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»

Отчёт
по лабораторной работе № 3

Название: Алгоритмы сортировок

Дисциплина: Анализ алгоритмов

Студент ИУ7-55Б _____ В.Г. Горячев
(Группа) (Подпись, дата) (И.О. Фамилия)

Преподаватель _____ Л.Л. Волкова
(Подпись, дата) (И.О. Фамилия)

Москва, 2020

Оглавление

Введение

Сортировка - это процесс упорядочивания наборов данных одного типа по возрастанию или убыванию значения какого-либо признака. Эта операция очень часто используется в самых разных программах, что, вместе с вариативностью выполнения самого процесса, послужило причиной создания множества алгоритмов. Три из них будут рассматриваться в рамках настоящей лабораторной работы.

1 Аналитическая часть

1.1 Цель и задачи

Целью данной лабораторной работы является реализация и сравнение по временной эффективности алгоритмов сортировки пузырьком с флагом, вставками и выбором, а также теоретическая оценка их трудоёмкости. Задачи:

1) реализовать алгоритмы сортировки;
2) дать теоретическую оценку трудоёмкости стандартного алгоритма умножения матриц, алгоритма Винограда и модифицированного алгоритма Винограда;
3) провести сравнительный анализ реализаций по затраченному времени.

1.2 Описание и формулы

В общем случае сортируемые значения называются ключами сортировки, поскольку сортировать можно абсолютно любые данные, если задать правила сравнения. В случае лабораторной работы ключами будут целые шестнадцатиразрядные числа, а упорядочивание проводится только по возрастанию.

1.2.1 Алгоритм сортировки пузырьком с флагом

Классический учебный алгоритм, не встречающий применения в реальных задачах. Идея состоит в повторяющихся проходах по сортируемому массиву. На

каждой итерации последовательно сравниваются соседние элементы, и, если порядок в паре неверный, то элементы меняют местами. За каждый проход по массиву как минимум один элемент встает на свое место, поэтому необходимо совершить не более $n-1$ проходов, где n - размер массива, чтобы отсортировать массив. Этот алгоритм можно немного оптимизировать, учтя, что после $i-$ прохода i последних элементов элементов будут уже упорядочены, что позволяет сократить ненужные проверки, а также учитывая то, что если за текущий проход не произошло ни одного обмена, то массив уже отсортирован.

1.2.2 Алгоритм сортировки выбором

Идея этого алгоритма состоит в том, чтобы на каждом $i-$ шаге находить $i-$ минимальный элемент и меняем его местами с $i-$ элементом в массиве. Таким образом будет получен массив, отсортированный по неубыванию.

1.2.3 Алгоритм сортировки вставками

Из всех перечисленных алгоритмов, сортировка вставками является одним из используемых на практике методов упорядочивания данных, правда, в качестве вспомогательной сортировки в составе более сложных алгоритмов. В этой сортировке на каждом шаге выбирается один из элементов входных данных и вставляется на нужную позицию в уже упорядоченной части массива до тех пор, пока весь массив не будет перебран.

1.3 Вывод

В этом разделе были сформулированы цели и задачи работы, а также даны описания алгоритмов сортировки пузырьком, вставками и выбором.

2 Конструкторская часть

Входные данные - размеры сортируемых массивов и их содержимое - задаются из массива длин и формируются программно с помощью генератора случайных чисел соответственно. После вывода демонстрационного примера программа начинает проводить тестовые измерения скорости работы реализаций алгоритмов.

2.1 Требования к программному обеспечению

Программа должна выводить пример сортировки массива небольшого размера с целью контроля правильности работы алгоритмов, а также информацию об измерениях времени.

Для удобства пользователя могут выводиться проценты, по которым можно наглядно увидеть соотношение времени работы алгоритмов.

2.2 Схемы алгоритмов

На рисунках 2.1, 2.2 и 2.3 представлены схемы алгоритмов сортировки пузырьком с флагом, вставками и выбором соответственно.

