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

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

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

Тема Алгоритмы сортировки

Студент Якуба Д. В.

Группа ИУ7-53Б

Оценка (баллы) _____

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

Москва — 2020 г.

Оглавление

Введение

Цели лабораторной работы

1. изучение алгоритмов сортировки пузырьком, вставками и выбором;
2. реализация алгоритмов сортировки пузырьком, вставками и выбором;
3. проведение сравнительного анализа трудоёмкости алгоритмов на основе теоретических расчетов и выбранной модели вычислений;
4. сравнительный анализ алгоритмов на основе экспериментальных данных;
5. подготовка отчёта по лабораторной работе;
6. получение практических навыков реализации алгоритмов на ЯП Kotlin.

Определение

Сортировка - это процесс перегруппироваки заданной последовательности объектов в некотором определённом порядке. Такой определённый порядок позволяет, в некоторых случаях, эффективнее работать с заданной последовательностью.

Пусть требуется упорядочить N элементов: $E_1, E_2, ..., E_n$. Каждый элемент представляет из себя запись E_j, содержащую некоторую информацию и ключ K_j, управляющий процессом сортировки. На множестве ключей определено отношение порядка $<$ так, чтобы для любых трёх значений ключей a, b, c выполнялись следующие условия:

- Либо $a < b$, либо $b < c$, либо $a = b$;
- Если $a < b$ и $b < c$, то $a < c$.

Данные условия определяют математическое понятие линейного и совершенного упорядочения, а удовлетворяющие им множества поддаются сортировке болшинством методов.

Задачей сортировки является нахождение такой перестановки записей $p(1)p(2)...p(n)$ с индексами $1, 2, ..., N$, после которой ключи расположились бы в порядке неубывания.

$$K_{p(1)} \leq K_{p(2)} \leq ... \leq K_{p(n)} \tag{1}$$

1 Аналитическая часть

1.1 Алгоритм сортировки пузырьком

Алгоритм заключается в повторяющих проходах по сортируемому массиву. За каждый проход элементы последовательно сравниваются попарно. В том случае, если два элемента расположены не по порядку, то они меняются местами [1]. Этот процесс повторяется до тех пор, пока элементы не будут упорядочены, то есть, в случае массива элементов размером n, проходы повторятся $n - 1$ раз.

1.2 Алгоритм сортировки вставками

Алгоритм заключается в следующей последовательности действий: элементы просматриваются по одному, и каждый новый элемент вставляется в подходящее место среди ранее упорядоченных элементов [1].

В начальный момент времени отсортированная последовательность пуста. На каждом шаге алгоритма выбирается один из элементов входных данных и помещается на нужную позицию в уже осотированной последовательности до тех пор, пока набор входных данных не будет исчерпан. В любой момент времени в отсортированной последовательности элеметы удовлетворяют требованиям к выходным данным алгоритма.

1.3 Алгоритм сортировки выбором

Алгорим заключается в следующей последовательности действий: сначала выделяется наименьший (наибольший) элемент последовательности и каким-либо образом отделяется от остальных, затем выбирается

наименьший (наибольший) из оставшихся и т.д. [1]

Вывод

Были рассмотрены алгоритмы сортировки пузырьком, вставками и выбором. В данной работе стоит задача реализации рассмотренных алгоритмов. Также будет необходимо оценить теоретическую оценку алгоритмов и проверить её экспериментально.

2 Конструкторская часть

2.1 Блок-схема алгоритма сортировки пузырьком

Блок-схема алгоритма сортировки пузырьком предоставлена на рисунке 2.1.

2.2 Блок-схема алгоритма сортировки вставками

Блок-схема алгоритма сортировки вставками предоставлена на рисунке 2.2.

2.3 Блок-схема алгоритма сортировки выбором

Блок-схема алгоритма сортировки выбором предоставлена на рисунке 2.3.

