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 о лабораторной работе №3 по дисциплине "Анализ алгоритмов" на тему "Алгоритмы сортировки"

Студент Коняев Е.А

Группа ИУ7-53Б

Преподаватели Волкова Л.Л., Строганов Ю.В.

Дата сдачи отчета_____

Оценка (баллы) _____

Москва — 2022 г.

Оглавление

Введение

Алгоритмы сортировки — это набор инструкций, которые принимают массив или список в качестве входных данных и упорядочивают элементы в определенном порядке. Сортировка чаще всего осуществляется в числовом или алфавитном (или лексикографическом) порядке и может быть в порядке возрастания (A-Z, 0-9) или убывания (Z-A, 9-0).

Сортировки важны потому что они часто могут уменьшить сложность решаемой задачи, а потому алгоритмы сортировки очень важны в компьютерных науках. Эти алгоритмы имеют прямое применение в алгоритмах поиска, алгоритмах баз данных, методах разделяй и властвуй, алгоритмах структуры данных и многом другом.

При выборе алгоритма сортировки необходимо учитывать:

1. количиество сортируемых элементов;
2. количество доступной памяти;
3. возможность увеличения коллекции с сортируемыми элементами.

Эти факторы могут определить, какой алгоритм будет работать лучше всего в каждой ситуации. Некоторым алгоритмам может потребоваться много места или памяти для запуска, в то время как другим, при том, что они не являются самыми быстрыми, не требуется много ресурсов для запуска.

Именно из-за разнообразия алгоритмов сортровок, их особенностей и областей применения, данная работа была посвещена анализу некоторых из этих алгоритмов.

Глава 1

Аналитическая часть

1.1 Цели и задачи

Целью данной работой является исследования алгоритмов сортировок пузырьком, вставками и выбором. Для этого в ходе исследования алгоритмов требуется решить следующие задачи:

1. изучить и рассмотреть выбранный алгоритмы сортировок;
2. построить блок-схемы выбранных алгоритмов;
3. реализовать каждый из алгоритмов;
4. рассчитать их трудоемкость;
5. эксперитментально оценить временные характеристики алгоритмов;
6. сделать вывод на основании проделанной работы.

1.2 Сортировка расческой

Сортировка расческой (на анг. Comb sort) является улучшенным вариантом сортировки пузырьком. В сортировке пузырьком сравнениваются и перестановляются соседние элементы, то есть разница индексов этих элементов всегда 1. В соритровке же рассческой это расстояние увеличено и сравниваются элементы стоящие на большем расстоянии друг от друга. С очередным проходом это расстояние уменьшается. Таким образом, максимальные элементы, находящиеся в начале массива, можно поместить в его конец за меньшее количество перестановок (в сортировке по возрастанию).

Число, на которое должен раз за разом уменьшаться разрыв называется фактором уменьшения и равно 1.247 (выведено авторами сортировки). После каждой итерации расстояние между элементами делится на фактор, округляется, и так продолжается, пока оно не станет равным 1.

1.3 Карманная сортировка

Карманная (блочная, корзинная) сортировка — это сортировка, в ходе которой элементы, которые необходимо отсортировать, распределяются между конечным числом отдельных блоков (карманов) так, чтобы все элементы в каждом следующем по порядку блоке были всегда больше (или меньше, в зависимости от возрастания/убывания), чем в предыдущем. После этого каждый блок сортируется отдельно с помощью какого-либо алгоритма сортировки, либо соритруется рекурсивно карманной сортировкой. После этого элементы из полученных блоков по порядку помещаются обратно в массив, в результате чего получается отсоритрованные по возрастанию/убыванию входные данные.

1.4 Битонная сортировка

Битонная сортировка — это алгоритм сортировки, который часто используется в параллельных сортировках, так элементы сравниваются в предопределенной последовательности, которая не зависит от входных данных. Эта предопределенная последовательность называется битонной последовательностью. Битонная последовательность - это последовательность, в которой элементы сначала неубывают, а потом с определенного элемента невозрастают.

На первом этапе соритровки необходимо создать битонную последовательность из заданной случайной последовательности сортируемых элементов (например, отсортировать элементы по возрастанию, а затем, начиная с середины последовательности, развернуть элементы в обратном порядке). Далее необходимо поменять местами элементы между двумя половинами последовательности, если какой-либо элемент во второй половине окажется меньше. Таким образом мы получили все элементы в первой половине меньше, чем все элементы во второй половине. Результаты сравнения и обмена в две последовательности длиной $N/2$ каждая, где N - кол-во элементов в последовательности. Далее необходимо повторять процесс обмена рекурсивно для каждой половины последовательности пока не получим одну отсортированную последовательность длины N.

1.5 Вывод

В данном разделе были рассмотрены 3 алгоритма сортировки: расческой, карманная и битонная.

Глава 2

Конструкторская часть

2.1 Модель вычислений

Для вычисления трудоемкости будем использовать следующую модель вычислений:

1. Операции из списка (2.1) имеют трудоемкость 1.

$$+, -, /, \%, ==, !=, <, >, <=, >=, [], ++, --\tag{2.1}$$

2. Трудоемкость оператора выбора if условие then A else B рассчитывается, как (2.2).

$$f_{if} = f_{\text{условия}} + \begin{cases} f_A, & \text{условие выполняется,} \\ f_B, & \text{иначе.} \end{cases}\tag{2.2}$$

3. Трудоемкость вызова функции равна 0.

4. Трудоемкость цикла рассчитывается, как (2.3).

$$f_{for} = f_{\text{инициализации}} + f_{\text{сравнения}} + N(f_{\text{тела}} + f_{\text{инкремента}} + f_{\text{сравнения}})\tag{2.3}$$

2.2 Трудоёмкость алгоритмов

Размер массива во всех вычислениях обозначим как N.

