



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	3
1 Аналитическая часть	4
1.1 Цель и задачи	4
1.2 Сортировка расческой	4
1.3 Карманная сортировка	5
1.4 Битонная сортировка	5
2 Конструкторская часть	6
2.1 Модель вычислений	6
2.2 Трудоемкость алгоритмов	6
2.2.1 Алгоритм сортировки расческой	6
2.2.2 Алгоритм карманной сортировки	7
2.2.3 Алгоритм битонной сортировки	8
2.3 Описания алгоритмов	8
3 Технологическая часть	17
3.1 Требования к ПО	17
3.2 Выбор языка программирования и среды разработки	17
3.3 Выбор библиотеки и способа для замера времени	17
3.4 Реализации алгоритмов	18
3.5 Тестирование алгоритмов	20
4 Экспериментальная часть	21
4.1 Технические характеристики	21
4.2 Замеры времени	21
Список использованных источников	26

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) количество 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 сортировок, их особенностей и областей применения, данная работа была посвящена анализу некоторых из этих алгоритмов.

Глава 1

Аналитическая часть

1.1 Цель и задачи

Целью данной работой является исследование алгоритмов сортировок расческой, карманной и битонной. Для этого в ходе исследования алгоритмов требуется решить следующие задачи:

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

1.2 Сортировка расческой

Сортировка расческой (на англ. Comb sort) является улучшенным вариантом сортировки пузырьком. В сортировке пузырьком сравниваются и переставляются соседние элементы, то есть разница индексов этих элементов всегда 1. В сортировке же расческой, 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 сортируется рекурсивно карманной сортировкой. После этого элементы из полученных блоков по порядку помещаются обратно в массив, в результате чего получается отсортированные 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 не убывают, а потом с определенного элемента не возрастают.

На первом этапе сортировки 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, все элементы в первой половине меньше, чем все элементы во второй половине. Результаты сравнения и обмена между двумя последовательностями длиной $N/2$ каждая, где N - количество элементов в последовательности. Далее необходимо повторять процесс обмена рекурсивно для каждой половины последовательности, пока не будет получена одна отсортированная последовательность длины N .

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.

$$+, -, /, \%, ==, !=, <, >, <=, >=, [], ++, -- \quad (2.1)$$

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

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

3. Трудоемкость вызова функции равна 0.
4. Трудоемкость цикла рассчитывается, как (2.3).

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

2.2 Трудоемкость 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 количество перебираемых элементов на i -м шаге внешнего цикла как N_i . Тогда:

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

gap_i на i -м шаге вычисляется как:

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

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, 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 количество проходов внутреннего цикла, на i -й итерации можно выразить формулой:

$$N_i = \frac{N}{1.247^i} \quad (2.6)$$

А количество всех проходов выражается как:

$$S_N = N \sum_{r=1}^n \frac{1}{1.247^r} \quad (2.7)$$

Исходя из вида (2.7) выражение под знаком суммы будет иметь асимптотику $O(\log(N))$. Тогда асимптотика всей суммы будет $O(N \log(N))$, что будет являться сложностью алгоритма в лучшем случае.

2.2.2 Алгоритм карманной сортировки

Обозначим количество карманов за k . Алгоритм состоит из четырех последовательно идущих циклов:

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

Для сортировки корзин в шаге 3 был выбран быстрый алгоритм сортировки.

Цикл поиска максимума и минимума среди всех элементов работает за:

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

Цикл распределения элементов массива по соответствующим корзинам работает за:

$$f_2 = 1 + 1 + N \cdot (12 + 1 + 1) = 2 + 14N = O(N) \quad (2.9)$$

Цикл сортировки элементов каждой корзины алгоритмом быстрой сортировки в **лучшем** случае (элементы распределены по корзинам равномерно, асимптотика их просмотра – $O(k)$, входной массив расположен так, что внутренняя сортировка работает за лучшее время – $O(N)$) имеет асимптотику:

$$f_3 = O(N + k) \quad (2.10)$$

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

$$f_3 = O(N^2) \quad (2.11)$$

Так как все N элементов распределены равномерно по k корзинам, то цикл соединения всех корзин воедино работает со следующей асимптотикой:

$$f_4 = O(N + k) \quad (2.12)$$

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

Сложность алгоритма битонной сортировки как в **лучшем**, так и в **худшем** случае равна $O(\log^2(N)[1])$.

