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
по дисциплине «Анализ алгоритмов»

Тема: Алгоритмы сортировки
Студент: Карпова Е. О.
Группа: ИУ7-52Б
Оценка (баллы): _____
Преподаватели: Волкова Л. Л., Строганов Ю. В.

Москва — 2022 г.

Оглавление

Введение

В сфере обработки информации существует множество разнообразных задач, для решения которых требуется найти наиболее оптимальный алгоритм. Решение многих подобных задач значительно упрощается при использовании сортировки данных, считающейся наиболее фундаментальной задачей при изучении алгоритмов.

Сортировка — это процесс упорядочения некоторого множества элементов, на котором определены отношения порядка:

- $>$;
- $<$;
- $>=$;
- $<=$.

Алгоритмы сортировки имеют большое практическое применение. Их можно встретить почти везде, где речь идет об обработке и хранении больших объемов информации, например, в обработке баз данных или математических программах.

В данной лабораторной работе алгоритм сортировки будет рассматриваться как алгоритм для упорядочивания элементов в массиве.

На данный момент определено множество алгоритмов сортировки данных в массиве, и они постоянно оптимизируются. Алгоритмы сортировки формируют отдельный класс алгоритмов.

Цель работы: получение навыков программирования, тестирования полученного программного продукта и проведения замеров времени выполнения и потребляемой памяти по результатам работы программы на примере реализации алгоритмов сортировки.

Задачи работы:

1) изучение алгоритмов блинной и быстрой сортировок и сортировки бусинами;
2) разработка схем данных алгоритмов и анализ их трудоёмкости;
3) реализация данных алгоритмов;
4) проведение замеров потребления памяти (в байтах) для данных алгоритмов;

5) проведение замеров времени работы (в нс) данных алгоритмов;

6) получение графической зависимости замеряемых величин от длины массива, предоставляемого на вход алгоритмам;

7) проведение сравнительного анализа данных алгоритмов на основе полученных зависимостей.

1. Аналитическая часть

В данном разделе будут рассмотрены теоретические основы алгоритмов блинной, быстрой и гравитационной сортировок. Все алгоритмы трактуются так, как будто сортировка выполняется по возрастанию.

1.1. Алгоритм блинной сортировки

Блинная сортировка — это класс алгоритмов, в которых допускается единственная операция — переворот элементов последовательности до заданного индекса. Соответственно минимизируемым параметром при оптимизации является количество произведённых переворотов. Это отличает данный класс алгоритмов от других, в которых к минимуму сводится количество сравнений.

Процесс можно визуально представить как стопку блинов, которую тасуют путём взятия нескольких блинов сверху и их переворачивания. На каждой итерации находится максимум текущего массива, и часть массива от начала до максимума включая переворачивается. Так текущий максимум становится первым элементом в массиве. Затем выполняется переворот всего массива так, что максимум оказывается в конце массива. На каждой следующей итерации количество анализируемых элементов массива уменьшается на один, так как максимум уже занял свою позицию — последнюю в массиве.

1.2. Алгоритм быстрой сортировки

Быстрая сортировка является существенно улучшенным вариантом алгоритма сортировки с помощью прямого обмена. Этот алгоритм является одним из самых быстрых известных универсальных алгоритмов сортировки массивов.

Основные этапы алгоритма можно изложить так:

1) выбирается опорный элемент массива (в разных реализациях в качестве опорных выбираются различные элементы — последний, первый, серединный);

2) массив делится на две (элементы больше опорного и элементы меньше опорного, элементы равные опорному добавляются в любую из двух частей) или три части (элементы больше опорного, элементы меньше опорного и элементы равные опорному);

3) первые два шага рекурсивно применяются к выделенным из массива группам элементов, если группа имеет больше одного элемента;

4) отсортированные группы соединяются в один массив в порядке: меньшие опорного, большие опорного, в случае разбиения на две части, и в порядке: меньшие опорного, равные опорному, большие опорного, в случае разбиения на три части.

1.3. Алгоритм сортировки бусинами (гравитационный)

Алгоритм сортировки бусинами (гравитационный алгоритм) — алгоритм сортировки, который может быть применён только к массиву натуральных чисел.

В процессе работы алгоритма будет заполнена матрица, имеющая размерность равную < количество элементов массива > × < значение максимального элемента массива >, нулями или единицами, в соответствии с величиной чисел, являющихся элементами сортируемого массива. Каждая строка такой матрицы будет представлять собой элемент массива с индексом, равным индексу строки в матрице, где сумма записанных подряд с первого элемента строки единиц будет равняться данному элементу массива. Эти единицы называются бусинами.

Алгоритм гравитационной сортировки может быть сравнен с тем, как бусины падают вниз на параллельных шестах (столбцах матрицы), например как в абаке, однако каждый из шестов может иметь разное количество бусин.

После того, как все строки заполнены, все бусины нужно опустить вниз по шестам, как будто под действием гравитации. Тогда последний ряд будет представлять собой самое большое число списка, а первый — наименьшее. В результате суммирования единиц каждой строки будут получены элементы исходного массива в порядке возрастания от первой строки матрицы к последней. Их запись в исходный массив в порядке от первой строки матрицы к последней позволит получить исходный массив, отсортированный по возрастанию.

2. Конструкторская часть

В данном разделе будут представлены схемы реализаций алгоритмов сортировки, среди которых блинный, быстрый и сортировка бусинами.

2.1. Разработка алгоритма блинной сортировки

На рисунке 2.1 представлена схема реализации алгоритма блинной сортировки. На рисунках 2.2 и 2.3 представлены схемы реализаций алгоритмов подпрограмм поиска индекса максимального элемента массива (рисунок 2.2) и переворота массива (рисунок 2.3), необходимых для реализации алгоритма блинной сортировки. Подпрограмма поиска индекса максимального элемента массива также используется в алгоритме сортировки бусинами.

