сортировок от размера массива.

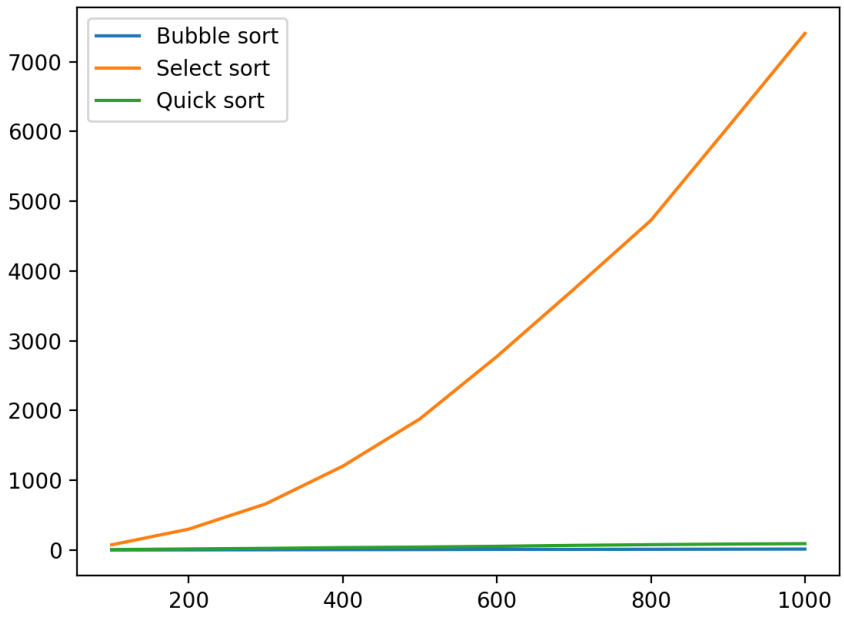

Рис. 4.1: Сортировки на отсортированных данных

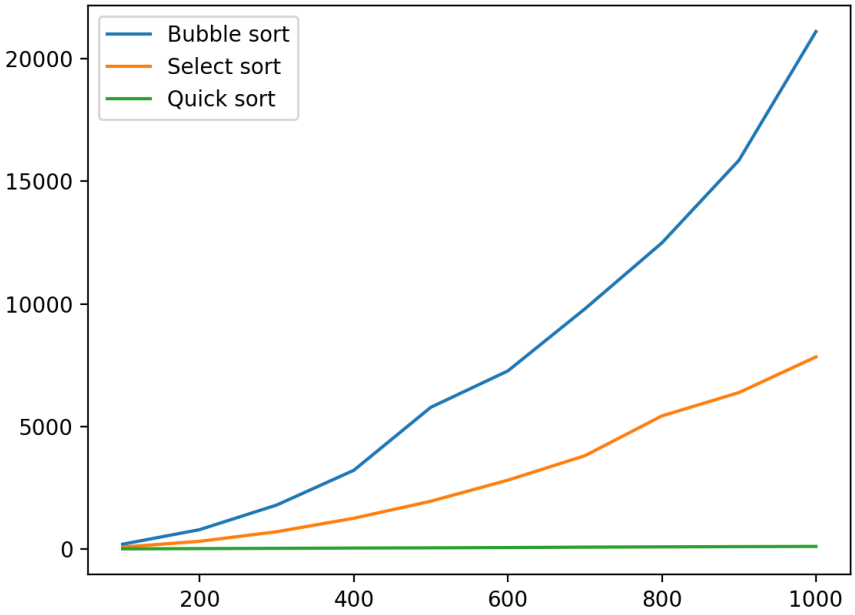

Рис. 4.2: Сортировки на отсортированных в обратном порядке данных

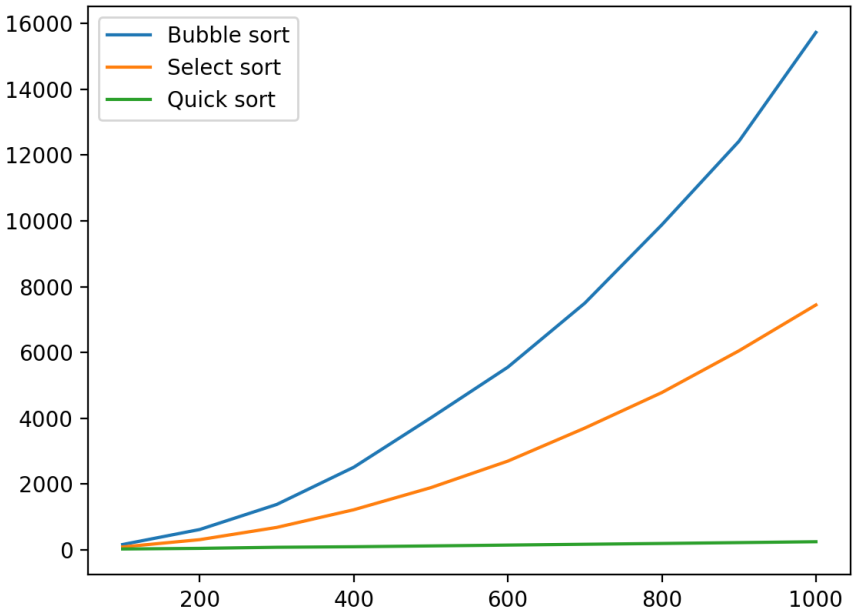

Рис. 4.3: Сортировки на произвольных данных

Таблица 4.1: Таблица времени выполнения сортировок на отсортированных данных (в мс)

Размер	bsort	ssort	qsort
100	2.30	88.39	8.21
200	3.65	330.28	19.30
300	5.89	712.55	28.20
400	6.95	1240.77	38.60
500	8.20	1994.25	46.69
600	11.26	2759.52	57.24
700	11.50	3726.45	69.03
800	12.59	4962.32	80.66

Таблица 4.2: Таблица времени выполнения сортировок на отсортированных данных в обратном порядке (в мс)

Размер	bsort	ssort	qsort
100	210.89	81.22	9.19
200	805.45	308.49	20.80
300	1798.66	701.04	29.60
400	3217.89	1642.31	91.78
500	5283.90	1886.37	49.99
600	7117.10	2815.29	65.26
700	9821.45	3722.31	78.11
800	12687.10	4902.62	89.13

Таблица 4.3: Таблица времени выполнения сортировок на случайных данных (в мс)

Размер	bsort	ssort	qsort
100	155.76	81.33	17.55
200	608.76	308.62	40.25
300	1397.26	697.15	65.97
400	2490.49	1249.19	90.36
500	4549.81	1868.65	107.29
600	5471.37	2758.52	140.66
700	8677.94	3838.68	162.69
800	9923.00	4747.77	184.41

4.3 Вывод

Как и ожидалось в результате оценки трудоемкости алгоритмов, сортировка пузырьком работает очень быстро на уже отсортированном массиве (лучший случай - оценка O(n)) и очень меделенно на отсортированном в обратном порядке (худший случай - оценка