2.3 Описания алгоритмов

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

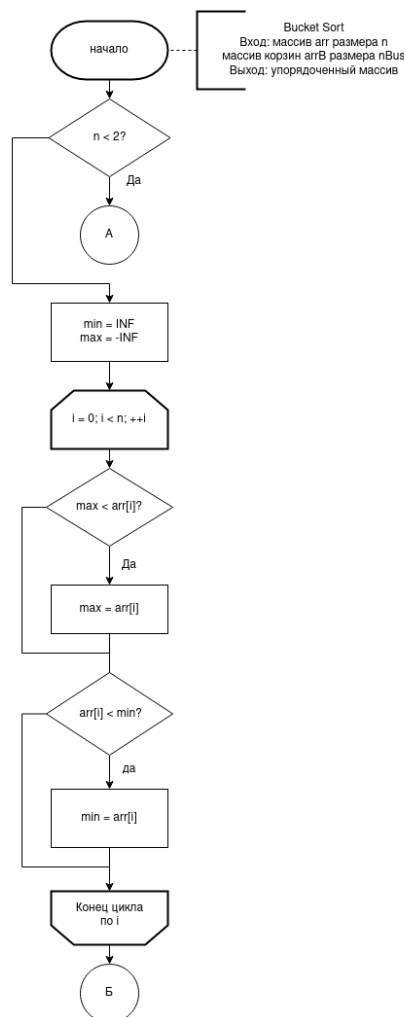


Рис. 2.1. Схема карманной сортировки, ч.1

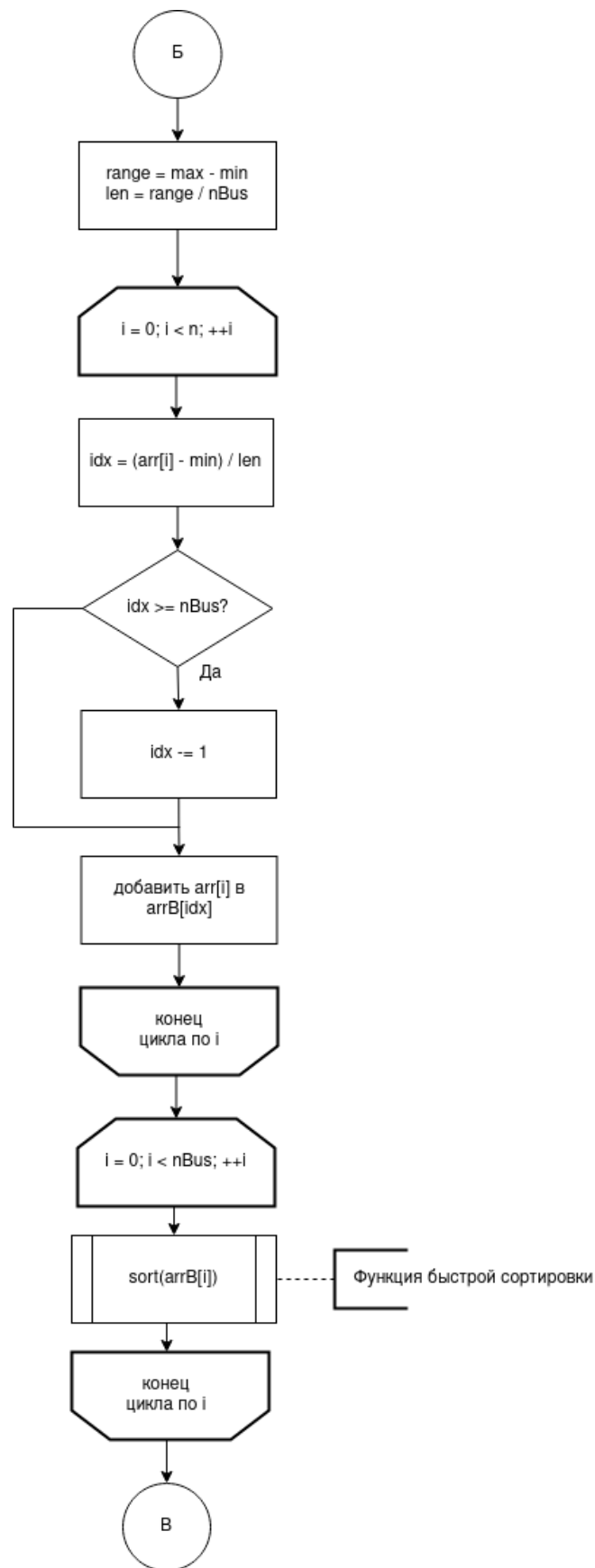


Рис. 2.2. Схема карманной сортировки, ч.2

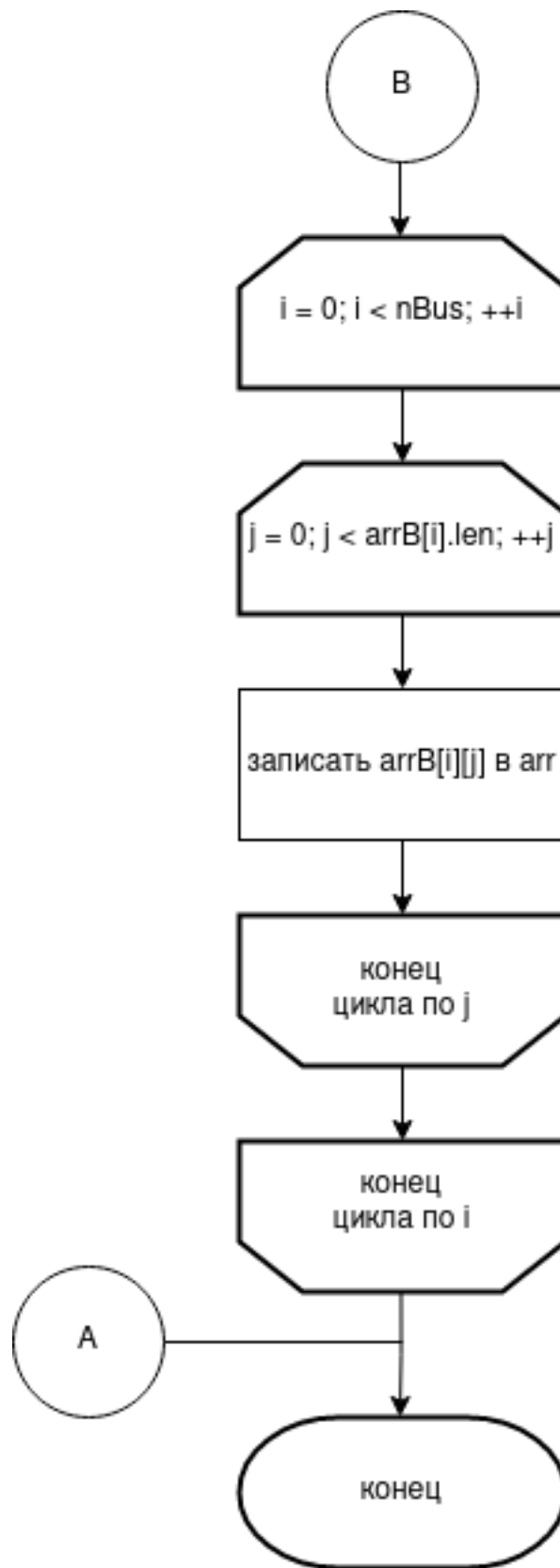


Рис. 2.3. Схема карманной сортировки, ч.3

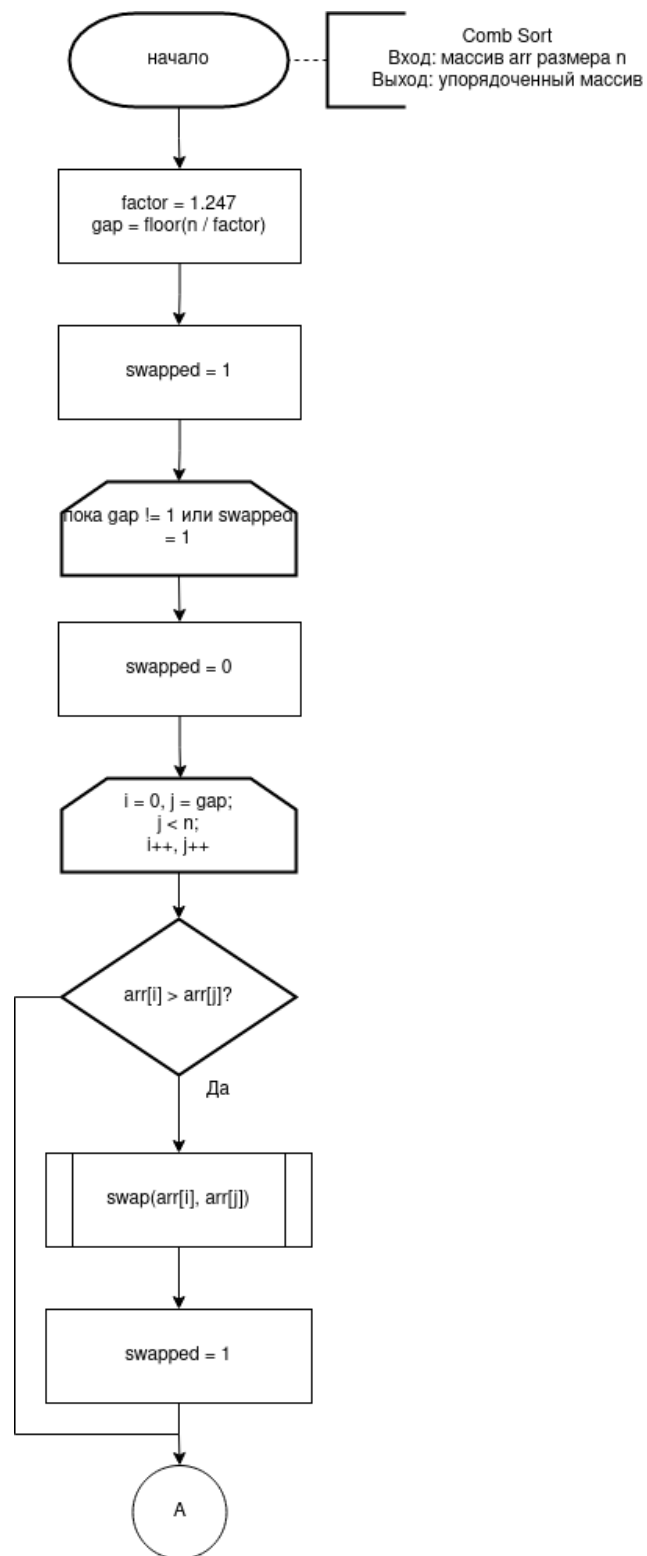


Рис. 2.4. Схема сортировки расческой, ч.1

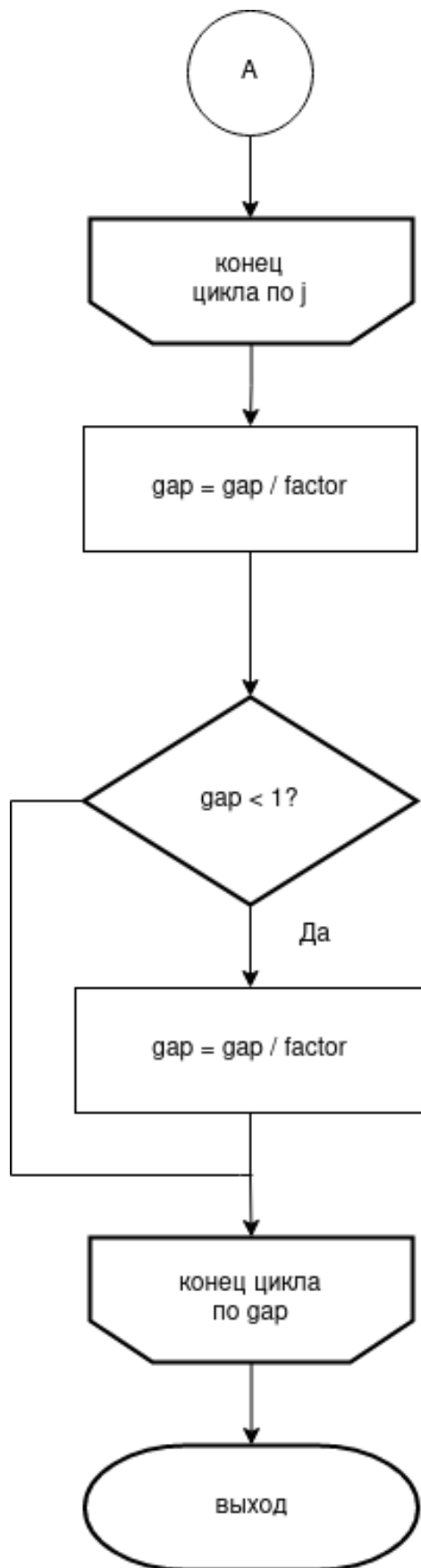


Рис. 2.5. Схема сортировки расческой, ч.2

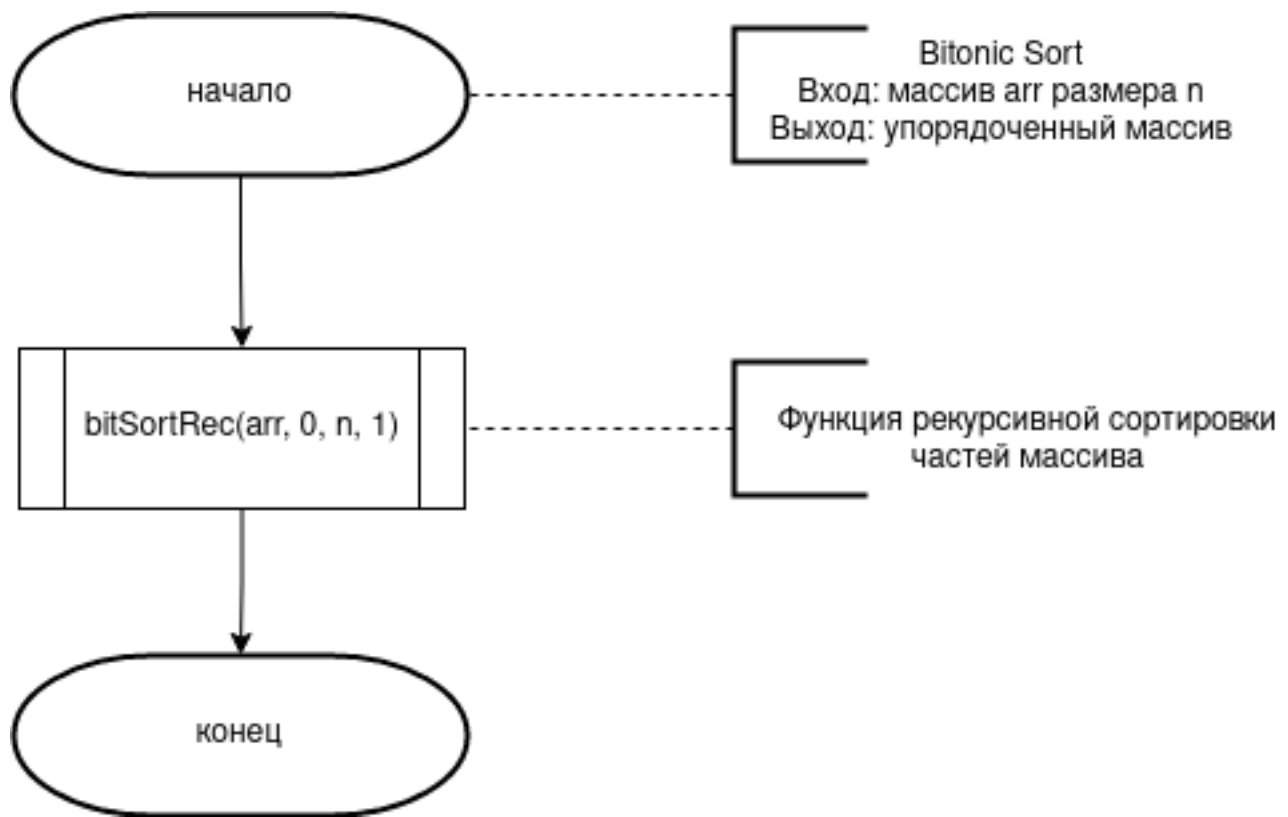


Рис. 2.6. Схема основной функции алгоритма битонной сортировки

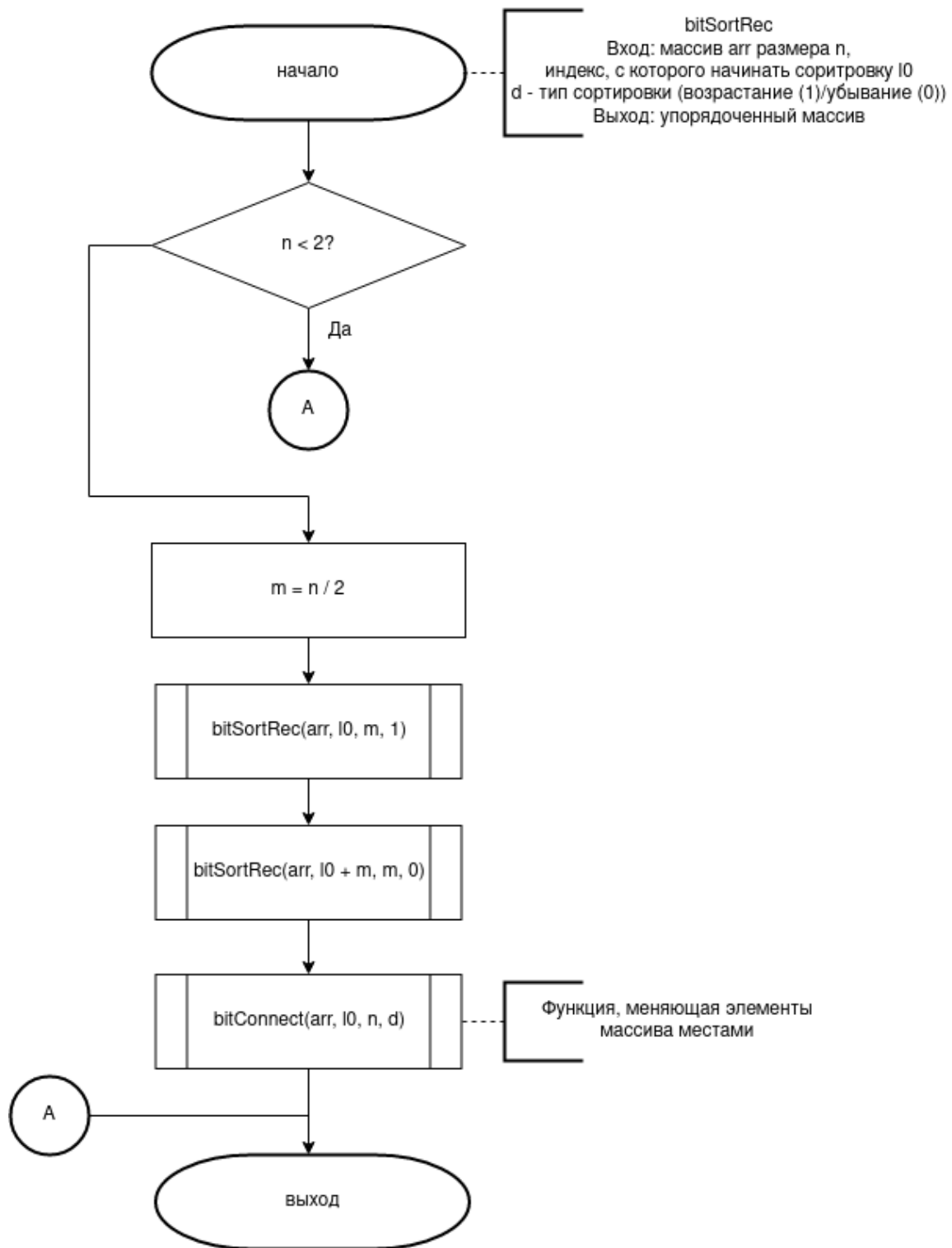