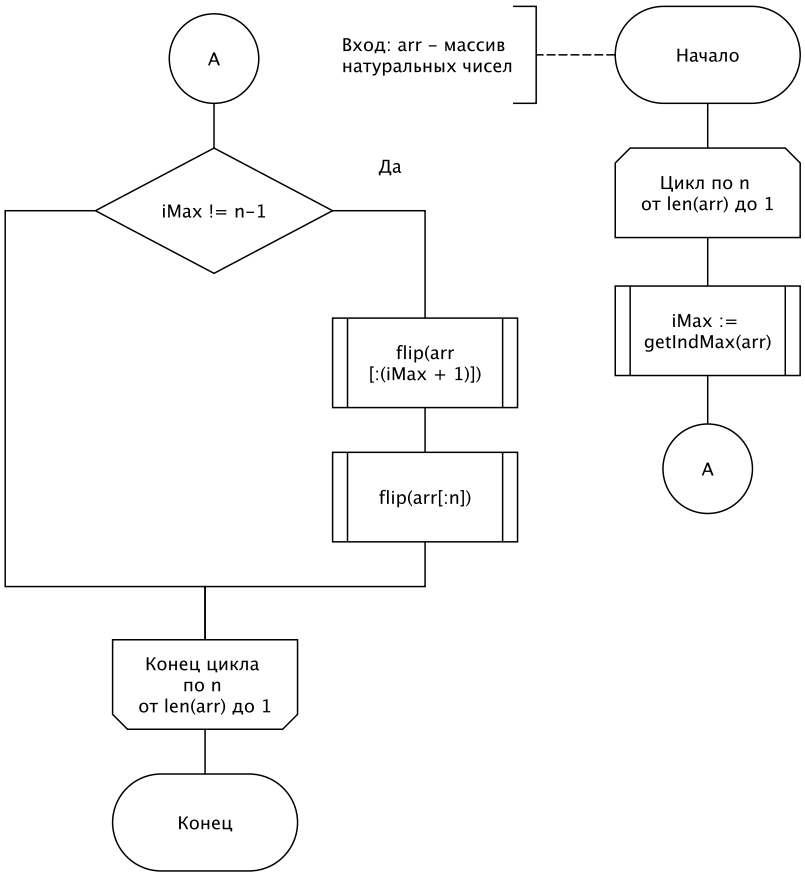

Рисунок 2.1 — Схема реализации алгоритма блинной сортировки

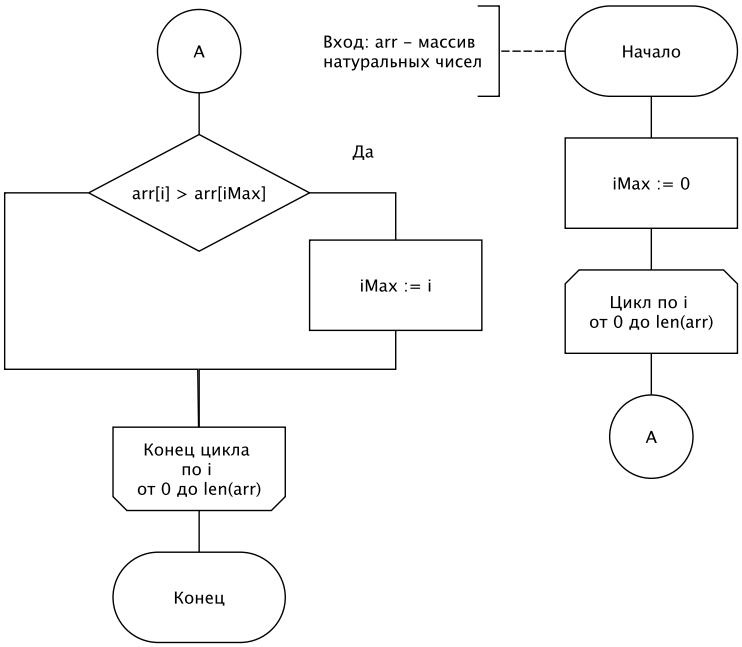

Рисунок 2.2 — Схема реализации алгоритма поиска индекса максимального элемента массива

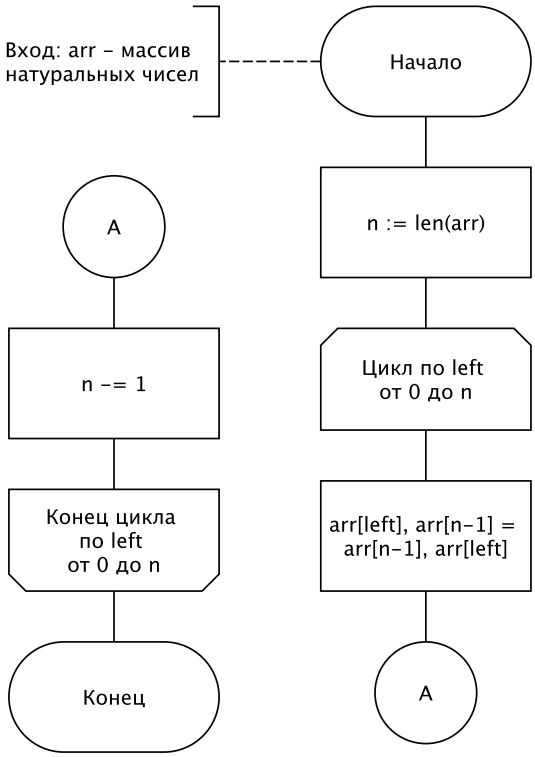

Рисунок 2.3 — Схема реализации алгоритма переворота массива

2.2. Разработка алгоритма быстрой сортировки

На рисунке 2.4 представлена схема реализации алгоритма быстрой сортировки.

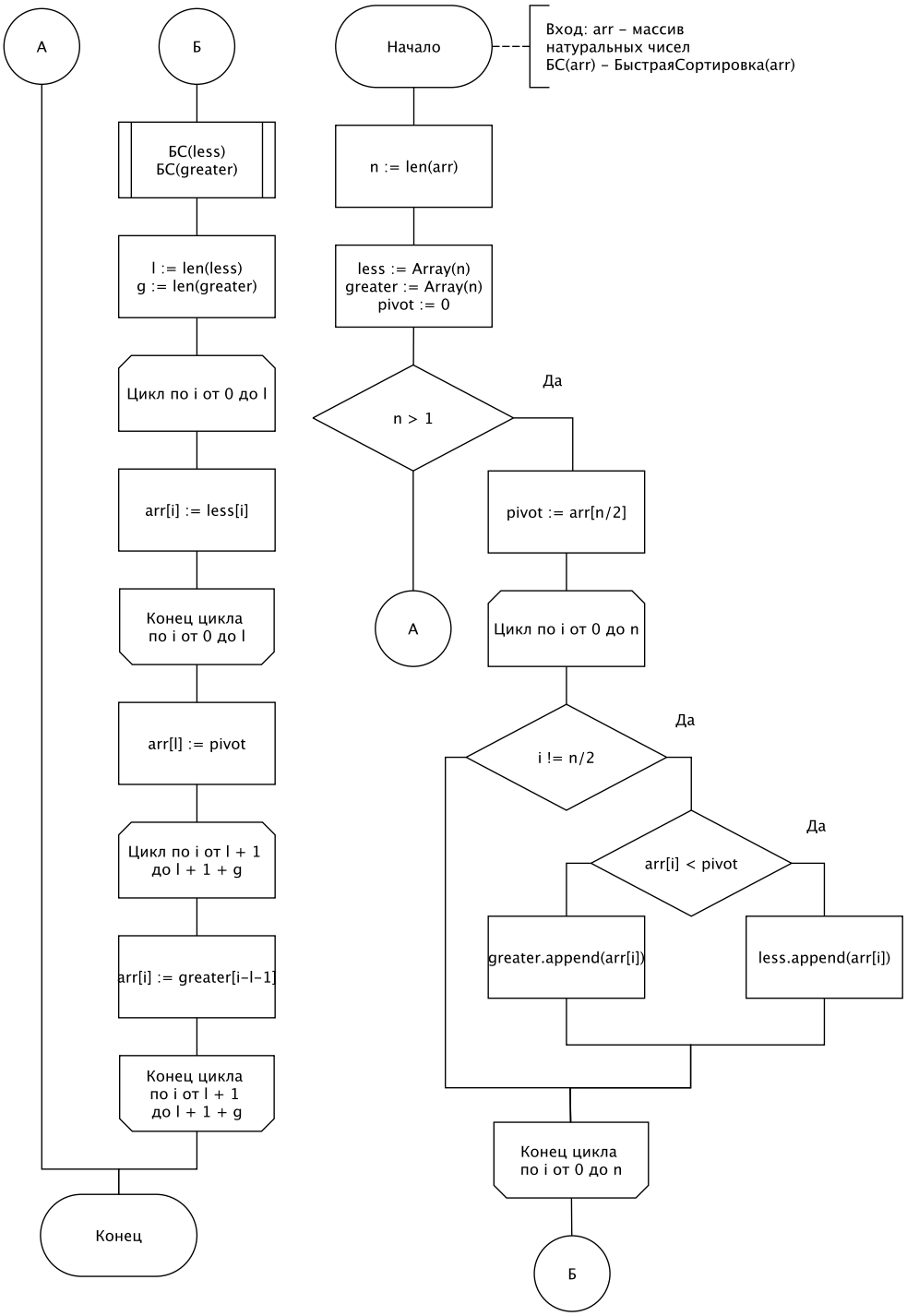

Рисунок 2.4 — Схема реализации алгоритма быстрой сортировки

2.3. Разработка алгоритма сортировки бусинами

На рисунке 2.5 представлена схема реализации алгоритма сортировки бусинами. Алгоритм использует подпрограмму поиска индекса максимального элемента массива, схема которой приведена на рисунке 2.2.