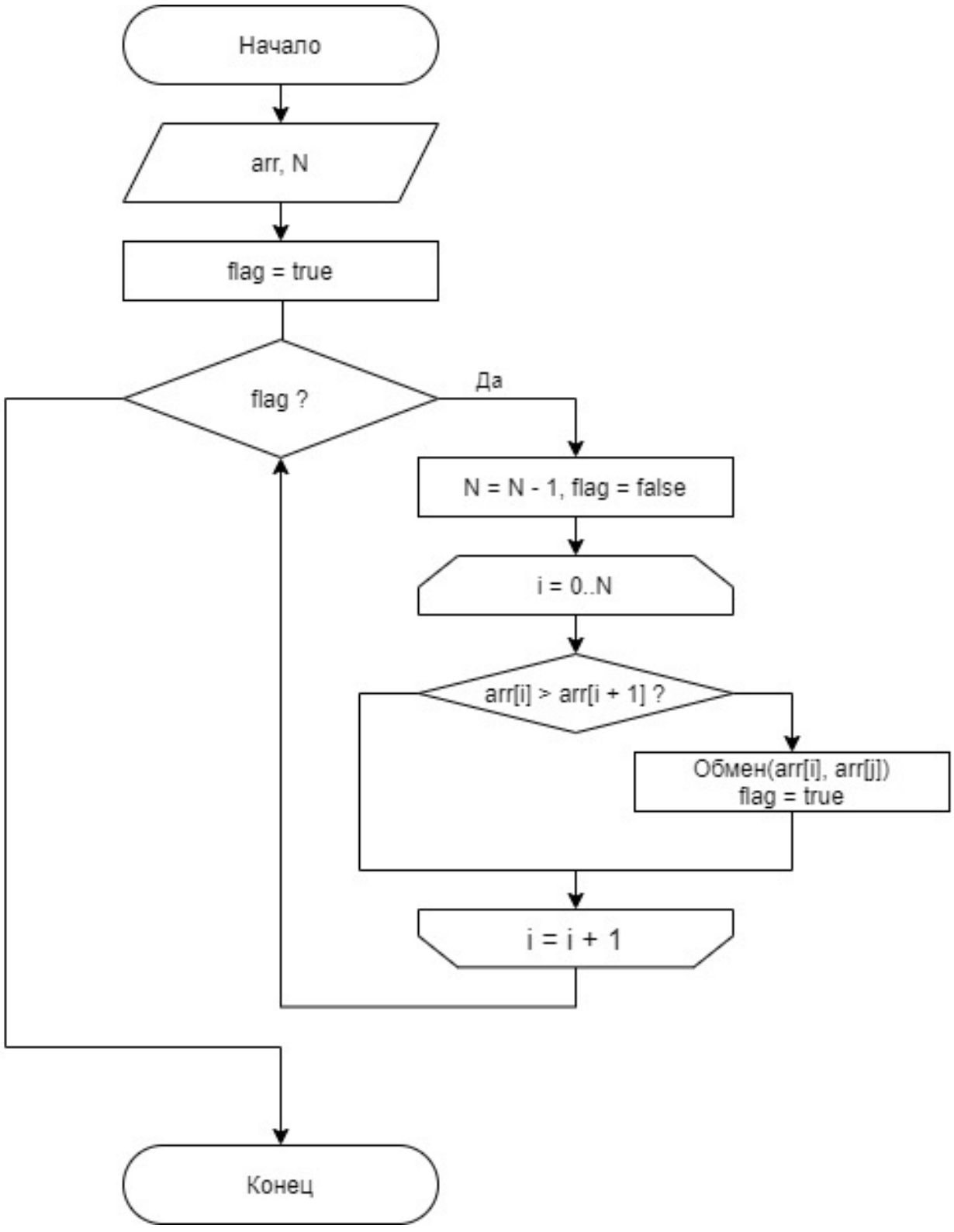


Рис. 2.1: Схема сортировки пузырьком с флагом

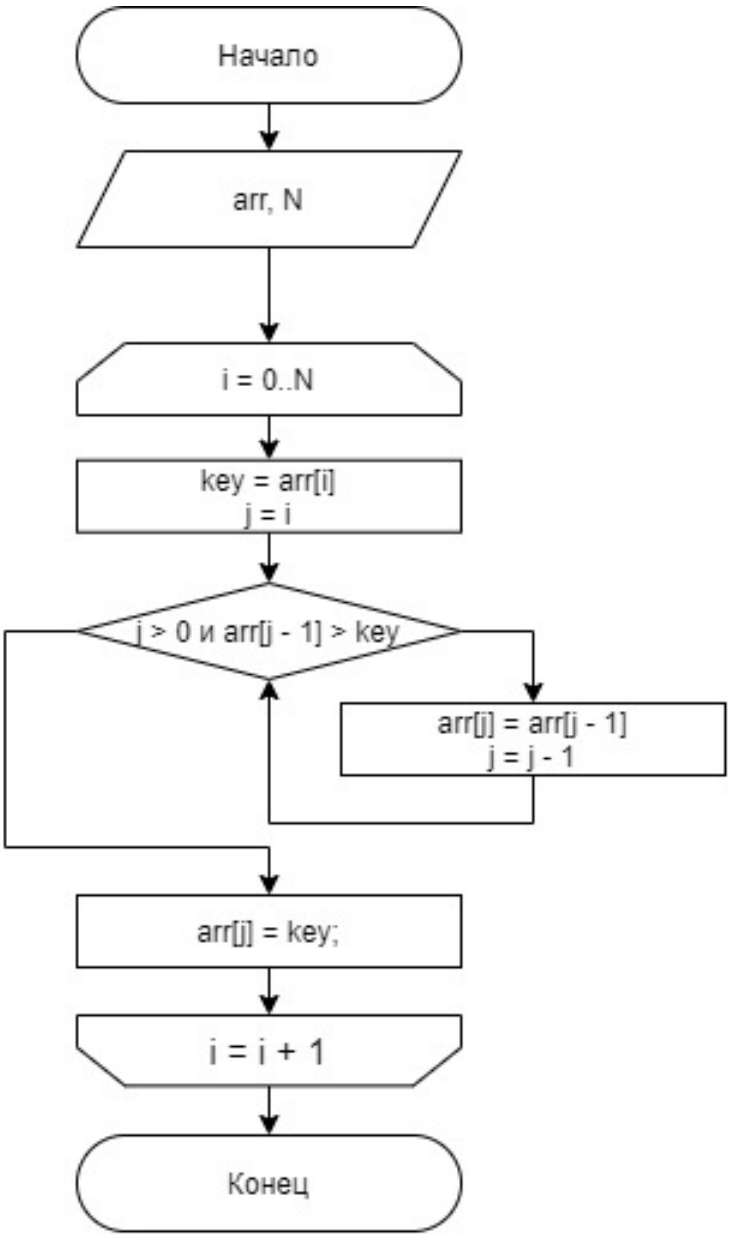


Рис. 2.2: Схема сортировки вставками

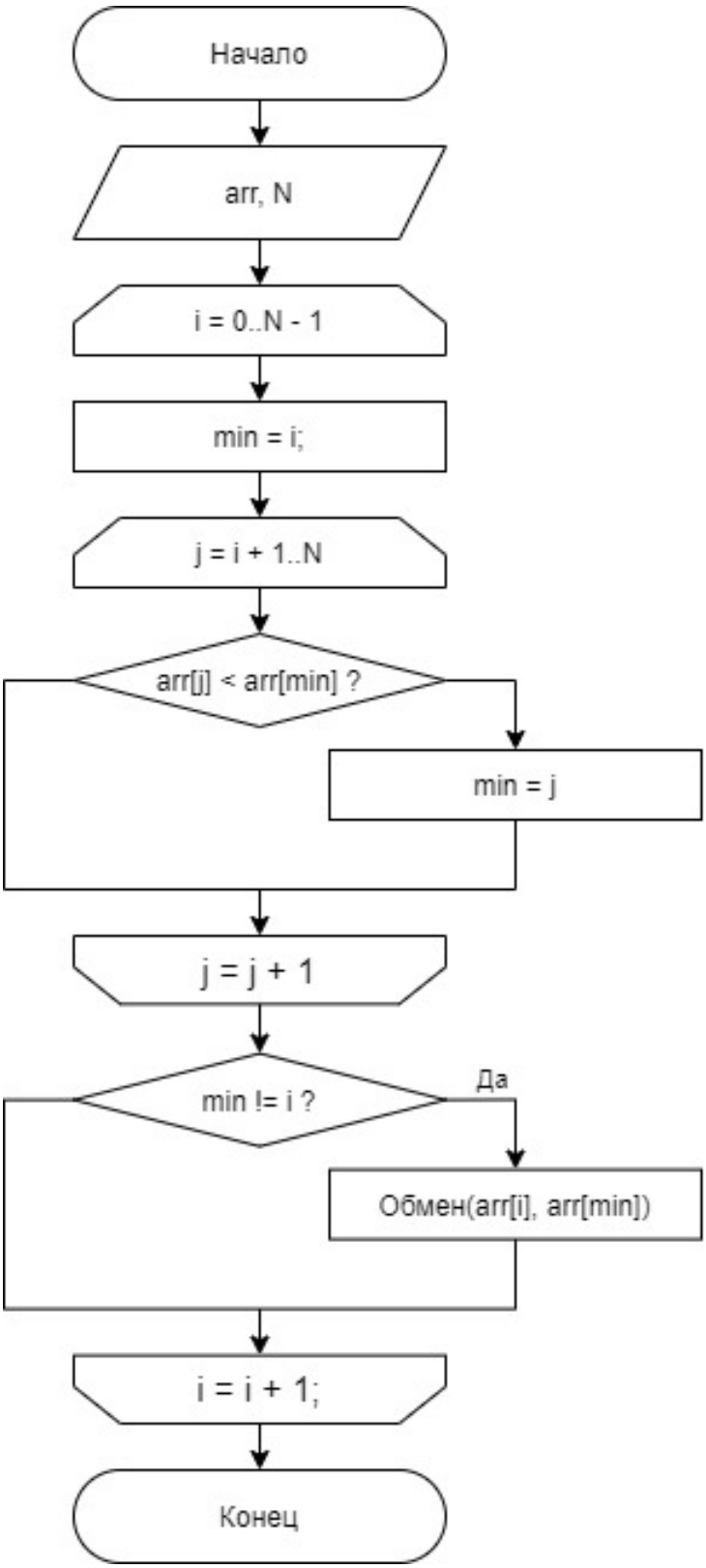


Рис. 2.3: Схема сортировки выбором

2.3 Модель трудоемкости

Модель трудоемкости для оценки алгоритмов:

1) стоимость базовых операций - 1:

   $=, +, \simeq, <, >, \geq, \leq, ==, !=, [], ++, --, <<, >>, , +=, -=, *, /, \%, *=, /=;$

2) стоимость цикла:

   $f_{for} = f_{init} + f_{comp} + M(f_{body} + f_{increment} + f_{comp})$