2.2.1 Алгоритм сортировки расческой

Обозначим расстояние между просматриваемыми элементами как gap. Алгоритм перебирает элементы от 0 до $N - gap$ на каждом шаге. Обозначим кол-во перебираемых элементов на i-м шаге внешнего цикла как N_i. Тогда:

$$N_i = N - floor(\frac{gap_i}{1.247})\tag{2.4}$$

gap_i на i-м шаге вычисляется как:

$$gap_i = gap_{i-1} - floor(\frac{gap_{i-1}}{1.247})\tag{2.5}$$

В **худшем** случае все элементы отсортированы в обратном порядке. Вычисляя верхнюю границу N_i из уравнения (2.4) и рекуррентного соотношения (2.5) для наихудшего случая, мы получаем выражение, содержащее члены второй и первой степени N с константой, дающей верхнюю границу N^2. Таким образом, временная сложность наихудшего случая равна $O(N^2)$

Лучший случай возникает, когда все элементы уже отсортированы . В этом случае цикл с $gap = 1$ будет запущен только один раз (как и остальные). Тогда кол-во проходов внутреннего цикла, на i-й итерации можно выразить формулой:

$$N_i = \frac{N}{1.247^i} \tag{2.6}$$

А кол-во всех проходов выражается как:

$$S_N = N \sum_{r=1}^{n} \frac{1}{1.247^r} \tag{2.7}$$

Выражение под знаком суммы в формуле (2.7) приблизительно равняется $O(log(N))$. Тогда вся сумма будет равнятся $O(Nlog(N)$, что будет являться сложностью алгоритма в лучшем случае.

2.2.2 Алгоритм карманной сортировки

Обозначим кол-во карманов за k. Алгоритм состоит из 4-х последовательно идущих циклов:

1. поиска минимума и максимума среди всех элементов массива;
2. распределения элементов массива по соответсвтующим корзинам;
3. сортироки элементов каждой корзины другим алгоритмом сортировки;
4. соединения всех корзин воедино.

Цикл поиска максимума и минимума среди всех элементов работает за:

$$f_1 = 1 + 1 + N \cdot (2 * 3 + 1 + 1) = 2 + 8N = O(N) \tag{2.8}$$

Цикл распределения элементов массива по соответсвтующим корзинам работает за:

$$f_2 = 1 + 1 + N \cdot (12 + 1 + 1) = 2 + 14N = O(N) \tag{2.9}$$

Цикл сортироки элементов каждой корзины другим алгоритмом сортировки работает за:

1. В **лучшем** случае (элементы распределены по корзинам равномерно $O(k)$, входной массив расположен так, что внутренняя сортировка работает за лучшее время $O(N)$):

$$f_3 = O(N + k) \tag{2.10}$$

2. В **худшем** случае (элементы не имеют математической разницы между собой и внутренняя сортировка работает за худшее время $O(N^2)$):

$$f_3 = O(N^2) \tag{2.11}$$

Так как все N элементов расспределены равномерно по k корзинам, то цикл соединения всех корзин воедино работает за:

$$f_4 = O(N + k) \tag{2.12}$$

Итоговая сложность алгоритма будет вычисляться как $f = f_1 + f_2 + f_3 + f_4$. В **худшем** случае эта сложность равна $O(N^2)$, а в **лучшем** $O(N + k)$

2.2.3 Алгоритм битонной сортировки

Исходя из источников литературы сложность алгоритма битонной сортировки как в **лучшем**, так и в **худшем** случае равна $O(log^2(N)$.

2.3 Схемы алгоритмов

На рисунках ниже показаны схемы алгоритмов карманной сортировки, расческой и битонной соответственно.