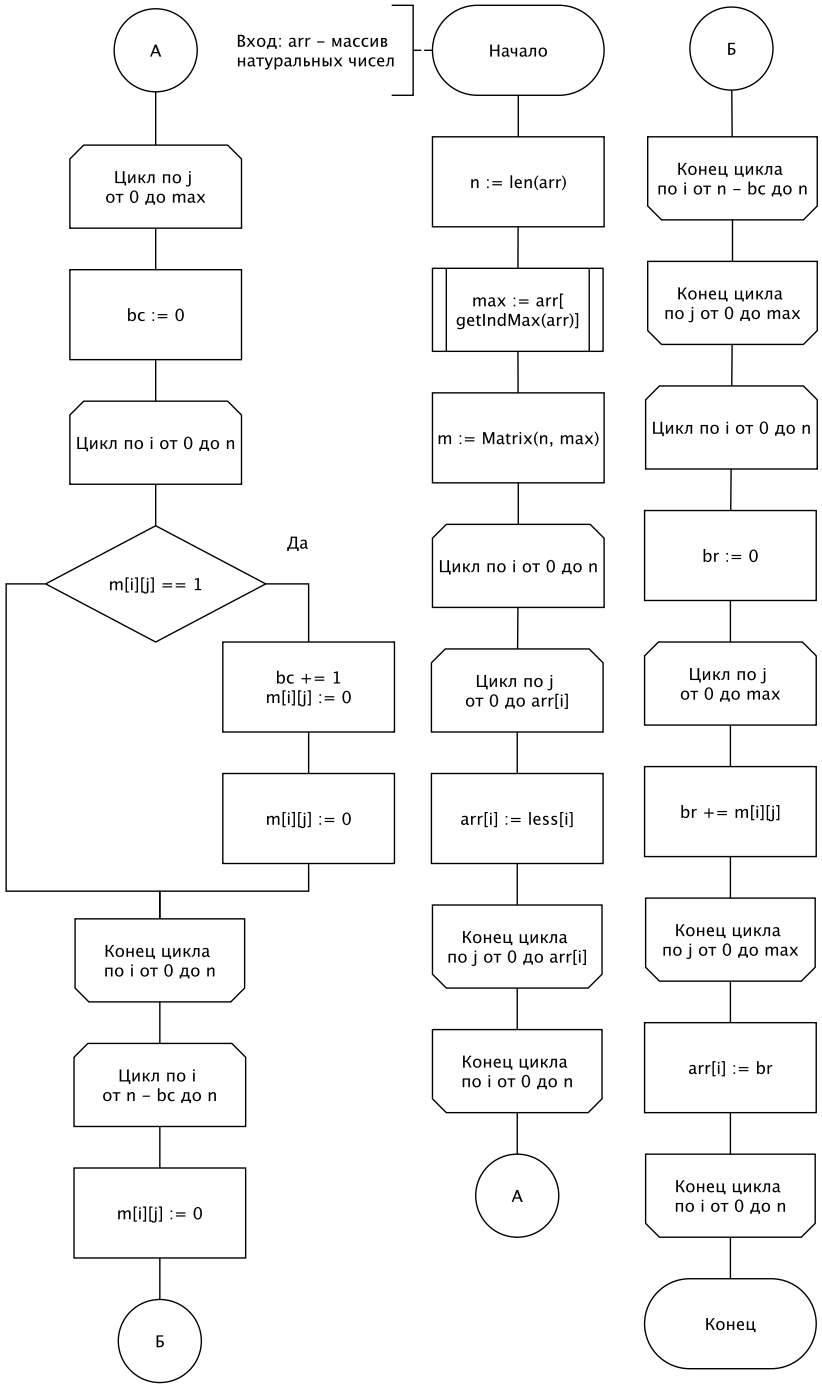

Рисунок 2.5 — Схема реализации алгоритма сортировки бусинами

2.4. Модель вычислений

Для вычисления трудоёмкости данных алгоритмов необходимо ввести модель вычислений.

Обозначим трудоёмкость [5] как f_a, где a — индекс, указывающий операцию, блок кода или оператор, для которого вычисляется трудоёмкость.

Определим трудоёмкость базовых операций как:

$$\begin{array}{rrrr} f_{+} = 1 & f_{-} = 1 & f_{+=} = 1 & f_{-=} = 1 \\ f_{:=} = 1 & f_{<<} = 1 & f_{>>} = 1 & f_{[]} = 1 \\ f_{++} = 1 & f_{--} = 1 & f_{>} = 1 & f_{<} = 1 \\ f_{>=} = 1 & f_{<=} = 1 & f_{!=} = 1 & f_{==} = 1 \\ f_{.} = 2 & f_{/} = 2 & f_{\%} = 2 & \end{array} \tag{2.1}$$

Определим трудоёмкость вызова функции как 0.

Определим трудоёмкость условия как

$$f_{if} = f_{cc} + \begin{cases} \min(f_1, f_2), & \text{в лучшем случае,} \\ \max(f_1, f_2), & \text{в худшем случае,} \end{cases} \tag{2.2}$$

где приняты следующие обозначения:

- f_{cc} — трудоёмкость вычисления условия;
- f_1 — трудоёмкость блока после if;
- f_2 — трудоёмкость блока после $else$.

Определим трудоёмкость цикла как

$$f_{loop} = f_{init} + f_{first_cmp} + n * (f_{body} + f_{inc} + f_{cmp}), \tag{2.3}$$

где приняты следующие обозначения:

- f_{init} — трудоёмкость инициализации;
- f_{first_cmp} — трудоёмкость первого сравнения;
- f_{body} — трудоёмкость тела цикла;
- n — количество итераций цикла;
- f_{inc} — трудоёмкость изменения индекса;
- f_{cmp} — трудоёмкость сравнения.

2.5. Трудоёмкость алгоритмов сортировки

Введём некоторые обозначения:

- N — размерность массива;
- f_{best} — трудоёмкость алгоритма в лучшем случае;
- f_{worst} — трудоёмкость алгоритма в худшем случае.