Рис. 2.7. Схема процедуры bitSortRec алгоритма битонной сортировки

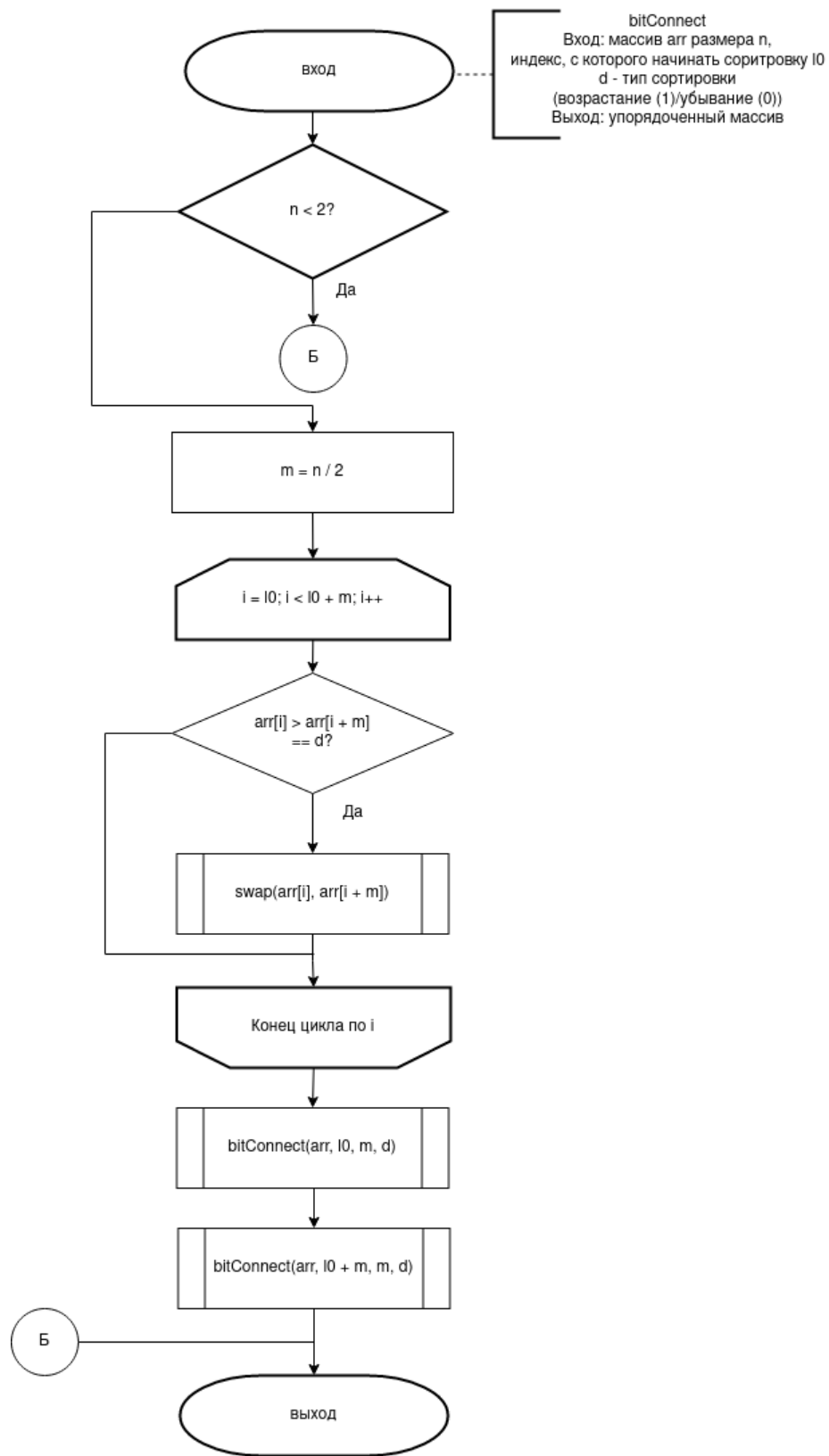


Рис. 2.8. Схема процедуры bitConnect алгоритма битонной сортировки

Вывод

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

Глава 3

Технологическая часть

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

3.1 Требования к ПО

В программе должна присутствовать возможность:

- 1) указания размера входного массива, а также его элементов;
- 2) сортировки входного массива одним из трех рассматриваемых способов сортировок;
- 3) замер процессорного времени выполнения реализаций алгоритмов сортировки.

3.2 Выбор языка программирования и среды разработки