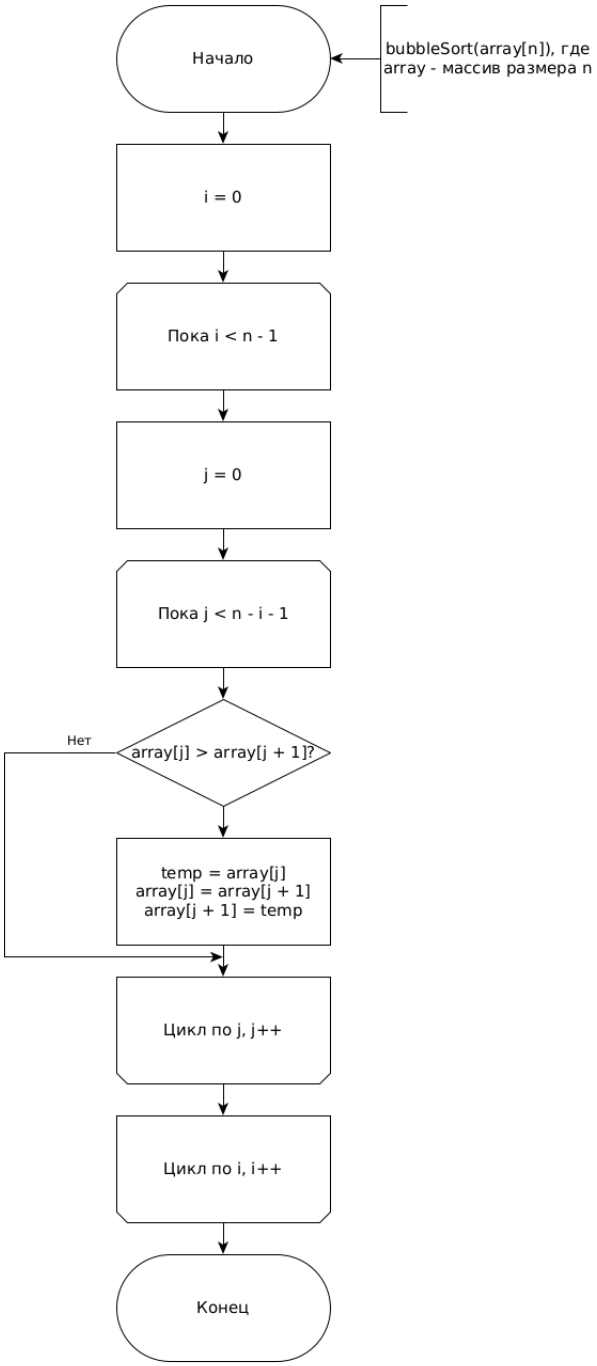

Рис. 2.1: Блок-схема алгоритма сортировки пузырьком.