2.5.1. Трудоёмкость алгоритма блинной сортировки

Рассчитаем трудоёмкость алгоритма блинной сортировки. Рассмотрим одну итерацию внешнего цикла f_n для массива фиксированной длины n:

$$f_n = 4 + f_{indMax} + 5 + \begin{cases} 0, & \text{в лучшем случае (максимум в конце),} \\ f_{flip(n)} + f_{flip(n-1)}, & \text{в худшем случае (иначе).} \end{cases} \tag{2.4}$$

При этом

$$f_{indMax} = 3 + n \cdot (3 + 2 + \begin{cases} 0, & \text{в лучшем случае (максимум не сменился),} \\ 1, & \text{в худшем случае (иначе),} \end{cases}) \tag{2.5}$$

то есть

$$f_{indMax} = 3 + n \cdot \begin{cases} 5, & \text{в лучшем случае (максимум не сменился),} \\ 6, & \text{в худшем случае (иначе).} \end{cases} \tag{2.6}$$

Также рассмотрим переворот элементов до максимума включая в ситуации худшего случая — когда максимумом является предпоследний элемент (приходится делать поворот для многих элементов):

$$f_{flip(n-1)} = 3 + 11 \cdot n \tag{2.7}$$

и для переворота полного массива:

$$f_{flip(n)} = 3 + 11 \cdot n. \tag{2.8}$$

Тогда для лучшего случая (в этом случае массив отсортирован по возрастанию, поэтому максимум будет обновляться на каждом шаге) будет получено соотношение

$$f_n = 9 + 6 \cdot n + 0 = 9 + 6 \cdot n, \tag{2.9}$$

а для худшего (когда максимумами становятся предпоследние элементы относительно рассматриваемого массива, то есть большие и маленькие значения чередуются в массиве) будет получено соотношение

$$f_n = 9 + n/2 \cdot 6 + n/2 \cdot 5 + 22 \cdot n - 5 = 27.5 \cdot n + 4. \tag{2.10}$$

Таким образом

$$f_n = \begin{cases} 9 + 6 \cdot n, & \text{в лучшем случае,} \\ 27.5 \cdot n + 4, & \text{в худшем случае.} \end{cases} \tag{2.11}$$

Для всего цикла будет получена последовательность

$$f_{loop} = 2 + f_2 + f_3 + ... + f_N, \tag{2.12}$$

которую можно представить в виде арифметической прогрессии

$$f_{loop} = 2 + \frac{f_2 + f_N}{2} \cdot (N - 1). \tag{2.13}$$

Итоговыми выражениями будут

$$\begin{cases} f = 3 \cdot N^2 + 10 \cdot N - 13, & \text{в лучшем случае,} \\ f = \frac{27.5 \cdot N^2 + 35.5 \cdot N - 59}{2}, & \text{в худшем случае.} \end{cases} \tag{2.14}$$

Можно сделать вывод, что асимптотика алгоритма блинной сортировки составляет $O(N^2)$ и в худшем, и в лучшем случаях.

2.5.2. Трудоёмкость алгоритма быстрой сортировки

Рассчитаем трудоёмкость алгоритма быстрой сортировки. Рассмотрим запуск функции на одной итерации и впоследствии умножим на глубину рекурсии.

На одном запуске функции массив делится на части, каждая из которых обрабатывается рекурсивно данной функцией. Эти части в сумме образуют исходный массив, поэтому быстродействие одного рекурсивного погружения можно сравнить с быстродействием одной итерации на полном массиве, поэтому в данном расчёте используется значение N.

Тогда для одного вызова функции, кроме вызова для полного разбиения массива на части из одного элемента, который будет учтён позже

$$f = 8 + f_{loop} + 4 + f_{loopless} + f_{loopgreater}. \tag{2.15}$$

При этом

$$f_{loop} = 2 + (N - 1) \cdot (4 + f_{cond}) + 4, \tag{2.16}$$

где последнее слагаемое соответствует обработке опорного элемента, а слагаемое в середине — остальным.

Обозначение f_{cond} соответствует следующей формуле:

$$f_{loop} = 2 + \begin{cases} 1, & \text{если элемент меньше опорного,} \\ 1, & \text{если элемент больше или равен опорному.} \end{cases} \tag{2.17}$$

Также в формуле (2.15)

$$f_{loopless} = 2 + 4 \cdot m, \tag{2.18}$$

где

$$m = \begin{cases} \frac{N-1}{2}, & \text{в лучшем случае (равномерное разбиение),} \\ N - 1, & \text{в худшем случае (неравномерное разбиение),} \end{cases} \tag{2.19}$$

и

$$f_{loopgreater} = 5 + 9 \cdot k, \tag{2.20}$$

где

$$k = \begin{cases} \frac{N-1}{2}, & \text{в лучшем случае (равномерное разбиение),} \\ 0, & \text{в худшем случае (неравномерное разбиение).} \end{cases} \tag{2.21}$$

Тогда в промежуточной общей формуле

$$f_{sub} = 18 + 7 \cdot N + 5 \cdot m + 9 \cdot k. \tag{2.22}$$

В лучшем случае, то есть когда в массиве существует примерно равно количество элементов больше и элементов меньше опорного, глубина рекурсии составит $\log_2 N$.

В худшем случае, когда опорный элемент является минимумом или максимумом последовательности, глубина рекурсии составит N.

Тогда после домножения результирующей формулы для одного погружения на соответствующие коэффициенты для двух случаев получается следующее соотношение:

$$f = \begin{cases} f_{sub} \cdot \log_2 N, & \text{в лучшем случае (равномерное разбиение),} \\ f_{sub} \cdot N, & \text{в худшем случае (неравномерное разбиение).} \end{cases} \tag{2.23}$$

Осталось учесть итерации, на которых в функцию приходит массив из одного элемента:

$$f = \begin{cases} f_{sub} \cdot (\log_2 N - 1) + 5, & \text{в лучшем случае (равномерное разбиение),} \\ f_{sub} \cdot (N - 1) + 5, & \text{в худшем случае (неравномерное разбиение).} \end{cases} \tag{2.24}$$

Можно сделать вывод, что асимптотика алгоритма быстрой сортировки составляет $O(N^2)$ в худшем случае, и $O(N \cdot \log_2 N)$ в лучшем случае.

2.5.3. Трудоёмкость алгоритма сортировки бусинами (гравитационной)

Рассчитаем трудоёмкость алгоритма сортировки бусинами.

$$f = 12 + f_{indMax} + N \cdot 3 + N \cdot a + c + (N \cdot (7 + m \cdot 5)). \tag{2.25}$$

При этом

$$f_{indMax} = 3 + N \cdot (3 + 2 + \begin{cases} 0, & \text{в лучшем случае (максимум не сменился),} \\ 1, & \text{в худшем случае (иначе),} \end{cases} \tag{2.26}$$

$$a = 5 + m \cdot 6, \tag{2.27}$$

$$b = \begin{cases} 4, & \text{при равенстве элемента матрицы единице,} \\ 0, & \text{иначе,} \end{cases} \tag{2.28}$$

$$c = m \cdot (8 + N \cdot (5 + b) + N \cdot 5), \tag{2.29}$$

а m — это максимальный элемент последовательности.

Так как худшим случаем для сортировки бусинами является ситуация, когда весь массив состоит из элементов с большими значениями, а лучший — из элементов с маленькими значениями, массивы при тестировании заполнялись одинаковыми элементами равными или размерности массива, или единице для двух случаев соответственно. Следовательно и значение m всегда равнялось либо размерности массива, либо единице соответственно. От порядка значений в массиве быстродействие алгоритма или не меняется совсем, или меняется совсем незначительно. Поэтому для f_{indMax} вариативная часть всегда будет равняться нулю, а b всегда будет равняться 4.