Для реализации трех алгоритмов сортировок был выбран язык C, так как я уже имею опыт разработки на данном языке программирования.

Средой разработки был выбран CLion. Данный выбор обусловлен тем, что данная среда предоставляет возможность разработки приложений под c/c++ и имеет инструменты для отладки кода.

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 приведена в [1].

Поскольку все процессорное время не отдается какой-либо одной задаче (в связи с явлением вытеснения процессов из ядра, квантование процессорного времени), то требуется усреднить результаты вычислений: замерить совокупное время выполнения реализации алгоритма N раз и вычислить среднее время выполнения.

3.4 Реализации алгоритмов

В листингах 3.1,3.2,3.3 приведены реализации алгоритмов карманной, битонной сортировок и сортировки расческой соответственно.

Листинг 3.1. Листинг карманной сортировки

```
1  void pushBackBucket(bucket *b, sortElem *e)
2  {
3      b->arr[b->len] = *e;
4      (b->len)++;
5  }
6
7  void bucketSort(sortElem *arr, size_t n)
8  {
9      if (n < 2)
10     return clock() - timeStart;
11
12     sortElem min = { INF, DEFAULT_C, DEFAULT_D };
13     sortElem max = { -INF, DEFAULT_C, DEFAULT_D };
14     for (size_t i = 0; i < n; ++i) {
15         if (comparatorIncrease(&max, arr + i))
16             max = arr[i];
17         if (comparatorIncrease(arr + i, &min))
18             min = arr[i];
19     }
20
21     double range = (max.val) - min.val;
22     double lengthBucket = range / nBus;
23
24     for (size_t i = 0; i < n; ++i) {
25         size_t idxBucket = (int)((arr[i].val - min.val) / lengthBucket);
26         idxBucket = idxBucket >= nBus ? idxBucket - 1 : idxBucket;
27         pushBackBucket(arrB + idxBucket, arr + i);
28     }
29
30     for (size_t i = 0; i < nBus; ++i)
31         qsort(arrB[i].arr, arrB[i].len, sizeof(sortElem), comparatorIncrease);
32
33     size_t idx = 0;
34     for (size_t i = 0; i < nBus; ++i)
35         for (size_t j = 0; j < arrB[i].len; ++j)
36             arr[idx++] = arrB[i].arr[j];
37 }
```