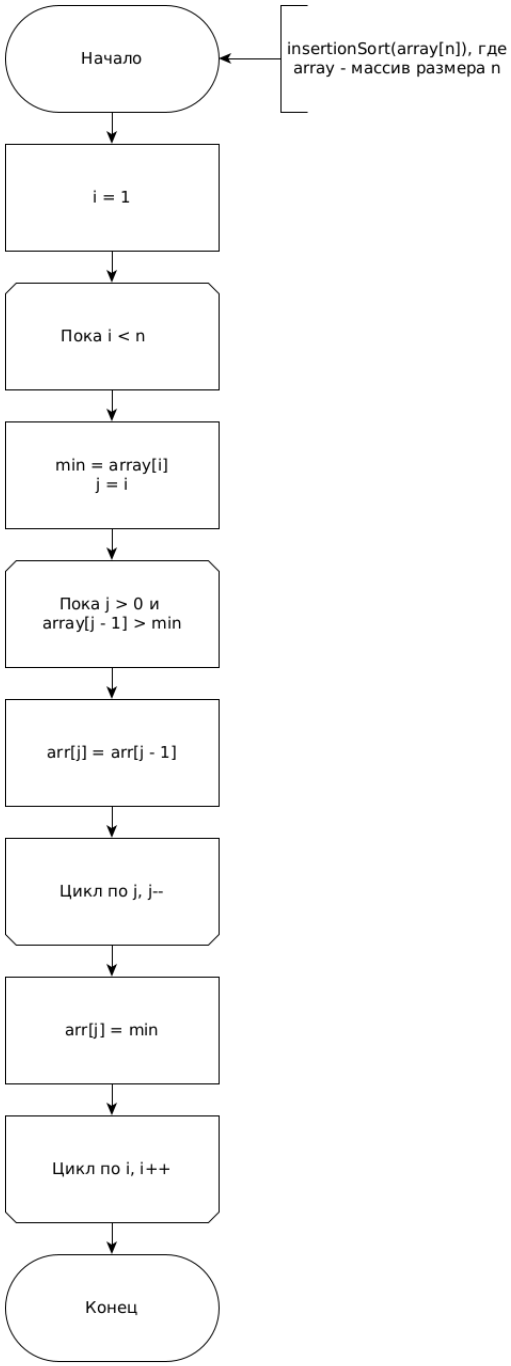

Рис. 2.2: Блок-схема алгоритма сортировки вставками.

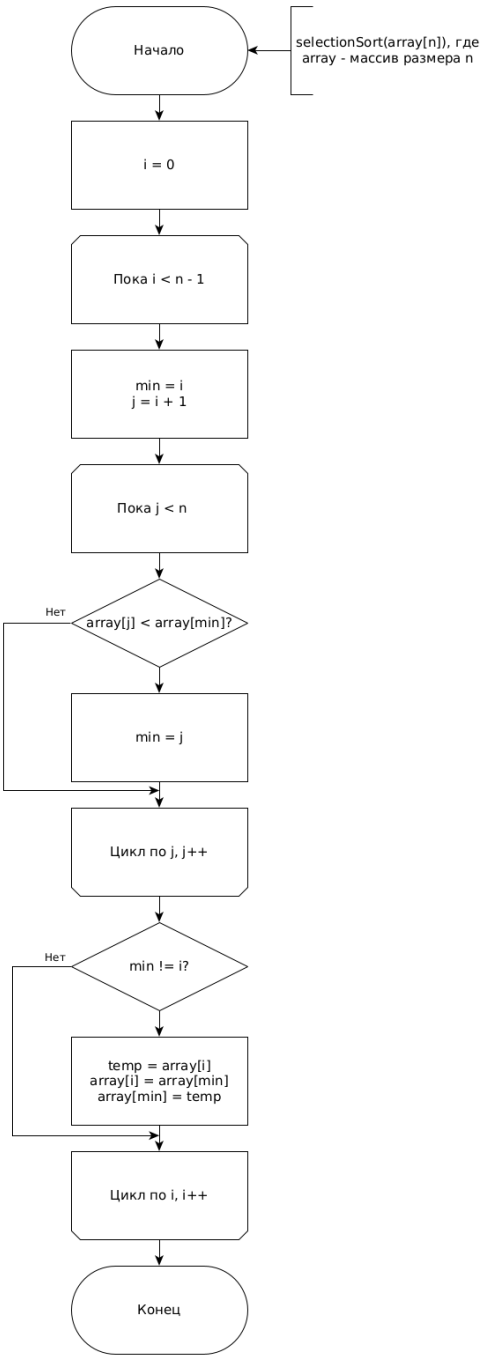

Рис. 2.3: Блок-схема алгоритма сортировки выбором.

2.4 Модель вычислений

ля последующего вычисления трудоемкости необходимо ввести модель вычислений:

1. операции из списка (2.1) имеют трудоемкость 1;

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

2. трудоемкость оператора выбора **условие then A else B** рассчитывается, как (2.2);

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

3. трудоемкость цикла рассчитывается, как (2.3);

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

4. трудоемкость вызова функции равна 0.