После выполнения расчётов по (2.25) получается следующее соотношение:

$$f = 21 \cdot N + 25 \cdot m \cdot N + 8 \cdot m + 15. \tag{2.30}$$

Как было упомянуло выше, значение m всегда равнялось либо размерности массива, либо единице соответственно для худшего и лучшего случаев, поэтому для этих случаев формула принимает следующий вид:

$$f = \begin{cases} 21 \cdot N + 25 \cdot 1 \cdot N + 23, & \text{в лучшем случае,} \\ 21 \cdot N + 25 \cdot N^2 + 8 \cdot N + 15, & \text{в худшем случае.} \end{cases} \tag{2.31}$$

Можно сделать вывод, что асимптотика алгоритма сортировки бусинами составляет $O(N^2)$ в худшем случае, и $O(N)$ в лучшем случае.

3. Технологическая часть

В данном разделе будет представлена реализация алгоритмов сортировки. Также будут указаны обязательные требования к ПО, средства реализации алгоритмов и результаты проведённого тестирования программы.

3.1. Требования к ПО

Для программы выделен перечень требований:

- программа предоставляет интерфейс в формате меню с возможностью ввода и изменения обрабатываемого массива, завершения работы с программой и выбора используемого для обработки введённого массива алгоритма сортировки;
- предлагается повторно выполнить ввод при невалидном выборе пункта меню;
- производится аварийное завершение с текстом об ошибке при иных ошибках;
- проводится модульное тестирование функций сортировки;
- производятся замеры времени выполнения и потребления памяти функциями сортировки;
- вызывается исключение при вводе пустого массива;
- сортируются массивы только по возрастанию;
- допускается только сортировка массивов натуральных чисел.

3.2. Средства реализации

Для реализации данной работы был выбран язык программирования Go [1]. Выбор обусловлен наличием в *Go* библиотек для тестирования ПО и проведения замеров времени выполнения в наносекундах [7], а также необходимых для реализации поставленных цели и задач средств. В качестве среды разработки была выбрана *GoLand* [3].

3.3. Реализация алгоритмов

В листингах 3.3 – 3.5 представлены реализации алгоритмов сортировок: блинной, быстрой и сортировки бусинами, и некоторых нужных подпрограмм в листингах 3.1 – 3.2.

Листинг 3.1 — Листинг функции поиска индекса максимума массива

```
func getIndMax(arr []int) int {
        iMax := 0
        for i := range arr {
                if arr[i] > arr[iMax] {
                        iMax = i
                }
        }
        return iMax
}
```

Листинг 3.2 — Листинг функции поиска переворота массива

```
func flip(arr []int) {
        n := len(arr)
        for left := 0; left < n; left++ {
                arr[left], arr[n-1] = arr[n-1], arr[left]
                n--
        }
}
```

Листинг 3.3 — Листинг алгоритма блинной сортировки

```
func Pancakesort(arr []int) {
        for n := len(arr); n > 1; n-- {
                iMax := getIndMax(arr[:n])
                if iMax != n-1 {
                        flip(arr[:(iMax + 1)])
                        flip(arr[:n])
                }
        }
}
```

Листинг 3.4 — Листинг алгоритма быстрой сортировки

```
func quicksort(arr []int) {
        n := len(arr)
        less := make([]int, 0, n)
        greater := make([]int, 0, n)
        pivot := 0

        if n > 1 {
                pivot = arr[n/2]

                for i, v := range arr {
                        if i != n/2 {
                                if v < pivot {
                                        less = append(less, v)
                                } else {
                                        greater = append(greater, v)
                                }
                        }
                }

                quicksort(less)
                quicksort(greater)

                l := len(less)
                g := len(greater)

                for i := 0; i < l; i++ {
                        arr[i] = less[i]
                }
                arr[l] = pivot
                for i := l + 1; i < l+g+1; i++ {
                        arr[i] = greater[i-l-1]
                }
        }
}
```

Листинг 3.5 — Листинг алгоритма сортировки бусинами

```
func Beadsort(arr []int) {
        n := len(arr)
        max := arr[getIndMax(arr)]
        m := make([][]int, n)
        for i := 0; i < n; i++ {
                m[i] = make([]int, max)
        }
        for i := 0; i < n; i++ {
                for j := 0; j < arr[i]; j++ {
                        m[i][j]++
                }
        }

        for j := 0; j < max; j++ {
                beadsInColumn := 0
                for i := 0; i < n; i++ {
                        if m[i][j] == 1 {
                                beadsInColumn++
                                m[i][j] = 0
                        }
                }
                for i := n - beadsInColumn; i < n; i++ {
                        m[i][j] = 1
                }
        }

        for i := 0; i < n; i++ {
                beadsInRow := 0
                for j := 0; j < max; j++ {
                        beadsInRow += m[i][j]
                }
                arr[i] = beadsInRow
        }
}
```

3.4. Тестирование

В таблице представлены тесты для алгоритмов блинной, быстрой и бусинной сортировки. Тестирование проводилось по методологии чёрного ящика. Все тесты пройдены успешно.

Таблица 3.1 — Тесты для алгоритмов блинной, быстрой и бусинной сортировки

№	Входной массив	Результат
1	4, 2, 7, 5, 8, 1, 6	1, 2, 4, 5, 6, 7, 8
2	4, 2, 2, 5, 2, 1, 6	1, 2, 2, 2, 4, 5, 6
3	2, 4, 5, 6, 8, 10	2, 4, 5, 6, 8, 10
4	10, 8, 6, 5, 4, 2	2, 4, 5, 6, 8, 10
5	2, 2, 2, 2, 2	2, 2, 2, 2, 2
6	2	2
7	4, 2, 7, 5, 8, 10, 6, 21, 7, 3, 11, 9, 1	1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10, 11, 21

4. Экспериментальная часть

В данном разделе описаны проведённые замеры и представлены результаты исследования. Также будут уточнены характеристики устройства, на котором проводились замеры.

4.1. Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование [4]:

- операционная система macOS Monterey 12.4;
- 8 Гб оперативной памяти;
- процессор Apple M2 (базовая частота — 2400 МГц, но поддержка технологии Turbo Boost позволяет достигать частоты в 3500 МГц [6]).

4.2. Измерение времени выполнения реализаций алгоритмов

Тестирование реализаций алгоритмов производилось при помощи встроенных в Go средств, а именно «бенчмарков» (*benchmarks* из пакета *testing* стандартной библиотеки *Go* [2]), представляющих собой тесты производительности. По умолчанию производятся только замеры времени выполнения в наносекундах [7][8], но при добавлении ключа *−benchmem* также выполняется замер потребления памяти и количества аллокаций памяти.

Значение N динамически изменяется для достижения стабильного результата при различных условиях, но гарантируется, что каждый «бенчмарк» будет выполняться хотя бы одну секунду. Для замеров использовались массивы равной длины, генерирующиеся различными способами, в зависимости от особенностей худших и лучших случаев для алгоритмов, перед началом выполнения «бенчмарков». Результаты тестирования возвращаются в структуре специального вида. Пример такой структуры представлен в листинге 4.6.

Листинг 4.6 — Листинг структуры результата «бенчмарка»

```
testing.BenchmarkResult{N:120000, T:1200000000, Bytes:0, MemAllocs:0x0,
MemBytes:0x0, Extra:map[string]float64{}}
```

В листинге 4.7 представлен пример реализации «бенчмарка», где $alg.function$ — объект типа функция (в данной реализации — функция, описывающая один из алгоритмов сортировки).