   Пример: $for(i = 0; i < N; i++)/*body*/$,

   Результат: $2 + N * (2 + f_{body})$ (в итоговых расчётах может отличаться из-за другого инкремента);

3) стоимость условного оператора:

   Пусть переход к одной из ветвей стоит 0, тогда

$$f_f = f_{cond} + \begin{cases} min(f_A, f_B), & \text{лучший случай} \\ max(f_A, f_B), & \text{худший случай} \end{cases}$$

2.4 Расчёт трудоёмкости алгоритмов

Будем считать, что обмен значений во всех алгоритмах реализован общим способом - через дополнительную переменную. Тогда трудоёмкость обмена может обыть вычислена по формуле (2.1).

$$f_{swap} = \text{`} = \text{`} + \text{`} = \text{`} + \text{`} = \text{`} = 1 + 1 + 1 = 3 \tag{2.1}$$

Также стоит заметить, что для расчётов, связанных со всеми выбранными алгоритмами, потребуется формула арифметической прогрессии (2.2), чтобы вычислить количество повторений за время работы.

$$S = \frac{N(N-1)}{2} \tag{2.2}$$

2.4.1 Трудоемкость сортировки пузырьком с флагом

Поскольку трудоёмкость этой сортировки зависит от случая, будут рассматриваться худший и лучший случаи.