2.5 Трудоёмкость алгоритмов

2.5.1 Соглашение

Далее под некоторой величиной $size$ будет подразумеваться длина переданного массива.

2.5.2 Алгоритм сортировки пузырьком

Трудоёмкость алгоритма сортировки пузырьком будет включать включать в себя:

- цикл по $i \in [0..size - 1]$;
- цикл по $j \in [0..size - 1 - i]$, включённый в цикл по i.

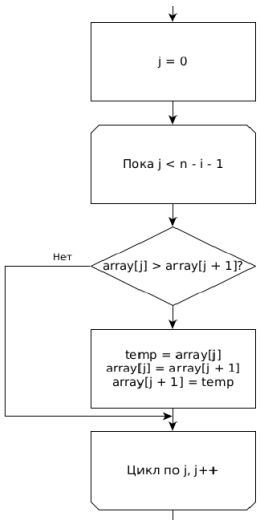

Рис. 2.4: Часть блок-схемы, определяющая f_j и f_{jbody}.

При этом для цикла по j из части блок-схемы, представленной на рисунке 2.4, будем иметь:

$$f_j = 1 + 3 + \frac{1 + (size - 1)}{2}(1 + 3 + f_{jbody}) \tag{2.4}$$

где f_{jbody} - это трудоёмкость тела цикла, которая определяется выражением:

$$f_{jbody} = 5 + \begin{cases} 0, & \textbf{лучший случай} \\ 9, & \textbf{худший случай} \end{cases} \tag{2.5}$$

Для цикла по i из части блок-схемы, представленной на рисунке 2.5, будем иметь:

$$f_i = 1 + 2 + (size - 1)f_j \tag{2.6}$$

Как итог, имеем в лучшем случае:

$$f = f_i = 3 + (size - 1)(4 + \frac{size}{2}(4 + 5) \tag{2.7}$$

$$f = -1 + 4size + \frac{9}{2}size^2 - \frac{9}{2}size \approx \frac{9}{2}size^2 = O(size^2) \tag{2.8}$$

В худшем случае:

$$f = f_i = 3 + (size - 1)(4 + \frac{size}{2}(4 + 14) \tag{2.9}$$

$$f = -1 + 4size + 9size^2 - 9size \approx 9size^2 = O(size^2) \tag{2.10}$$

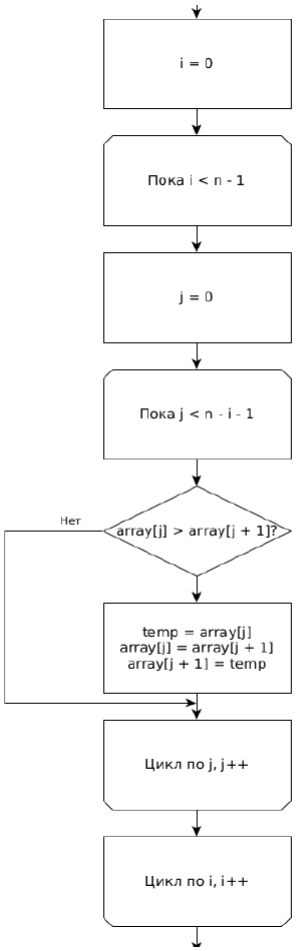

Рис. 2.5: Часть блок-схемы, определяющая f_i.

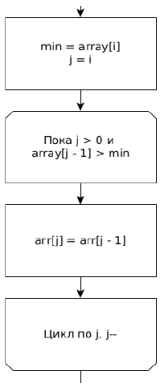

Рис. 2.6: Часть блок-схемы, определяющая f_j и f_{jbody}.

2.5.3 Алгоритм сортировки вставками

Трудоёмкость алгоритма сортировки вставками будет включать включать в себя:

- цикл по $i \in [1..size]$;
- цикл по $j \in [i..0]$, включённый в цикл по i.