Листинг 4.7 — Листинг примера реализации «бенчмарка»

```
func NewBenchmark(arr []int, alg algorithm) func(*testing.B) {
        return func(b *testing.B) {
                copyA := make([]int, len(arr))
                b.ResetTimer()
                for j := 0; j < b.N; j++ {
                        b.StopTimer()
                        copy(copyA, arr)
                        b.StartTimer()
                        alg.function(copyA)
                }
        }
}
```

Входные массивы для проведения замеров генерируются следующими способами:

- генерация случайных чисел в диапазоне от 1 до значения длины массива;
- заполнение числами от 1 до значения длины массива в порядке возрастания;
- заполнение числами от 1 до значения длины массива в порядке убывания;
- заполнение одним значением;
- заполнение с чередованием элементов по величине.

Пример функции для генерации массива представлен в листинге 4.8.

Листинг 4.8 — Листинг примера функции для генерации массива с заполнением случайными числами в диапазоне от 1 до значения длины массива

```
func generateArray(size int) []int {
        rand.Seed(time.Now().UnixNano())

        arr := make([]int, 0, size)
        for j := 0; j < size; j++ {
                arr = append(arr, rand.Intn(size)+1)
        }

        return arr
}
```

Результаты замеров времени выполнения (в нс.) приведены в таблицах 4.1 — 4.3. На рисунках 4.1 — 4.3 приведены графики, отображающие зависимость времени работы алгоритмов от длины массива для произвольного, лучшего и худшего случаев. Массивы заполнялись случайными числами.

Таблица 4.1 — Результаты замеров времени для произвольного случая (нс.)

Длина массива	Блинная	Быстрая	Бусинами
1	50	452	508
10	224	1701	2067
50	2594	8156	16 532
100	9898	12 416	48 092
200	40 020	25 810	247 140
300	87 373	34 310	782 144
500	230 788	59 792	2 566 986
600	326 217	67 896	2 606 156
800	568 568	98 698	4 774 437
1000	874 516	127 788	11 649 473

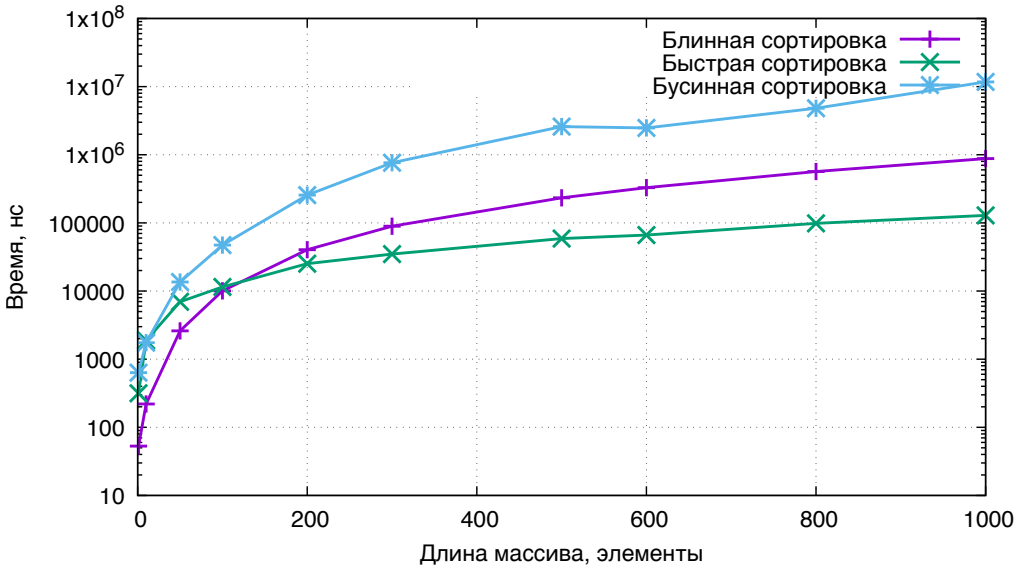

Рисунок 4.1 — Зависимость времени работы алгоритмов сортировки от длины массива

Таблица 4.2 — Результаты замеров времени для лучшего случая (нс.)

Длина массива	Блинная	Быстрая	Бусинами
1	51	326	516
10	131	1476	928
50	1690	5718	1972
100	5658	9916	3518
200	21 382	17 774	5847
300	47 410	26 439	8335
500	130 256	41 003	12 737
600	186 585	51 371	16 124
800	330 680	70 031	20 843
1000	515 007	84 321	25 283

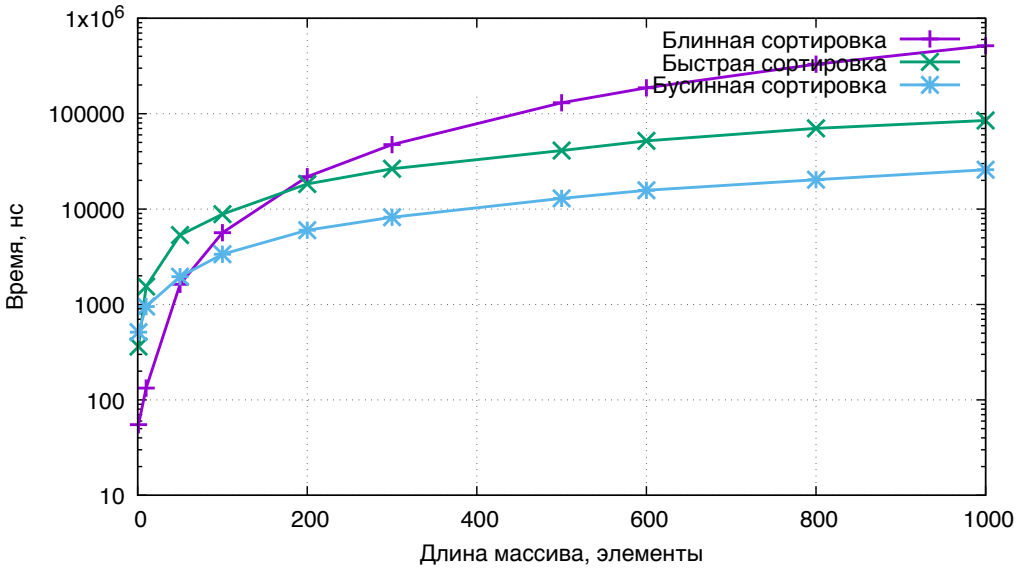

Рисунок 4.2 — Зависимость времени работы алгоритмов сортировки от длины массива

Таблица 4.3 — Результаты замеров времени для худшего случая (нс.)