Листинг 3.2. Листинг битонной сортировки

```
1  void compare(sortElem *arr, size_t i, size_t j, int d)
2  {
3      if (d == comparatorIncrease(arr + i, arr + j))
4      {
```

```

5      sortElem tmp = arr[i];
6      arr[i] = arr[j];
7      arr[j] = tmp;
8      }
9  }
10
11 void bitonicConnect(sortElem* arr, size_t lo, size_t n, int d)
12 {
13     if (n < 2)
14         return;
15
16     size_t m = n / 2;
17     for (size_t i = lo; i < lo + m; ++i)
18     {
19         compare(arr, i, i + m, d);
20     }
21     bitonicConnect(arr, lo, m, d);
22     bitonicConnect(arr, lo + m, m, d);
23 }
24
25 void bitonicSortRecursive(sortElem *arr, size_t lo, size_t n, int dir)
26 {
27     if (n < 2)
28         return;
29
30     size_t m = n / 2;
31     bitonicSortRecursive(arr, lo, m, INC);
32     bitonicSortRecursive(arr, lo + m, m, DEC);
33     bitonicConnect(arr, lo, n, dir);
34 }
35
36 void bitonicSort(sortElem *arr, size_t n)
37 {
38     bitonicSortRecursive(arr, 0, n, INC);
39 }