При этом для цикла по j из части блок-схемы, представленной на рисунке 2.6, будем иметь:

$$f_j = 3 + 4 + \frac{size}{2}\begin{cases} 4, & \textbf{лучший случай} \\ 1 + 4 + f_{jbody}, & \textbf{худший случай} \end{cases} \tag{2.11}$$

где f_{jbody} - это трудоёмкость тела цикла, которая определяется выражением:

$$f_{jbody} = 4 \tag{2.12}$$

Для цикла по i из части блок-схемы, представленной на рисунке 2.7, будем иметь:

$$f_i = 1 + 1 + (size - 1)(1 + 1 + f_j) \tag{2.13}$$

Как итог, имеем в лучшем случае:

$$f = f_i = 2 + (size - 1)(9 + 2size) \tag{2.14}$$

$$f = 2size^2 + 7size - 7 \approx 2size^2 = O(size^2) \tag{2.15}$$

В худшем случае:

$$f = f_i = 3 + (size - 1)(4 + \frac{size}{2}(4 + 14) \tag{2.16}$$

$$f = -1 + 4size + 9size^2 - 9size \approx 9size^2 = O(size^2) \tag{2.17}$$

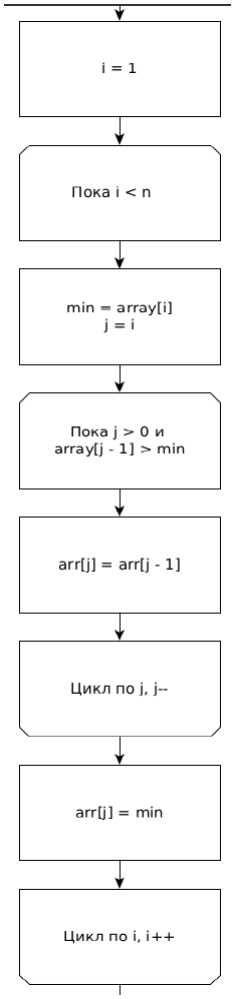

Рис. 2.7: Часть блок-схемы, определяющая f_i.

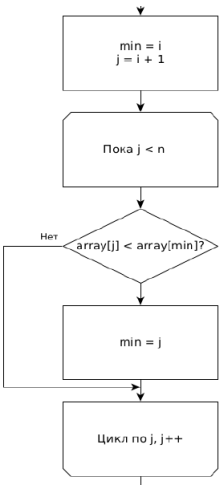

Рис. 2.8: Часть блок-схемы, определяющая f_j и f_{jbody}.

2.5.4 Алгоритм сортировки выбором

Трудоёмкость алгоритма сортировки выбором будет включать включать в себя:

- цикл по $i \in [0..size - 1]$;
- цикл по $j \in [i + 1..size]$, включённый в цикл по i.

При этом для цикла по j из части блок-схемы, представленной на рисунке 2.8, будем иметь:

$$f_j = 3 + 1 + \frac{size}{2}(1 + 1 + f_{jbody}) \tag{2.18}$$

где f_{jbody} - это трудоёмкость тела цикла, которая определяется выражением:

$$f_{jbody} = \begin{cases} 3, & \textbf{лучший случай} \\ 4, & \textbf{худший случай} \end{cases} \tag{2.19}$$

Для цикла по i из части блок-схемы, представленной на рисунке 2.9, будем иметь:

$$f_i = 1 + 1 + (size - 1)(1 + 1 + 2 + f_j) \tag{2.20}$$

Как итог, имеем в лучшем случае:

$$f = f_i = 2 + (size - 1)(4 + 4 + \frac{size}{2}(1 + 1 + 3)) \tag{2.21}$$

$$f = -6 + \frac{11}{2}size + \frac{5}{2}size^2 \approx \frac{5}{2}size^2 = O(size^2) \tag{2.22}$$