Длина массива	Блинная	Быстрая	Бусинами
1	52	272	528
10	258	2085	2654
50	3209	16 191	17 435
100	12 278	38 466	57 557
200	47 484	122 015	432 274
300	103 360	241 772	1 295 117
500	278 107	683 739	4 296 667
600	394 568	879 124	5 070 336
800	693 190	1 549 153	7 614 436
1000	1 074 233	2 262 414	20 137 148

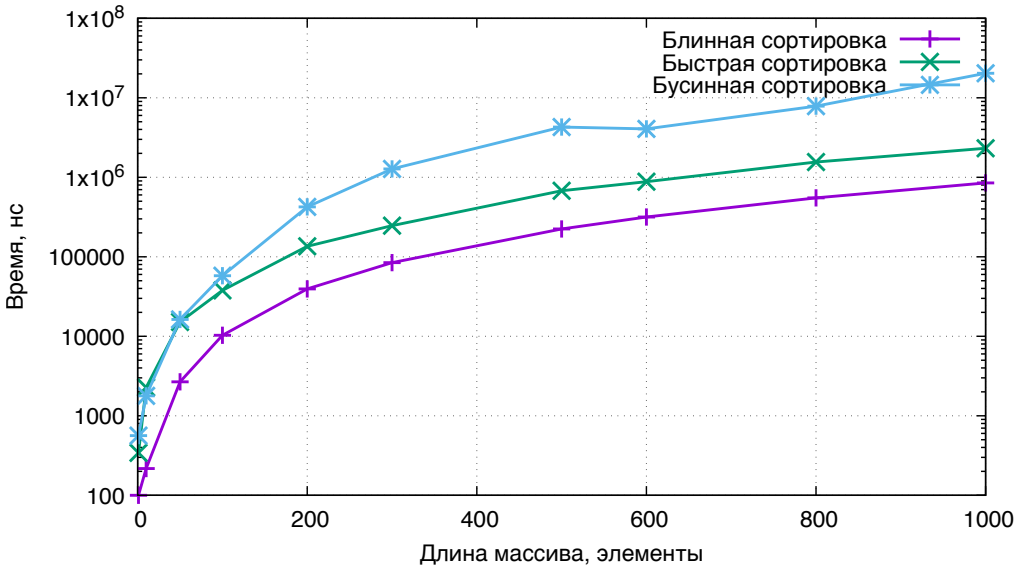

Рисунок 4.3 — Зависимость времени работы алгоритмов сортировки от длины массива

4.3. Измерение объёма потребляемой памяти реализаций

Измерение объёма потребляемой памяти производилось посредством создания собственного программного модуля memory, использующего функции пакета unsafe языка *Go* [2]. Реализация основного функционала модуля, а также пример функции расчёта памяти, затрачиваемой на алгоритм блинной сортировки, и пример использования указанных функций приведены в Приложении А.

Результаты замеров потребляемой памяти (в байтах) приведены в таблице 4.4. На рисунке 4.4 приведён график, отображающий зависимость потребляемой памяти от длины массива. Массивы заполнялись случайными числами.

Таблица 4.4 — Результаты замеров потребляемой памяти (в байтах)

Длина массива	Блинная	Быстрая	Бусинами
1	128	176	168
10	128	1232	384
50	128	3176	4544
100	128	4872	64 944
200	128	11 208	160 144
300	128	12 320	72 144
500	128	16 816	1 372 144
600	128	20 184	2 380 944
800	128	31 896	1 043 344
1000	128	35 896	712 144

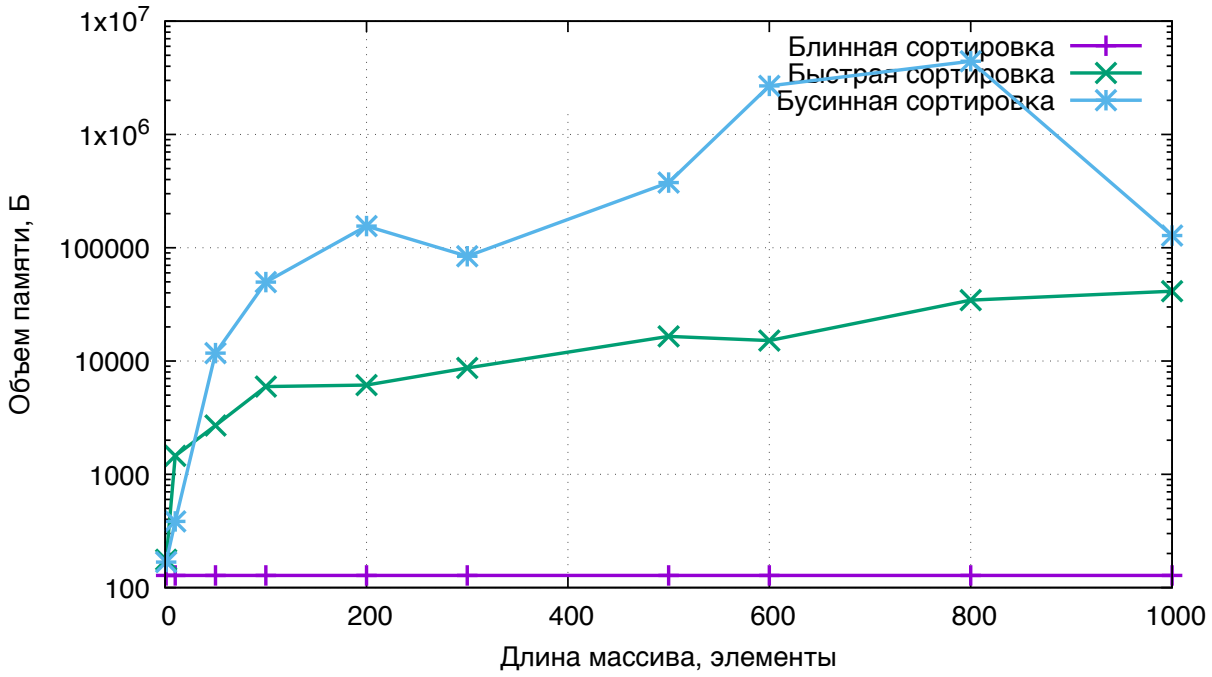

Рисунок 4.4 — Зависимость потребляемой памяти при сортировке от длины массива

Заключение

Для каждого алгоритма были выявлены лучший и худший случаи (данные, при которых алгоритмы работают быстрее всего и медленнее всего, соответственно).

Для блинной сортировки худшим случаем является массив с чередованием больших и маленьких значений элементов, так как на таких данных приходится делать более трудоёмкие перевороты. Лучший случай для этого алгоритма — уже отсортированный по возрастанию массив, в этом случае нет необходимости осуществлять перевороты.

Для быстрой сортировки худшим случаем является массив заполненный, например, одинаковыми значениями, так как на таких данных размеры частей, на которые алгоритм делит исходный массив, будут неравномерны, и трудоёмкость резко возрастёт. Лучший случай для этого алгоритма — уже отсортированный по возрастанию массив, в этом случае размеры частей, на которые алгоритм делит исходный массив, будут более равномерны, а глубина рекурсии — меньше.

Для сортировки бусинами худшим случаем является массив заполненный большими значениями, так как на таких данных размер выделяемой матрицы (в частности, количество столбцов) увеличивается. Соответственно лучший случай для этого алгоритма — массив заполненный небольшими значениями. Изменение порядка элементов в исходном массиве влияет на время выполнения незначительно.

В результате проведения замеров с учётом разных случаев были сформулированы нижеперечисленные выводы.

В произвольных случаях (когда массив заполнялся случайными числами) до размера массива в 100 элементов блинная сортировка работает быстрее всего, примерно в 1.25 раз опережая быструю сортировку. Медленнее всего работает сортировка бусинами, уступая блинной почти в 5 раз. Начиная с размера массива в 200 элементов быстрее остальных начинает работать быстрая сортировка. На 600 элементах она быстрее блинной в 5 раз, а сортировки бусинами в 38 раз.

В лучших случаях на размерах массивов начиная со 100 сортировка бусинами работает быстрее остальных. На 300 элементах она работает меньше блинной в 5 раз и меньше быстрой в 3 раза. На небольших длинах массивов (до 50) остальные сортировки уступают блинной, но уже на 600 элементах быстрая сортировка работает быстрее блинной в 3 раза.