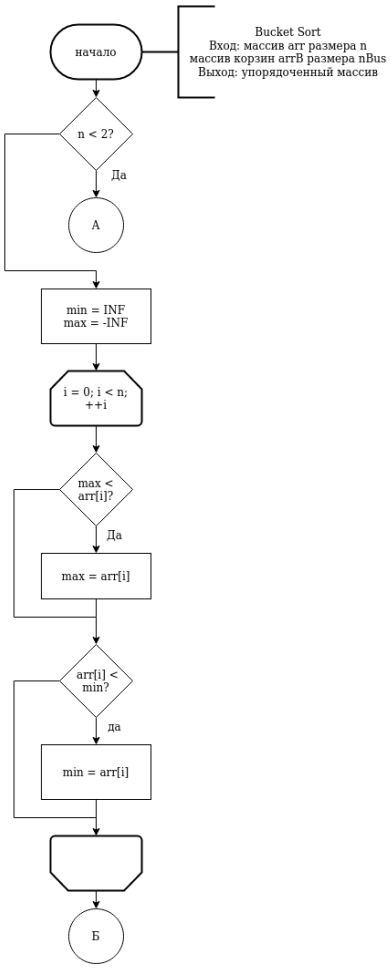


Рис. 2.1: Схема карманной сортировки ч.1

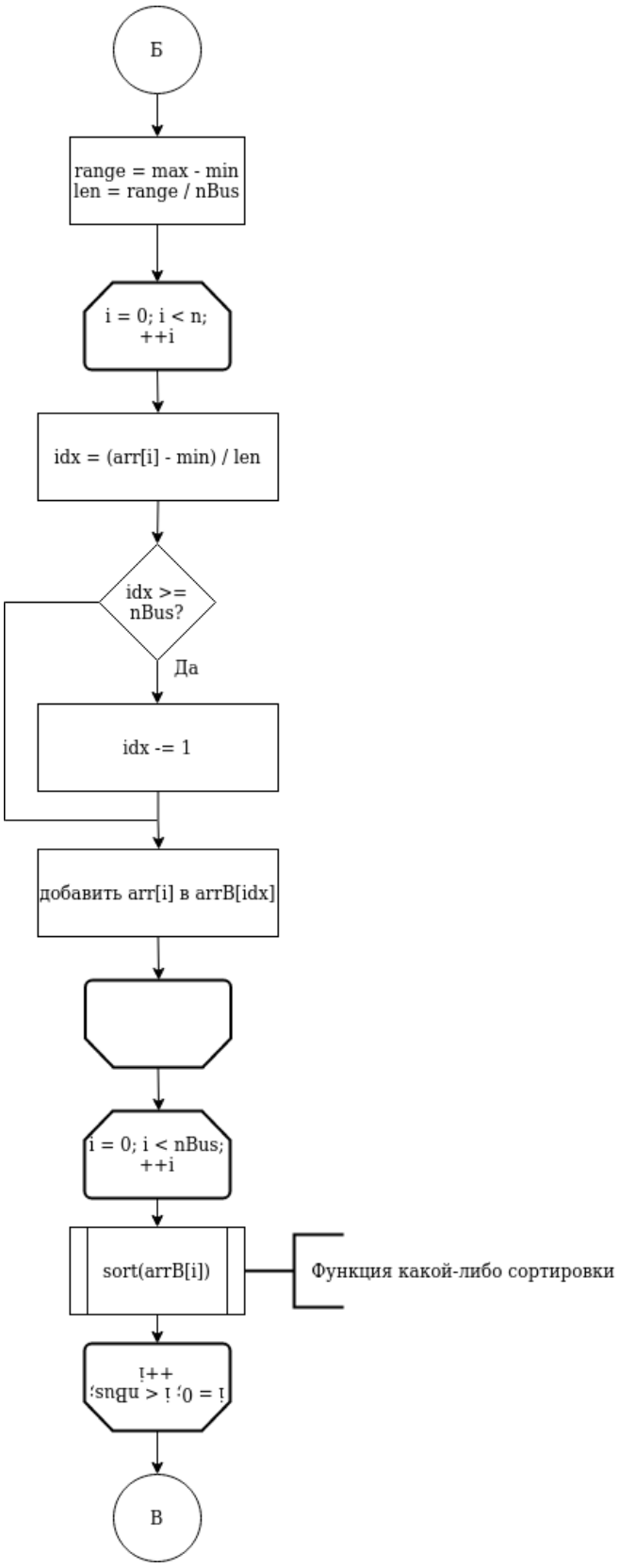


Рис. 2.2: Схема карманной сортировки ч.2

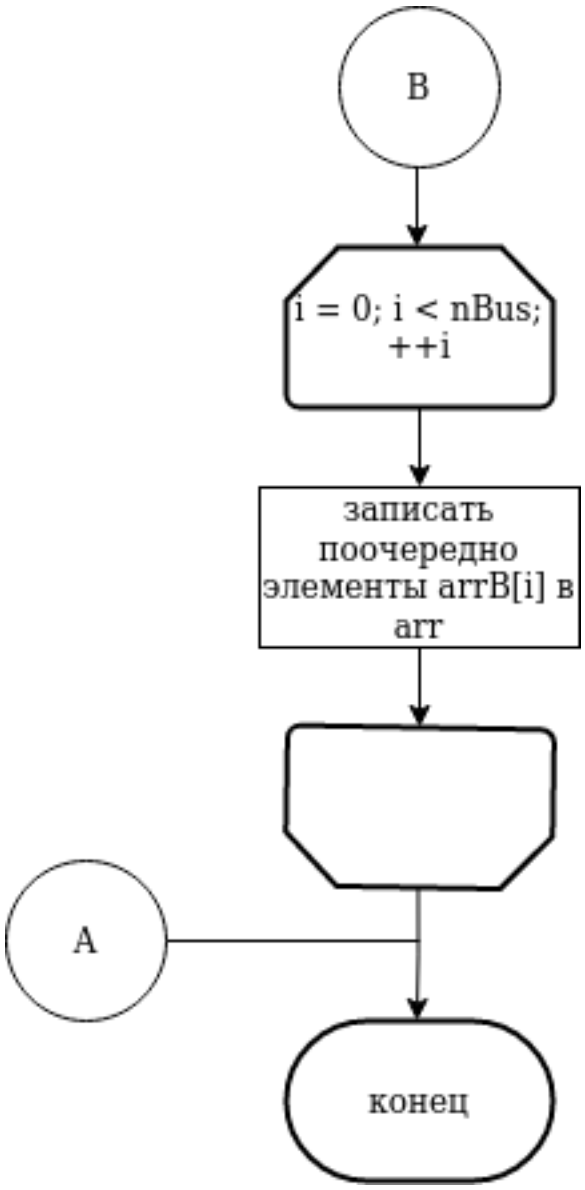


Рис. 2.3: Схема карманной сортировки ч.3

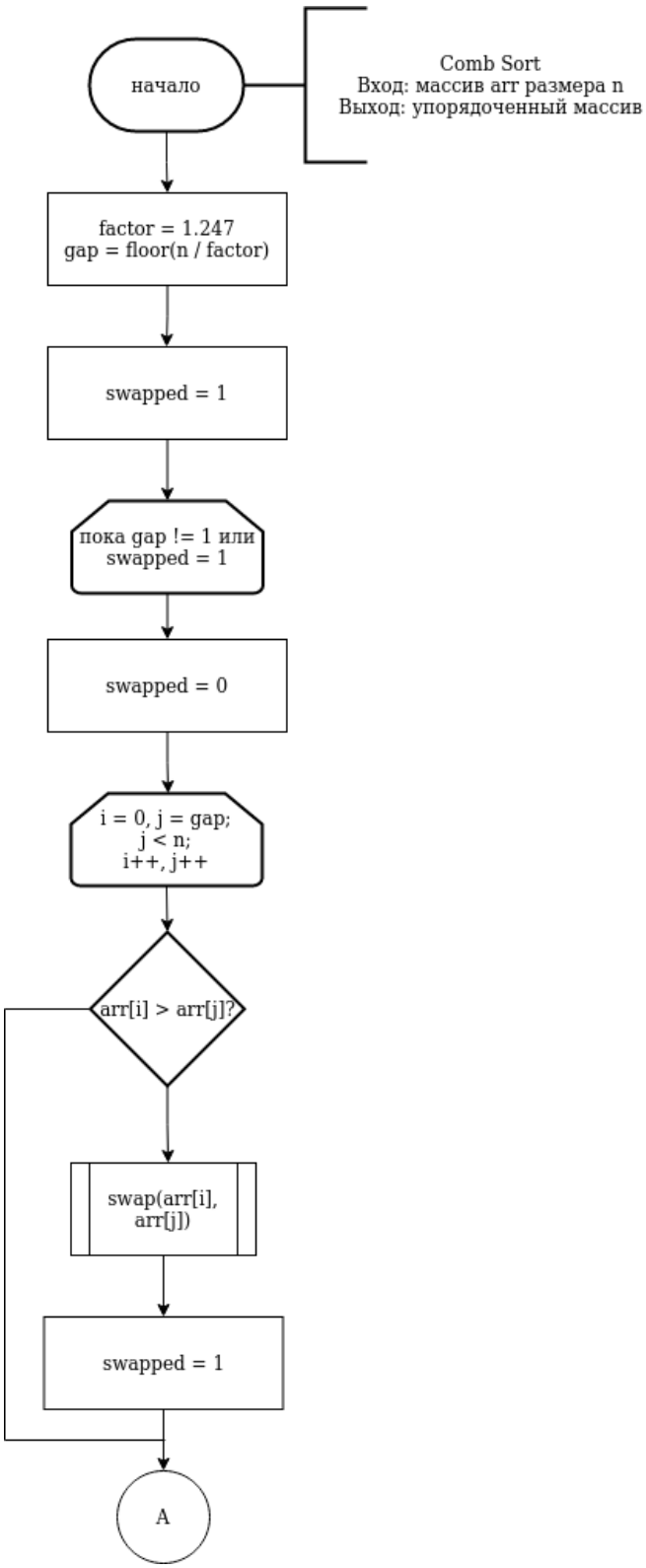


Рис. 2.4: Схема сортировки расческой ч.1

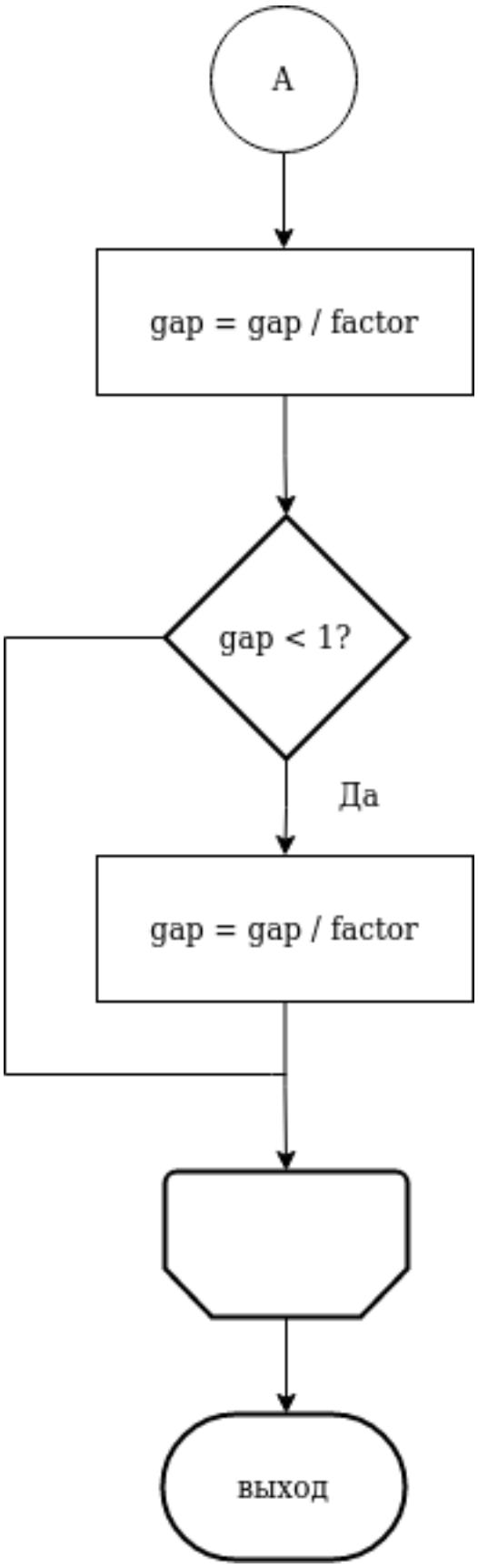


Рис. 2.5: Схема сортировки расческой ч.2

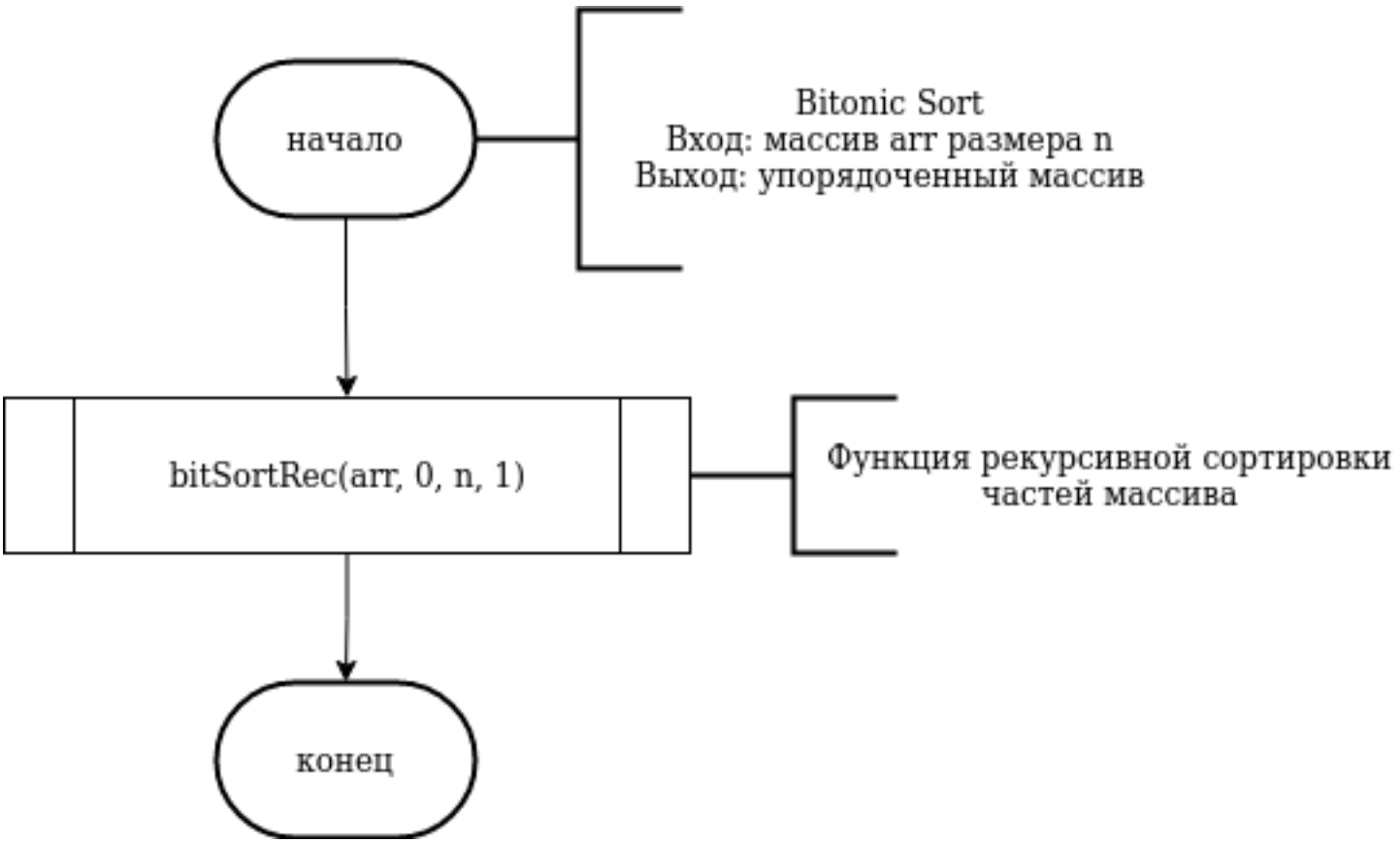


Рис. 2.6: Схема основной функции алгоритма битонной сортировки

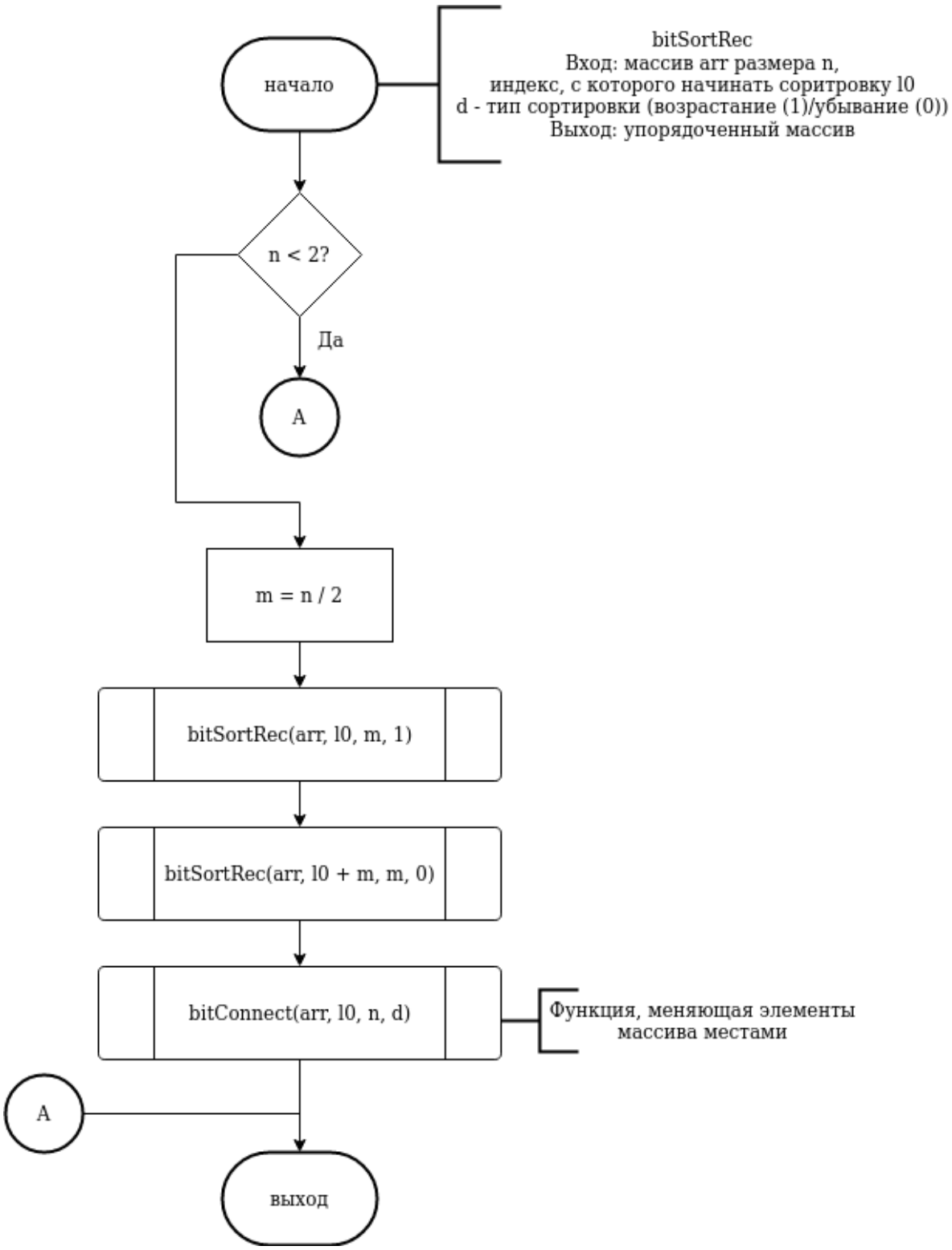


Рис. 2.7: Схема процедуры bitSortRec алгоритма битонной сортировки

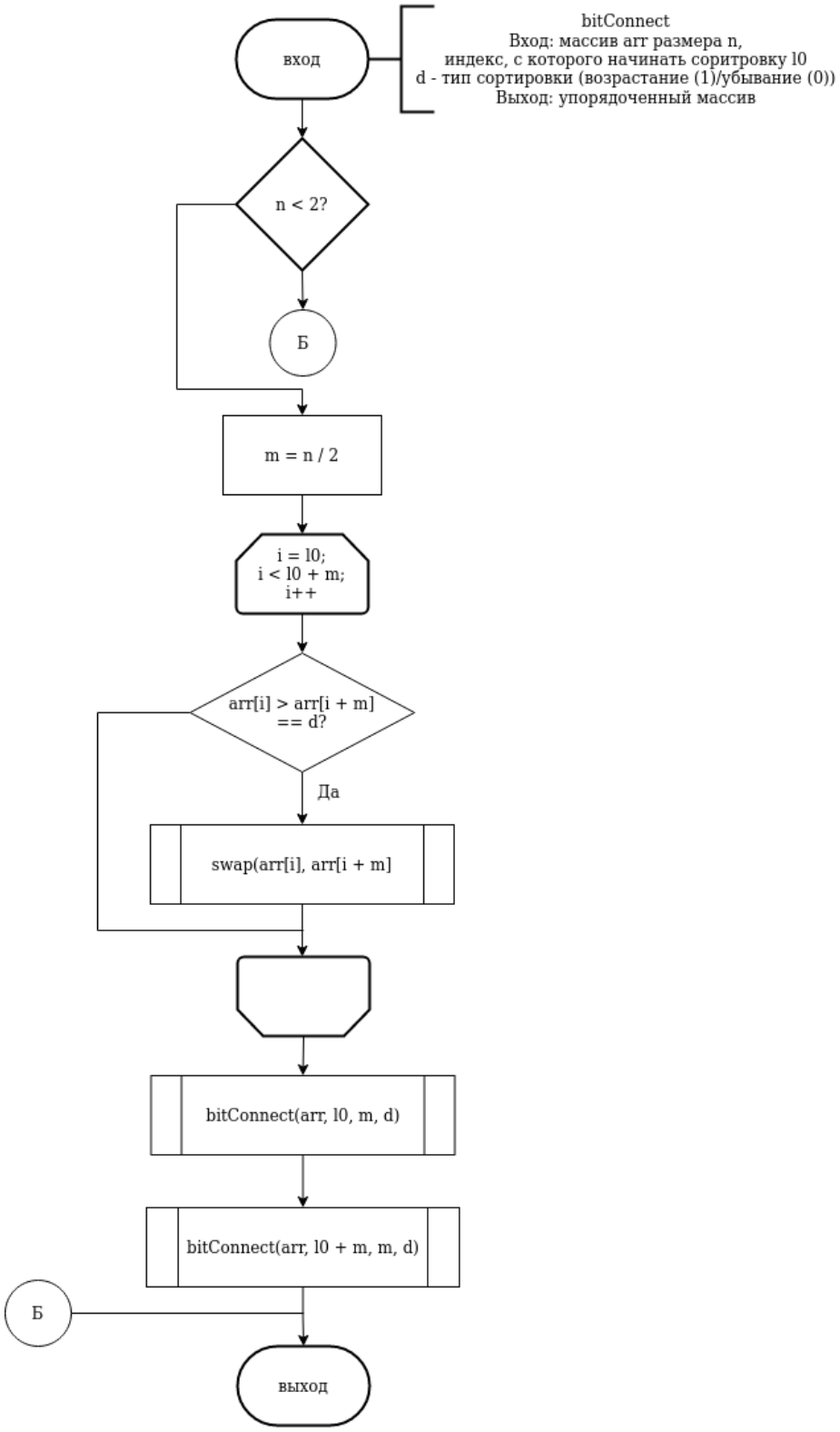