В худшем случае:

$$f = f_i = 2 + (size - 1)(4 + 4 + \frac{size}{2}(1 + 1 + 4)) \tag{2.23}$$

$$f = -6 + 5size + 3size^2 \approx 3size^2 = O(size^2) \tag{2.24}$$

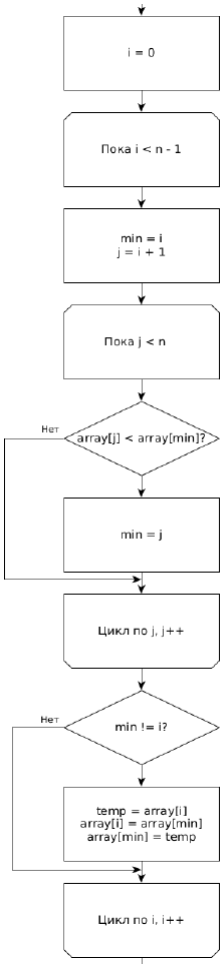

Рис. 2.9: Часть блок-схемы, определяющая f_i.

Вывод

На основе теоретичесих данных, полученных из аналитического раздела, были построены схемы рассматриваемых алгоритмов сортировок, оценены их трудоёмкости в лучшем и худшем случаях.

3 | Технологическая часть

3.1 Требования к программному обеспечению

Программное обеспечение принимает на вход массив целочисленных значений.

В результате выполнения программы должен быть получен упорядоченный по возрастанию массив.

3.2 Средства реализации программного обеспечения

При написании программного продукта был использован язык программирования Kotlin [2].

Данный выбор обусловлен следующими факторами:

- Возможность портирования алгоритмов для работы с Android;
- Возможность часто употреблять выражение "Ко-ко-котлин"или "Котик";
- Большое количество справочной литературы, связанной с ЯП Java.

Для тестирования производительности реализаций алгоритмов использовалась утилита measureTimedValue.

При написаннии программного продукта использовалась среда разработки IntelliJ IDEA.

Данный выбор обусловлен тем, что язык программирования Kotlin - это разработка компании JetBrains, поставляющей данную среду разработки.

3.3 Листинг кода

В листингах 3.1 - 3.3 предоставлены реализации рассматриваемых алгоритмов.

Листинг 3.1: Функция реализации алгоритма сортировки пузырьком

```
fun bubbleSort(arr: IntArray)
{
    var i = 0;
    while (i < arr.size - 1)
    {
        var j = 0;
        while (j < arr.size - 1 - i)
        {
            if (arr[j] > arr[j + 1])
                arr[j + 1] = arr[j].also { arr[j] = arr[j + 1] }
            j++
        }
        i++
    }
}
```

Листинг 3.2: Функция реализации алгоритма сортировки вставками

```
fun insertionSort(arr: IntArray)
{
    var i = 1;
    while (i < arr.size)
    {
        var min = arr[i]
        var j = i;
        while (j > 0 && arr[j - 1] > min)
        {
            arr[j] = arr[j - 1]
            j--
        }
        arr[j] = min
        i++
    }
}
```

Листинг 3.3: Функция реализации алгоритма сортировки выбором

```
fun selectionSort(arr: IntArray)
```

```
{
    var i = 0;
    while (i < arr.size - 1)
    {
        var j = i + 1
        var min = i
        while (j < arr.size)
        {
            if (arr[j] < arr[min])
                min = j
            j++
        }
        if (min != i)
            arr[i] = arr[min].also { arr[min] = arr[i] }
        i++
    }
}
```

3.4 Тестирование программного продукта

В таблице 3.1 приведены тесты для функций, реализующих алгоритм сортировки пузырьком, вставками и выбором. Тесты пройдены успешно.