Лучший случай: массив уже отсортирован в правильном порядке. По массиву будет совершён только один проход, благодаря наличию флага, и число сравнений составит $N - 1$, при этом не произойдёт ни одного обмена.

Тогда трудоёмкость будет представлена формулой (2.3).

$$1+1+1+[1+1+1+2+(N-1)(2+1+2+1)] = 3+[5+6N-6] = 6N+2 \approx 6N \quad (2.3)$$

Худший случай (массив отсортирован в обратном порядке): будет необходимо обойти весь массив целиком, количество обменов и сравнений при этом составит $\frac{n(n-1)}{2}$.

Вычислим трудоёмкость по формуле (2.4). Весь расчёт становится арифметической прогрессией из-за того, что N уменьшается на каждом шаге.

$$\begin{gathered} 3 + [5 + (N - 0 - 1)(6 + 3 + 1 + 1 + 1 + 1] + ... + [5 + 13(N - (N - 2) - 1)] = \\ 3 + 5 * (N - 1) + \frac{13}{2}N(N - 1) = 3 + 5N - 5 + \frac{13}{2}N^2 - \frac{13}{2}N = \\ \frac{13}{2}N^2 - 1.5N - 2 \approx \frac{13}{2}N^2 \end{gathered} \quad (2.4)$$

2.4.2 Трудоемкость сортировки вставками

Поскольку трудоёмкость этой сортировки зависит от случая, будут рассматриваться худший и лучший случаи.

Лучший случай: массив уже отсортирован в правильном порядке.

Тогда затраты будут представлены формулой (2.5).

$$2 + N(2 + 1 + 1 + (1 + 2 + 1 + 1 + 1) + 1 + 1) = 2 + 12N \approx 12N \quad (2.5)$$

Худший случай (массив отсортирован в обратном порядке): будет необходимо

обойти весь массив целиком, и количество сравнений при этом составит $\frac{n(n-1)}{2}$.

Вычислим трудоёмкость по формуле (2.6).

$$\begin{gathered} 2 + (12) + (12 + 10 * 1) + .. + (12 + 10 * (N - 1)) = \\ 2 + 12N + 10 * \frac{N(N-1)}{2} = 2 + 7N + 5N^2 \approx 5N^2 \end{gathered} \tag{2.6}$$

2.4.3 Трудоемкость сортировки выбором

Поскольку трудоёмкость этой сортировки зависит от случая, будут рассматриваться худший и лучший случаи.

Лучший случай: массив уже отсортирован в правильном порядке.

Тогда затраты будут представлены формулой (2.7).

$$\begin{gathered} 3 + (2 + 1 + 2 + 1 + (N - 1) * 5 + 1) + .. + (7 + 1 * 5) = \\ 3 + 7(N - 1) + 5 * \frac{N(N-1)}{2} = \\ \frac{5}{2}N^2 + \frac{9}{2}N - 4 \approx \frac{5}{2}N^2 \end{gathered} \tag{2.7}$$

Худший случай: массив отсортирован в обратном порядке.

Вычислим трудоёмкость по формуле (2.8).

$$\begin{gathered} 3 + (2 + 1 + 2 + 1 + (N - 1) * 6 + 1 + 3 + 2) + .. + (12 + 1 * 6) = \\ 3 + 12(N - 1) + 6 * \frac{N(N-1)}{2} = \\ 3N^2 + 9N - 9 \approx 3N^2 \end{gathered} \tag{2.8}$$

2.5 Вывод

В ходе конструкторской части были сформулированы требования к программе, разработаны схемы алгоритмов, с помощью которых можно осуществить написание программы, а также проведена теоретическая оценка в рамках выбранной

модели трудоёмкости алгоритмов.

3 Технологическая часть

3.1 Выбор языка программирования

Для написания программных реализаций алгоритмов был выбран язык программирования C++, ввиду изучения оного на прошлых курсах, привычности в работе и быстродействия программ.

Среда разработки – Visual Studio 2019.

В компиляторе полностью отключена автоматическая оптимизация, насколько это позволяют сделать настройки.

3.2 Реализация алгоритмов

В приводимых ниже листингах используется при подстановке в шаблоны тип Word – псевдоним для целочисленного типа short int, размер которого составляет 2 байта.