O(n^2)). Время сортировки выбором на всех трёх видах массивов примерно одинаково, поскольку лучший и худший случаи работают за квадратичное время. Быстрая сортировка работает значительно лучше двух других сортировок во всех трёх приведённых случаях.

Заключение

В рамках данной лабораторной работы были решены следующие задачи:

- изучены и реализованы 3 алгоритма сортировки: пузырьком, выбором, быстрая сортировка;
- протестированы реализованные алгоритмы;
- проведён сравнительный анализ трудоёмкости алгоритмов на основе теоретических расчетов;
- проведён сравнительный анализ алгоритмов по затраченному процессорному времени.

На основании анализа трудоемкости алгоритмов было показано, что алгоритм сортировки пузырьком имеет наименьшую сложность (линейную) в уже отсортированном массиве и квадратичную в отсортированном обратно, сортировка выбором имеет квадратичную сложность в лучшем, худшем и среднем случаях, а быстрая сортировка работает в лучшем и среднем случае значительно лучше (O(nlogn)) двух других алгоритмов, однако при неудачно выбранном опорном элементе и она может достигать квадратичной сложности. Те же выводы были подверждены экспериментально на основе замеров процессорного времени работы алгоритмов.

Литература

1. Кормен Т. Алгоритмы: построение и анализ [Текст] / Кормен Т. - Вильямс, 2014. - 198 с. - 219 с.
2. Стандартная функция clock(), измеряющая процессорное время [Электронный ресурс]. Режим доступа: https://en.cppreference.com/w/cpp/chrono/c/clock. Дата обращения: 17.09.2021