Начальный массив	Ожидаемый результат	Полученный результат
$[5, 4, 3, 2, 1]$	$[1, 2, 3, 4, 5]$	$[1, 2, 3, 4, 5]$
$[0, 0, 0]$	$[0, 0, 0]$	$[0, 0, 0]$
$[1, 2, 3]$	$[1, 2, 3]$	$[1, 2, 3]$
$[-8, 2, 3, -2]$	$[-8, -2, 2, 3]$	$[-8, -2, 2, 3]$
$[]$	$[]$	$[]$

Таблица 3.1: Тестирование функций

Вывод

Спроектированные алгоритмы сортировок были реализованы и протестированы.

4 | Исследовательская часть

4.1 Пример работы программного обеспечения

Ниже на рисунках 4.1- 4.2 предоставлены примеры работы каждого из алгоритмов на введённых пользователем данных.

```
Number of elements:
10
Array elements: -666 777 -1337 5051 322 228 0 1234567890 505 314505
Ready in bubble sort:
-1337 -666 0 228 322 505 777 5051 314505 1234567890
Ready in selection sort:
-1337 -666 0 228 322 505 777 5051 314505 1234567890
Ready in insertion sort:
-1337 -666 0 228 322 505 777 5051 314505 1234567890

Process finished with exit code 0
```

Рис. 4.1: Пример работы ПО.

```
Number of elements:
20
Array elements: 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
Ready in bubble sort:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Ready in selection sort:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Ready in insertion sort:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Process finished with exit code 0
```

Рис. 4.2: Пример работы ПО.

4.2 Технические характеристики

Технические характеристики ЭВМ, на котором выполнялись исследования:

- ОС: Manjaro Linux 20.1.1 Mikah
- Оперативная память: 16 Гб
- Процессор: Intel Core i7-10510U

  При проведении замеров времени ноутбук был подключен к сети электропитания.

4.3 Время выполнения алгоритмов

Алгоритмы тестировались на данных, сгенерированных случайным образом один раз.

Результаты замеров времени приведены в таблице 4.1. На рисунках 4.3 и 4.4 приведены графики зависимостей времени работы алгоритмов от количества строк и столбцов матриц (в чётном и нечётном вариантах). В таблице СА - Сортировка Пузырьком, СВст - Алгоритм Копперсмита-Винограда, УКВ - Улучшенный Алгоритм Копперсмита-Винограда.

Таблица 4.1: Замеры времени для квадратных матриц различных размеров

Размер матрицы	КА	КВ	УКВ
100	2148121	2302331	1921005
101	2312114	2623891	2032155
200	17350292	22592034	1425033
201	20247410	21694235	17554153
300	68920554	73453362	57000923
301	75166547	77955778	64421195
400	211301483	205981760	172968826
401	227614782	218087162	171881527
500	367822853	351730341	340284336
501	364368768	362588416	358108198
600	678478122	658012453	625149992
601	672846913	671159157	647843183

Вывод

При сравнении результатов замеров времени заметно, что скорость работы классического алгоритма однозначно отстаёт от скорости работы улучшенного Алгоритма Копперсмита-Винограда. Уже на 600 элементах улучшенный алгоритм Копперсмита-Винограда работает быстрее классического на $\approx 8\%$. При нечётном количестве строк и столбцов матриц улучшенный алгоритм способен быть медленнее $\approx 4\%$, при факте того, что классический алгоритм похожей динамики не имеет. Обычный алгоритм Копперсмита-Винограда начинает выигрывать по скорости классический только по достижению 300 строк и столбцов в матрице, при факте того, что в случае матрицы с нечётной размерностью он всё ещё будет проигрывать. При размерности 600 он будет выгрывать у классической реализации на $\approx 3\%$. В случае матриц размера меньше 400 на 400 его использование не будет целесообразным.