В функциях используется встроенная стандартная функция обмена значений двух переменных $std :: swap(T\&a, T\&b)$.

Реализация алгоритма сортировки пузырьком представлена на листинге 3.1.

Листинг 3.1: Реализация сортировки пузырьком с флагом

```
1 void bubbleSort(std::vector<Word>& arr) {
2     size_t N = arr.size();
3     bool swappedElements = true;
4     while (swappedElements) {
5         --N;
6         swappedElements = false;
```

```
7           for (size_t i = 0; i < N; ++i)
8               if (arr[i] > arr[i + 1])
9                   swap(arr[i], arr[i + 1]), swappedElements = true;
10      }
11  }
```

Реализация алгоритма сортировки вставками представлена на листинге 3.2.

Листинг 3.2: Реализация сортировки вставками

```
1  void insertionSort(std::vector<Word>& arr) {
2      for (size_t i = 1, j; i < arr.size(); ++i) {
3          Word key = arr[i];
4          for (j = i; j > 0 && arr[j - 1] > key; --j)
5              arr[j] = arr[j - 1];
6          arr[j] = key;
7      }
8  }
```

Реализация алгоритма сортировки выбором представлена на листинге 3.3.

Листинг 3.3: Реализация сортировки выбором

```
1  void selectionSort(std::vector<Word>& arr) {
2      for (size_t i = 0; i < arr.size() - 1; ++i) {
3          size_t min = i;
4
5          for (size_t j = i + 1; j < arr.size(); ++j)
6              if (arr[j] < arr[min])
7                  min = j;
8
9          if (min != i)
10             swap(arr[i], arr[min]);
11     }
12 }
```

3.3 Тестирование

Программа не подразумевает ручной ввод, что отбрасывает ситуации с заведомо ошибочными данными. Правильность расчётов можно отследить, сравнивая результаты между собой и с самостоятельно полученным расчётом по выведенным исходным данным. Пример такой проверки можно увидеть в таблице 3.1.

Таблица 3.1: Результаты тестирования программы

Исходн. массив	Ожидается	Пузырёк	Вставки	Выбор
Случайный массив				
15 23 4 0 16 2 27 17 29 23	0 2 4 15 16 17 23 23 27 29	0 2 4 15 16 17 23 23 27 29	0 2 4 15 16 17 23 23 27 29	0 2 4 15 16 17 23 23 27 29
Упорядоченный массив				
0 2 4 15 16 17 23 23 27 29	0 2 4 15 16 17 23 23 27 29	0 2 4 15 16 17 23 23 27 29	0 2 4 15 16 17 23 23 27 29	0 2 4 15 16 17 23 23 27 29
Обратный порядок				
29 27 23 23 17 16 15 4 2 0	0 2 4 15 16 17 23 23 27 29	0 2 4 15 16 17 23 23 27 29	0 2 4 15 16 17 23 23 27 29	0 2 4 15 16 17 23 23 27 29

Все результаты, полученные в ходе тестирования, соответствуют ожидаемым.

3.4 Вывод

В ходе технологической части были успешно реализованы и протестированы необходимые алгоритмы сортировки, что позволит произвести измерения времени работы и перейти к следующей части.

4 Исследовательская часть

4.1 Сравнительный анализ на основе замеров времени работы алгоритмов

Для получения более равномерного результата сортируемый массив генерируется заново в начале каждого повторного прогона сортировок. Для замеров быстродействия проводится несколько десятков повторных вызовов функций, количество повторений рассчитывается динамически с учётом размерна массива - чтобы сократить время ожидания результатов до разумных пределов, хотя в целом в этом нет необходимости - всё выполняется меньше, чем за полминуты реального времени. Тесты проводятся для массивов, длиной в 10, 100, 200, 500, 1000, 2000, 5000 и 10000 элементов. Быстродействие алгоритмов измеряется в процессорном времени, которое более точно отражает время работы алгоритмов, с помощью функции _rdtsc()[1].

Полученные данные представлены в табл. 4.1. Всё время в таблице разделено на 10^4 - чтобы упростить и сократить запись.