Рис. 2.8: Схема процедуры bitConnect алгоритма битонной сортировки

2.4 Вывод

В данном разделе на основе теоретических данных, полученных в аналитическом разделе были разработаны схемы алгоримов трех рассматриваемых алгоритмов сортировок.

Глава 3

Технологическая часть

В данном разделе преведены требования к ПО, обоснования выбора языка программирования, среды разработки, приведен способ замера времени выполнения, а так же приведены листинги кода.

3.1 Требования к ПО

В программе должна присутсвовать возможность :

1. указания размера входного массива, а так же его элементов;
2. сортировки входного массива одним из трех рассматриваемых способов сортировок;
3. проверки временных характеристик на разных типах входных массивов (упорядочен по возрастанию/убыванию, заполнен случайными числами), для рассматриваемых способов сортировок.

3.2 Выбор языка программирования и среды разработки

Для реализации трех алгоритмов сортировок был выбор язык C, так как я уже знаком с данным языком и поставленная задача будет решена максимально быстро.

Средой разработки был выбран CLion. Данный выбор обусловлен тем, что данная среда на сегодняшний момент является по-моему мнению самой удобной IDE для разработки под C/C++, в силу того, что предоставляет мощные инструменты для отладки кода и его быстрого написания.

3.3 Выбор библиотеки и способа для замера времени

Для замера времени выполнения сортировок использовалась стандартная функция библиотеки <time.h> языка C — clock(), которая замеряет процессорное время. Если измерить время перед началом выполнения алгоритма, и после его окончания, то можно получить время выполнения функции. Реализация данной функции привидена в списке литературы[1].

Посколько все процессорное время не отдается какоЙ-либо одной задаче (явление вытеснения процессов из ядра, квантование процессорного времени), то усреднить результаты

вычислений: замерить совокупное время выполнения реализации алгоритма N раз и вычислить среднее время выполнения.