```

Листинг 3.3. Листинг сортировки расческой

```

1 void combSort(sortElem *arr, size_t n)
2 {
3     float factor = 1.247f;
4     size_t gap = (size_t)(n / factor);
5     int swapped = 1;
6     while (gap != 1 || swapped) {
7         swapped = 0;
8         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.

Таблица 3.1. : Тестирование функций

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

Вывод

В данном разделе были разработаны исходные коды трёх алгоритмов сортировок: расчётной, 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 ГБ;
- 3) процессор Intel(R) Core(TM) i7-8565U CPU @ 1.80 ГГц, 4 ядра, 8 логических процессоров.

4.2 Замеры времени

Введем обозначения: comb – алгоритм сортировки расческой, bucket - алгоритм карманной сортировки, bitonic - алгоритм битонной сортировки.

В таблице 4.1 приведены результаты замеров времени для лучшего случая работы алгоритмов.

Таблица 4.1. Таблица замера времени выполнения сортировок на данных, отсортированных 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

Зависимость времени от количества элементов для лучшего случая работы алгоритмов представлена на рис. 4.1.

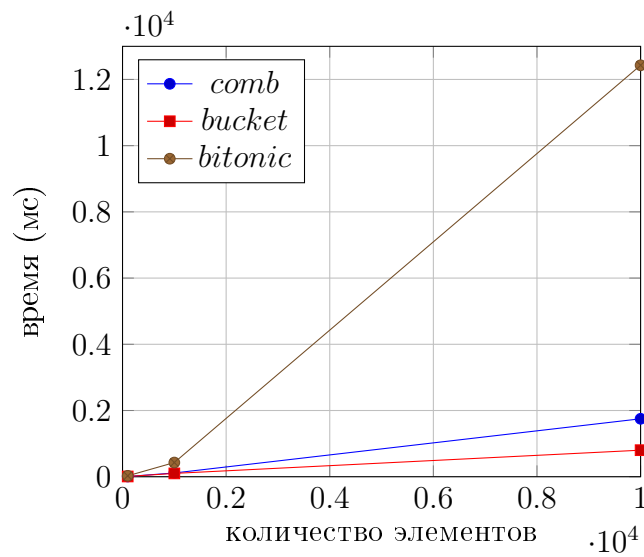


Рис. 4.1. Зависимость времени от количества элементов

В таблице 4.2 приведены результаты замеров времени для худшего случая работы алгоритмов:

Таблица 4.2. Таблица замера времени выполнения сортировок на данных, отсортированных 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

Зависимость времени от количества элементов для худшего случая работы алгоритмов представлена на рис. 4.2.

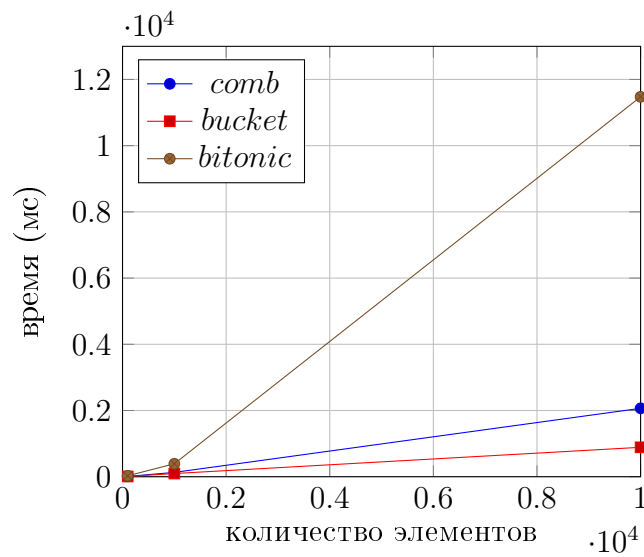


Рис. 4.2. Зависимость времени от количества элементов

В таблице 4.3 приведены результаты замеров времени работы алгоритмов на случайно заполненном массиве:

Таблица 4.3. Таблица замера 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

Зависимость времени от количества элементов для работы алгоритмов на случайно заполненном массиве представлена на рис. 4.3.

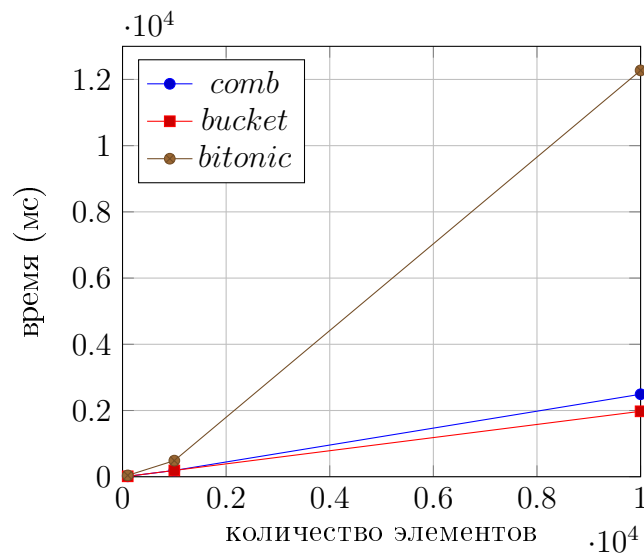


Рис. 4.3. Зависимость времени от количества элементов

В таблице 4.4 приведены результаты замеров времени работы алгоритмов на случайно заполненном массиве:

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

Зависимость времени от количества элементов для работы алгоритмов на случайно заполненном массиве представлена на рис. 4.4.

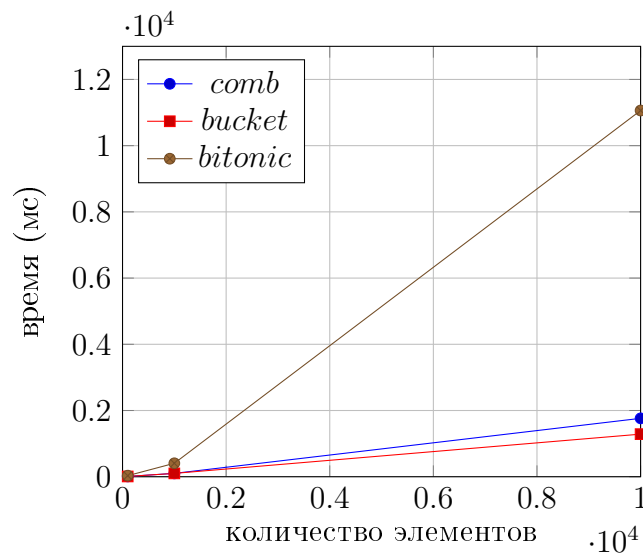


Рис. 4.4. Зависимость времени от количества элементов

Вывод

Результаты замеров показали, что алгоритм карманной сортировки, который сортирует корзины быстрой сортировкой, работает быстрее на случайно отсортированном массиве с элементами от 0 до 1000, чем сортировка битонная сортировка (так как массив из 10000 элементов дополняется до 2^N) и сортировка расческой (так как быстрая сортировка в модификации может сортировать элементы за $O(N)$ при делении массива на 3 части). При этом карманная сортировка работает быстрее на предварительно отсортированных массивах, чем на случайным образом заполненных массивах, (при условии, что количество корзин берется как $0.5 \cdot N$), так как в этом случае меньше элементов попадает в одну и ту же корзину.

Заключение

Цель работы достигнута: выполнено исследование алгоритмов сортировок расческой, карманной и битонной. Все задачи решены:

- 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 экспериментально получены временные характеристики алгоритмов;
- 6) были сделаны выводы на основании проделанной работы

На основании анализа трудоемкости алгоритмов в выбранной модели вычислений, было показано, что алгоритм карманной сортировки с количеством выбираемых карманов $0.5 \cdot N$ и алгоритмом быстрой сортировки карманов эффективнее работает на отсортированных по возрастанию/убыванию массивах (нежели на случайно отсортированных), обгоняя по скорости алгоритмы битонной сортировки и сортировки расческой.

Список использованных источников

- [1] Lacey S., Box R. A Fast, Easy Sort: A novel enhancement makes a bubble sort into one of the fastest sorting routines / Журнал, Byte. apr. 1991., vol. 16, № 4. — Canada: Ottawa, 1991. — Pp. 315—320.
- [2] Laxmikant V. Kale, Edgar Solomonik. Sorting// Encyclopedia of Parallel Computing : encyclopedia. — Berlin: Springer, 2011. — Pp. 1855—1861.
- [3] М. В. Ульянов Ресурсно-эффективные компьютерные алгоритмы. Разработка и анализ. Учебное пособие / М.В. Ульянов. — М.: Издательство НАУКА ФИЗМАТЛИТ, 2007. - 376 с.
- [4] Классификация алгоритмов сортировки [Электронный ресурс]. Режим доступа: <http://dander.ru/gos/22.html> (дата обращения: 23.09.2022).