Таблица 4.1: Результаты тестовых измерений скорости работы алгоритмов (в тактах процессора) в зависимости от размера массива

N	T1 пузырёк	T2 вставки	T3 выбор
Упорядоченный массив			
10	0.018	0.031	0.117
50	0.074	0.136	2.383
100	0.139	0.256	9.042
200	0.286	0.537	36.459
500	0.753	1.436	239.196
1000	1.354	2.590	862.114
2000	2.699	5.059	3429.785
5000	6.663	12.841	21275.035
10000	14.057	27.023	86894.130
Произвольный массив			
10	0.199	0.105	0.195
50	4.491	1.585	3.007
100	18.186	6.066	11.247
200	72.625	23.370	42.254
500	454.877	140.746	239.948
1000	1748.111	517.587	904.662
2000	7207.294	2057.435	3524.982
5000	45086.734	12370.685	21382.270
10000	183879.607	49637.040	85473.846
Упорядоченный в обратном порядке массив			
10	0.207	0.117	0.107
50	5.298	2.743	2.735
100	22.488	11.362	10.539
200	88.478	43.955	40.057
500	524.449	262.728	235.900
1000	2115.284	1054.893	934.065
2000	8212.252	4073.004	3573.043
5000	51760.799	25681.077	22772.017
10000	203674.547	102849.666	90175.558

Полученные данные удобно воспринимать в виде графиков, представленных на рисунках 4.1, 4.2 и 4.3.

Рис. 4.1: Результаты тестовых измерений скорости работы алгоритмов (в тактах процессора) в зависимости от размера упорядоченного массива

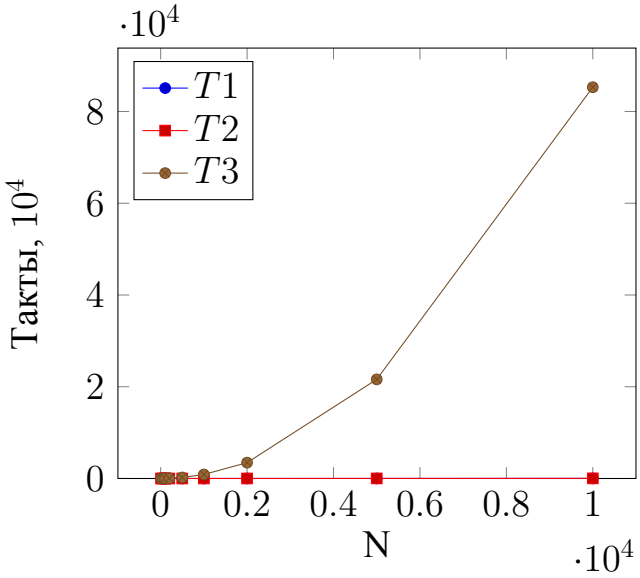
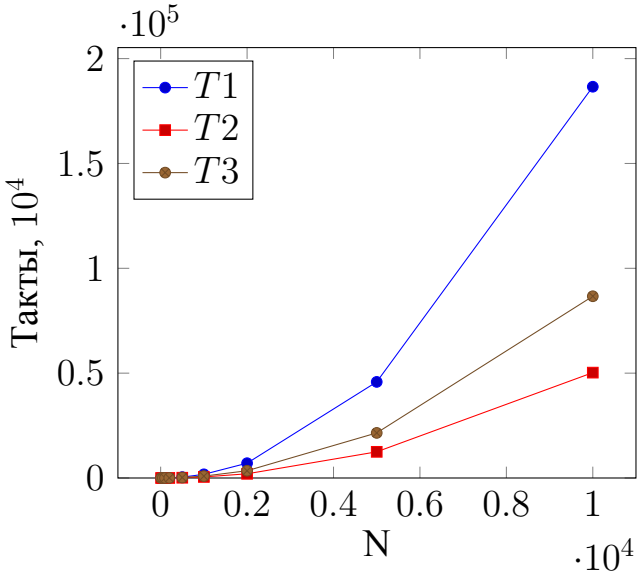
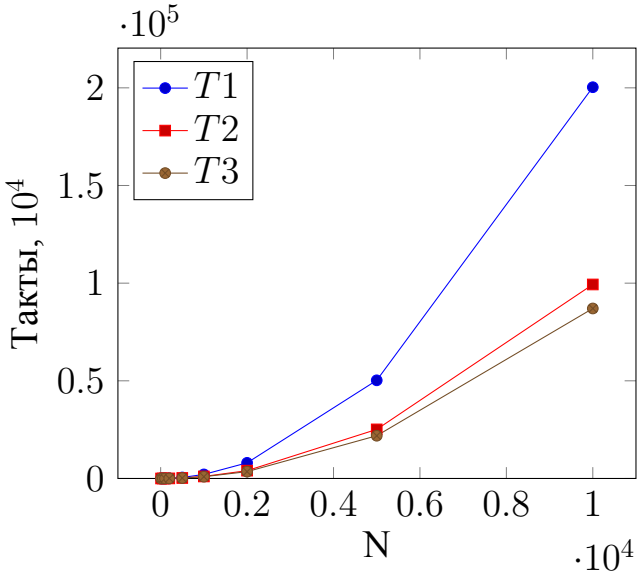