3.4 Листинги кода

В листингах 3.1,3.2,3.3 приведены листинги алгоритмов карманной, битонной сортировок и сортировки расческой соответсвенно.

Листинг 3.1: Листинг карманной сортировки

```
    void pushBackBucket(bucket *b, sortElem *e)
    {
        b->arr[b->len] = *e;
        (b->len)++;
    }

    void bucketSort(sortElem *arr, size_t n)
    {
        if (n < 2)
      return clock() - timeStart;

        sortElem min = { INF, DEFAULT_C, DEFAULT_D };
        sortElem max = { -INF, DEFAULT_C, DEFAULT_D };
        for (size_t i = 0; i < n; ++i) {
      if (comparatorIncrease(&max, arr + i))
          max = arr[i];
      if (comparatorIncrease(arr + i, &min))
          min = arr[i];
        }

        double range = (max.val) - min.val;
        double lengthBucket = range / nBus;

        for (size_t i = 0; i < n; ++i) {
      size_t idxBucket = (int)((arr[i].val - min.val) / lengthBucket);
      idxBucket = idxBucket >= nBus ? idxBucket - 1 : idxBucket;
      pushBackBucket(arrB + idxBucket, arr + i);
        }

        for (size_t i = 0; i < nBus; ++i)
      qsort(arrB[i].arr, arrB[i].len, sizeof(sortElem), comparatorIncrease);

        size_t idx = 0;
        for (size_t i = 0; i < nBus; ++i)
      for (size_t j = 0; j < arrB[i].len; ++j)
          arr[idx++] = arrB[i].arr[j];
    }
```

Листинг 3.2: Листинг битонной сортировки

```
    void compare(sortElem *arr, size_t i, size_t j, int d)
    {
        if (d == comparatorIncrease(arr + i , arr + j))
        {
      sortElem tmp = arr[i];
      arr[i] = arr[j];
      arr[j] = tmp;
        }
    }

    void bitonicConnect(sortElem* arr, size_t lo, size_t n, int d)
    {
        if (n < 2)
      return;

        size_t m = n / 2;
        for (size_t i = lo; i < lo + m; ++i)
        {
      compare(arr, i, i + m, d);
        }
        bitonicConnect(arr, lo, m, d);
        bitonicConnect(arr, lo + m, m, d);
    }

    void bitonicSortRecursive(sortElem *arr, size_t lo, size_t n, int dir)
    {
        if (n < 2)
      return;

        size_t m = n / 2;
        bitonicSortRecursive(arr, lo, m, INC);
        bitonicSortRecursive(arr, lo + m, m, DEC);
        bitonicConnect(arr,lo, n, dir);
    }

    void bitonicSort(sortElem *arr, size_t n)
    {
        bitonicSortRecursive(arr, 0, n, INC);
    }
```

Листинг 3.3: Листинг сортировки расческой

```
    void combSort(sortElem *arr, size_t n)
    {
        float factor = 1.247f;
        size_t gap = (size_t)(n / factor);
        int swapped = 1;
        while (gap != 1 || swapped) {
      swapped = 0;
      for(size_t i = 0, j = gap; j < n; ++i, ++j)
```

```
9         {
10            if (comparatorIncrease(arr + i, arr + j)) {
11          sortElem tmp = arr[i];
12          arr[i] = arr[j];
13          arr[j] = tmp;
14          swapped = 1;
15            }
16        }
17        gap = gap / factor < 1 ? 1 : gap / factor;
18          }
19      }
```

3.5 Тестирование алгоритмов

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

Входной массив	Результат	Ожидаемый результат
$[10, 20, 30, 40, 50]$	$[10, 20, 30, 40, 50]$	$[10, 20, 30, 40, 50]$
$[50, 40, 30, 20, 10]$	$[10, 20, 30, 40, 50]$	$[10, 20, 30, 40, 50]$
$[-15, -21, -37, -24, -54]$	$[-54, -37, -24, -21, -15]$	$[-54, -37, -24, -21, -15]$
$[4]$	$[4]$	$[4]$
Пустой массив	Пустой массив	Пустой массив

Таблица 3.1: Тестирование функций

3.6 Вывод

В данном разделе были разработаны исходные коды трёх алгоритмов сортировок: расческой, битонной и карманной.

Глава 4

Экспериментальная часть

4.1 Технические характеристики

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

1. Операционная система: Windows-10, 64-bit;
2. Оперативная память: 16 GB;
3. Процессор: Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz, 1992 МГц, 4 ядра, 8 логических процессоров.

4.2 Время выполнения алгоритмов

Таблица 4.1: Таблица времени выполнения сортировок на данных, отсоритрованных по возрастанию (в мс)

Размер	comb	bucket	bitonic
100	7.8	7.61	27.3
1000	113	100.4	428.1
10000	1750	801.57	12427.6

График зависимости времени сортировок от кол-ва элементов на упорядоченном массиве

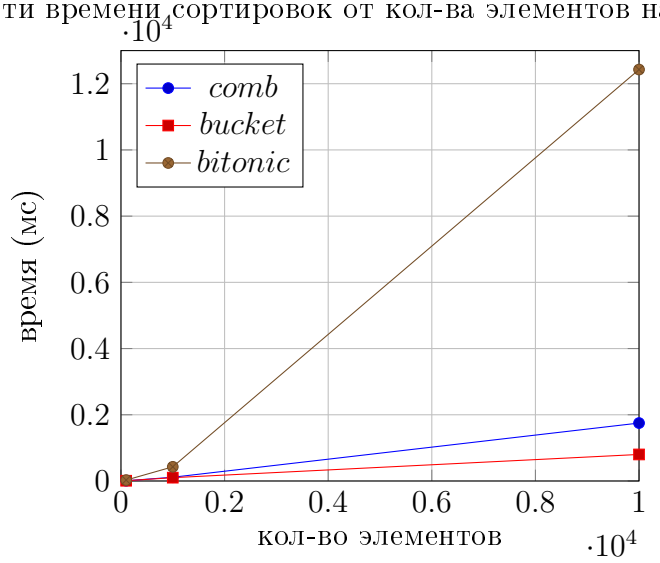


Таблица 4.2: Таблица времени выполнения сортировок на данных, отсоритрованных по убыванию (в мс)