В худших случаях алгоритм сортировки бусинами работает дольше остальных алгоритмов на любых длинах строк. Также на всех рассматриваемых длинах в худших случаях быстрее остальных работает блинная сортировка. На длине массива в 500 символов она

быстрее быстрой в 2 раза и гравитационной в 15 раз.

Если говорить об объёме занимаемой памяти, то алгоритм сортировки бусинами практически на всех длинах массива (начиная от 500-100) потребляет сильно больше памяти, чем остальные. Это объясняется необходимостью хранить матрицу значений. Уже на длине в 100 элементов этот алгоритм потребляет в 507 раз больше памяти, чем блинная сортировка, и в 13 раз, чем сортировка бусинами. Меньше всех памяти (одинаковый объём при любых длинах, так как это независимый параметр) потребляет блинная сортировка, так как не хранит большие массивы данных, как гравитационная, и не потребляет память за счёт выделения собственной области стека под рекурсивные вызовы, в отличие от быстрой сортировки.

В ходе выполнения лабораторной работы была достигнута поставленная цель: были получены навыки программирования, тестирования полученного программного продукта и проведения замеров времени выполнения и потребляемой памяти по результатам работы программы на примере реализации алгоритмов сортировки.

В процессе выполнения лабораторной работы были также реализованы все поставленные задачи, а именно:

- были изучены алгоритмы блинной, быстрой и бусинной сортировки;
- были разработаны схемы данных алгоритмов и проведён анализ их трудоёмкости;
- была выполнена программная реализация данных алгоритмов;
- были проведены замеры потребления памяти (в байтах) и времени работы (в нс) для данных алгоритмов;
- была получена графическая зависимость измеренных величин от длины последовательности символов, предоставляемой на вход алгоритмам;
- был проведен сравнительный анализ данных алгоритмов на основе полученных зависимостей.

Список использованных источников

[1] Документация по языку программирования *Go* [Электронный ресурс]. Режим доступа: *https://go.dev/doc/* (дата обращения: 20.09.2022).

[2] Документация по пакетам языка программирования *Go* [Электронный ресурс]. Режим доступа: *https://pkg.go.dev* (дата обращения: 20.09.2022).

[3] GoLand: IDE для профессиональной разработки на *Go* [Электронный ресурс]. Режим доступа: *https://www.jetbrains.com/ru-ru/go/* (дата обращения: 20.09.2022).

[4] Техническая спецификация ноутбука *MacBookAir* [Электронный ресурс]. Режим доступа: *https://support.apple.com/kb/SP869* (дата обращения: 20.09.2022).

[5] Сложность алгоритмов [Электронный ресурс]. Режим доступа: *http://kuimova.ucoz.ru/modul_6-slozhnost_algoritmov.pdf* (дата обращения: 20.09.2022).

[6] *AppleM2* [Электронный ресурс]. Режим доступа: *https://www.notebookcheck.net/Apple-M2-Processor-Benchmarks-and-Specs.632312.0.html* (дата обращения: 10.10.2022).

[7] Исходный код *src/testing/benchmark.go* [Электронный ресурс]. Режим доступа: *https://go.dev/src/testing/benchmark.go* (дата обращения: 10.10.2022).

[8] Документация по функции *mach_absolute_time* [Электронный ресурс]. Режим доступа: *https://go.dev/src/testing/benchmark.go* (дата обращения: 10.10.2022).

Приложение А

В ходе выполнения лабораторной работы в соответствии с поставленными задачами было необходимо произвести замеры потребляемой при выполнении функций, реализующих заданные алгоритмы, памяти. В связи с этим был разработан программный модуль *memory* для измерения потребляемой функцией памяти в байтах (замеры реализованы только для функций, представляющих алгоритмы по заданию лабораторной работы).

«Бенчмарки», использованные для измерения затрачиваемого на исполнение функции времени, предоставляют возможность измерить только память, выделяемую на куче, что не соответствует поставленной задаче.

При вызове функции в языке *Go* для неё выделяется область собственного стека, что особенно критично для рекурсивных алгоритмах. Соответственно, необходимо иметь возможность измерять потребляемую память и на стеке, так как в ином случае не будут получены реалистичные результаты замеров и построить зависимость, отражающую действительное потребление памяти в зависимости от длины массива, будет невозможно.

В листингах 5.1 — 5.3 приведена реализация основных структур и базового функционала модуля *memory* для измерения потребляемой функцией памяти в байтах.

Листинг 5.1 — Листинг основных структур и базового функционала модуля *memory* для измерения потребляемой функцией памяти в байтах (начало)

```
package algorithms

import (
        "unsafe"
)

var MemoryInfo Metrics

type Metrics struct {
        current int
        max     int
}
```

Листинг 5.2 — Листинг основных структур и базового функционала модуля *memory* для измерения потребляемой функцией памяти в байтах (продолжение листинга 5.1)

```
// сброс рассчитанных значений
func (m *Metrics) Reset() {
        m.current = 0
        m.max = 0
}

// добавление значения к общей сумме потребляемой памяти,
// обновление максимума
func (m *Metrics) Add(v int) {
        m.current += v
        if m.current > m.max {
                m.max = m.current
        }
}

// вычитание значения из общей суммы потребляемой памяти
func (m *Metrics) Done(v int) {
        m.current -= v
}

// получение значения макс. потребления памяти за выполнение функции
func (m *Metrics) Max() int64 {
        return int64(m.max)
}

// получение размера типа данных
// пример вызова: sizeOf[int]()
func sizeOf[T any]() int {
        var v T
        return int(unsafe.Sizeof(v))
}
```

Листинг 5.3 — Листинг основных структур и базового функционала модуля *memory* для измерения потребляемой функцией памяти в байтах (окончание листинга 5.2)

```
// получение полного размера среза (заголовок + элементы)
// пример вызова: sizeOfArray[int](10)
func sizeOfArray[T any](n int) int {
        return sizeOf[[]T]() + n*sizeOf[T]()
}
```

В листинге 5.4 приведена реализация одной из функций (в данном случае функции, реализующей алгоритм блинной сортировки), вычисляющих потребление памяти конкретной функцией, модуля *memory*.

Листинг 5.4 — Листинг функции, вычисляющей потребление памяти функцией, реализующей алгоритм блинной сортировки

```
func memoryPancake(arr []int) int {
        a := sizeOf[[]int]()

        vars := 2 * sizeOf[int]()

        getMaxFunc := sizeOf[[]int]() + 3*sizeOf[int]()
        flipFunc := sizeOf[[]int]() + 2*sizeOf[int]()

        return a + vars + getMaxFunc + flipFunc
}
```

В листинге 5.5 приведен пример использования функций модуля *memory* в реализации алгоритма блинной сортировки.

Листинг 5.5 — Листинг использования функций модуля *memory* в реализации алгоритма блинной сортировки

```
func Pancakesort(arr []int) {
        MemoryInfo.Reset()
        MemoryInfo.Add(memoryPancake(arr))
        defer MemoryInfo.Done(memoryPancake(arr))

        for n := len(arr); n > 1; n-- {
                iMax := getIndMax(arr[:n])
                if iMax != n-1 {
                        flip(arr[:(iMax + 1)])
                        flip(arr[:n])
                }
        }
}
```

Таким образом, данный модуль позволяет измерить полное потребление памяти функциями, реализующими алгоритмы сортировки, и получить реалистичные данные по результатам измерений.