Рис. 4.2: Результаты тестовых измерений скорости работы алгоритмов (в тактах процессора) в зависимости от размера произвольного массива



Из оценок трудоёмкости видно, что сложность всех алгоритмов пропорциональна N^2, что и можно наблюдать на графике. Однако они обладают разными константами-множителями, что и обеспечивает различие положения линий.

Рис. 4.3: Результаты тестовых измерений скорости работы алгоритмов (в тактах процессора) в зависимости от размера обратно упорядоченного массива



4.2 Вывод

Результаты тестирования показывают ожидаемый результат, что в произвольном случае самым быстрым способом отсортировать массив является сортировка вставками, а самым медленным - сортировка пузырьком. Сортировка выбором занимает промежуточное положение.

При этом в случае упорядоченного массива лучший результат показывает сортировка пузырьком, а в худшей для всех трёх ситуации, когда массив отсортирован в обратном порядке, выигрывает сортировка выбором.

Стоит заметить, что все эти алгоритмы имеют одинаковую асимптотику для худшего случая - $O(n^2)$, но различия в скорости их работы обусловлены более аккуратным внутренним устройством, минимизирующим обмены, благодаря чему сортировки вставками и выбором имеют меньший множитель перед n^2, а значит, более высокую скорость работы.

Все рассмотренные сортировки имеют считанное количество внутренних переменных и поэтому не требуют никаких затрат памяти, пропорциональных длине входа.

Заключение

В ходе лабораторной работы были реализованы и изучены различные алгоритмы сортировок, проведены экспериментальное сравнение по затраченному времени и теоретическая оценка трудоёмкости.

Проведённое исследование позволяет сделать вывод, что фактически алгоритм сортировки вставками является лучшим среди изученных в произвольном случае.

Приведённое утверждение можно продемонстрировать в числах: сортировка вставками, для всех N, участвующих в эксперименте, быстрее сортировки выбором на 50-58% и требует кратно меньше времени, чем сортировка пузырьком, начиная от тех же 50% в пользу ”вставок”и заканчивая 3,7-кратной длительностью пузырька для N = 10 000.

Сортировка вставками несколько уступает другим алгоритмам в этой лабораторной работе, когда речь заходит о предельных ситуациях - упорядоченном или упорядоченном в обратном порядке массивах. В случае первого она оуступает сортировке пузырьком приблизительно в 2 раза, а в случае второго - сортировке выбором на 12-13%.

Так или иначе, все рассмотренные сортировки имеют квадратичную сложность, и поэтому даже быстрейшая из них будет уступать более сложным алгоритмам, если количество сортируемых элементов достаточно велико.

Список литературы

1. Функция _rdtsc() – документация Майкрософт [эл. ресурс]. Режим доступа: https://docs.microsoft.com/ru-ru/cpp/intrinsics/rdtsc?view=vs-2019 (дата обращения: 27.09.2020).

2. Дж. Макконнелл ”Основы современных алгоритмов. 2-е дополненное издание”. (дата обращения: 20.10.2020).