Размер	comb	bucket	bitonic
100	9.6	7.71	28.49
1000	130.27	96.4	390.85
10000	2066.09	888.5	11475.05

График зависимости времени сортировок от кол-ва элементов на обратно упорядоченном массиве

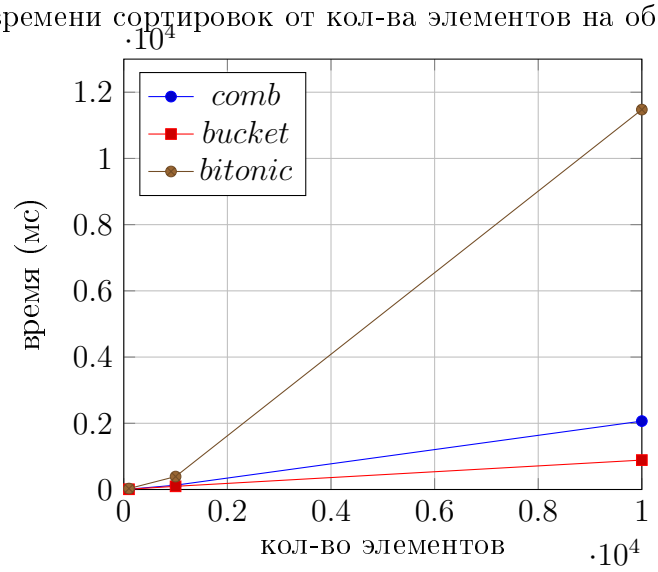


Таблица 4.3: Таблица времени выполнения сортировок на случайных данных (от 0 до 1000) (в мс)

Размер	comb	bucket	bitonic
100	12.38	12.78	39.79
1000	189.44	189.77	486.02
10000	2491.19	1975.79	12273.07

График зависимости времени сортировок от кол-ва элементов на массиве случайных чисел

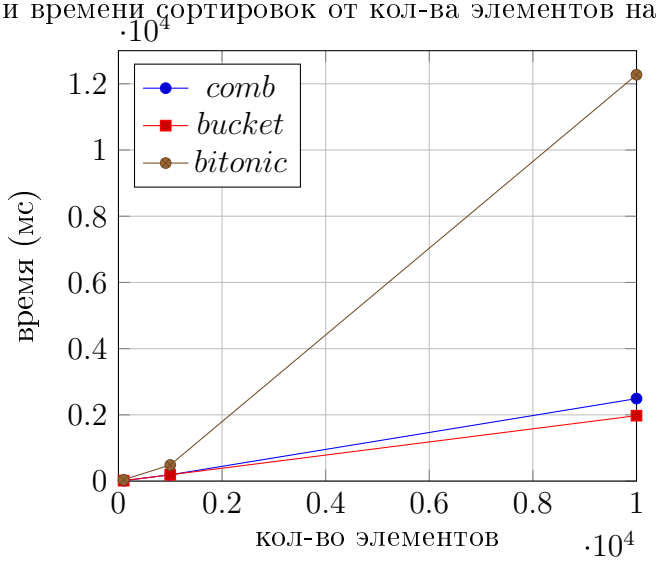
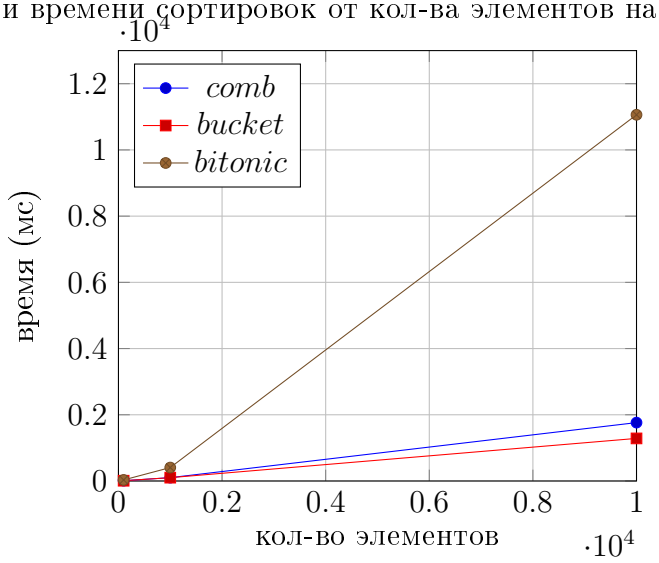


Таблица 4.4: Таблица времени выполнения сортировок на случайных данных (от 0 до 3) (в мс)

Размер	comb	bucket	bitonic
100	8.89	8.08	31.94
1000	100.6	100.66	405.61
10000	1763.35	1286.04	11061.61

График зависимости времени сортировок от кол-ва элементов на массиве случайных чисел



4.3 Вывод

Алгоритм сортировки вставками работает лучше сортировок выбором и пуззырьком на случайных и уже отсортированных числах. Сортировка вставками массива из 10000 элементов в 3.3 раза быстрее сортировки пузырьком и в 1.6 раз ыстрее сортировки выбором. Следовательно, алгоритм сортировки вставками работает быстрее, чем алгоритм сортировки пузырьком и выбором на больших массивах данных.

Заключение

В рамках данной лабораторной работы:

1. были изучены и рассмотрены алгоритмы сортировок пузырьком, вставками и выбором;
2. были построены блок-схемы выбранных алгоритмов;
3. был реализован каждый из алгоритмов;
4. была рассчитана их трудоемкость;
5. были эксперитментально оценены временные характеристики алгоритмов;
6. были сделаны выводы на основании проделанной работы

На основании анализа трудоемкости алгоритмов в выбранной модели вычислений, было показано, что алгоритм сортировки вставками имеет наименьшую сложность в уже отсортированном массиве, а так же эффективнее работает на массиве размером 10000, обгоняя по скорости алгоритмы пузырьком и выбором.