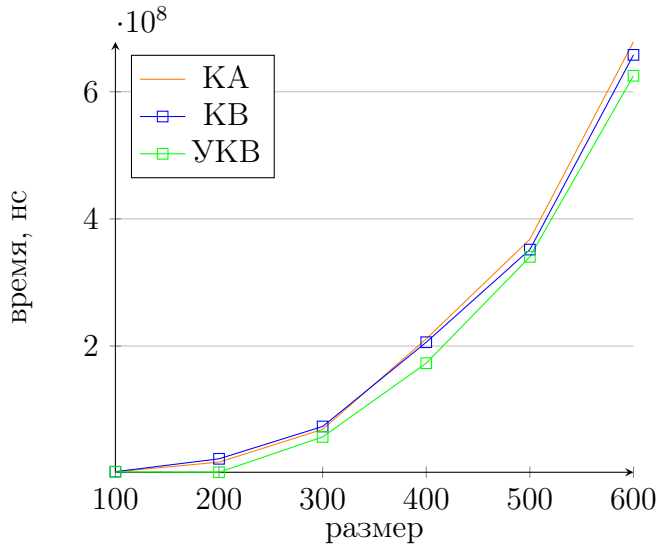

Рис. 4.3: Зависимость времени работы от размера матриц (чётные значения размерностей)

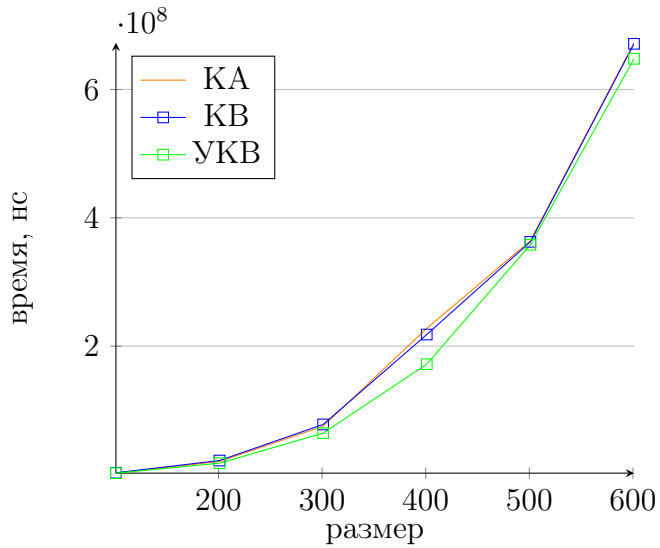

Рис. 4.4: Зависимость времени работы от размера матриц (нечётные значения размерностей)

Заключение

В ходе выполнения лабораторной работы:

- были изучены алгоритмы умножения матриц: классический, Копперсмита-Винограда и улучшенный Копперсмита-Винограда;
- были реализованы алгоритмы умножения матриц: классический, Копперсмита-Винограда и улучшенный Копперсмита-Винограда;
- был произведён анализ трудоёмкости указанных алгоритмов на основе теоретических расчётов и выбранной модели вычислений;
- был выполнен сравнительный анализ производительности алгоритмов на основе полученных экспериментальных данных;
- был подготовлен отчёт по проделанной работе;
- были получены практические навыки реализации алгоритмов на ЯП Kotlin.

Исследования показали, что использование алгоритма Копперсмита-Винограда способно оправдать себя только в случае матриц, размерность которых не менее 400. При этом выигрыш будет составлять $\approx 0.2\%$ только в случае чётной размерности. Реализация улучшенного алгоритма Копперсмита-Винограда показывает результаты быстрее классического алгоритма уже при размерности матрицы 100. Чем больше элементов в матрице, тем заметнее разница во времени работы этих двух алгоритмов. При размерности матрицы 600 модифицированный алгоритм Копперсмита-Винограда показывает себя лучше классического алгоритма на $\approx 8\%$.

Литература

[1] Кнут Дональд. Сортировка и поиск. Вильямс, 2000. Т. 3 из *Искусство программирования*. с. 834.

[2] Kotlin language specification [Электронный ресурс]. Режим доступа: `https://kotlinlang.org/spec/introduction.html` (дата обращения 09.10.2020.