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 №4 по курсу "Анализ алгоритмов"

Тема Реализация параллельного алгоритма Копперсмита-Винограда

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 параллельных вычислений;
2. изучение последовательного и двух параллельных реализаций алгоритмов Винограда;
3. реализация последовательного и двух параллельных алгоритмов Винограда;
4. проведение сравнительного анализа временных характеристик реализованных алгоритмов на основе экспериментальных данных;
5. подготовка отчёта по лабораторной работе;
6. получение практических навыков реализации алгоритмов на ЯП Nim.

Определение

Многопоточность — это способность центрального процессора (CPU) или одного ядра в многоядерном процессоре одновременно выполнять несколько процессов или потоков, соответствующим образом поддерживаемых операционной системой. Этот подход отличается от многопроцессорности, так как многопоточность процессов и потоков совместно использует ресурсы одного или нескольких ядер: вычислительных блоков, кэш-памяти ЦПУ или буфера перевода с преобразованием (TLB).

В тех случаях, когда многопроцессорные системы включают в себя несколько полных блоков обработки, многопоточность направлена на

максимизацию использования ресурсов одного ядра, используя параллелизм на уровне потоков, а также на уровне инструкций. Поскольку эти два метода являются взаимодополняющими, их иногда объединяют в системах с несколькими многопоточными ЦП и в ЦП с несколькими многопоточными ядрами.

Многопоточная парадигма стала более популярной с конца 1990-х годов, поскольку усилия по дальнейшему использованию параллелизма на уровне инструкций застопорились. Смысл многопоточности — квазимногозадачность на уровне одного исполняемого процесса. Значит, все потоки процесса помимо общего адресного пространства имеют и общие дескрипторы файлов. Выполняющийся процесс имеет как минимум один (главный) поток.

1 Аналитическая часть

1.1 Алгоритм Копперсмита-Винограда умножения матриц

Пусть даны две прямоугольные матрицы

$$A_{lm} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{l1} & a_{l2} & \dots & a_{lm} \end{pmatrix}, \quad B_{mn} = \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \dots & b_{mn} \end{pmatrix}, \tag{1.1}$$

тогда матрица C

$$C_{ln} = \begin{pmatrix} c_{11} & c_{12} & \dots & c_{1n} \\ c_{21} & c_{22} & \dots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{l1} & c_{l2} & \dots & c_{ln} \end{pmatrix}, \tag{1.2}$$

где

$$c_{ij} = \sum_{r=1}^{m} a_{ir} b_{rj} \quad (i = \overline{1,l}; j = \overline{1,n}) \tag{1.3}$$

будет называться произведением матриц A и B.

Если посмотреть на результат умножения двух матриц, то видно, что каждый элемент в нем представляет собой скалярное произведение соответствующих строки и столбца исходных матриц. Можно заметить также, что такое умножение допускает предварительную обработку, позволяющую часть работы выполнить заранее.

Рассмотрим два вектора $V = (v_1, v_2, v_3, v_4)$ и $W = (w_1, w_2, w_3, w_4)$. Их скалярное произведение равно: $V \cdot W = v_1w_1 + v_2w_2 + v_3w_3 + v_4w_4$, что эквивалентно (1.4):

$$V \cdot W = (v_1+w_2)(v_2+w_1)+(v_3+w_4)(v_4+w_3)-v_1v_2-v_3v_4-w_1w_2-w_3w_4. \quad (1.4)$$

Несмотря на то, что второе выражение требует вычисления большего количества операций, чем стандартный алгоритм: вместо четырех умножений - шесть, а вместо трех сложений - десять, выражение в правой части последнего равенства допускает предварительную обработку: его части можно вычислить заранее и запомнить для каждой строки первой матрицы и для каждого столбца второй, что позволит для каждого элемента выполнять лишь два умножения и пять сложений, складывая затем только лишь с 2 предварительно посчитанными суммами соседних элементов текущих строк и столбцов [1]. Из-за того, что операция сложения быстрее операции умножения в ЭВМ, на практике алгоритм должен работать быстрее стандартного [2].

Важно заметить, что каждый элемент матрицы C вычисляется независимо от других, а матрицы A и B не изменяются, поэтому для распараллеливания алгоритма будет достаточно распределить элементы каждой строки матрицы C между потоками.

Вывод

Был рассмотрен алгоритм умножения матриц Копперсмита-Винограда. В данной работе стоит задача реализации данного алгоритма в последовательном и параллельных видах. Независимость вычислений элементов результата даёт возможность реализовать несколько параллельных вариантов исполнения данного алгоритма.

2 | Конструкторская часть

2.1 Блок-схема алгоритма Копперсмита-Винограда

Блок-схема алгоритма Копперсмита-Винограда предоставлена на рисунках 2.1-2.3.

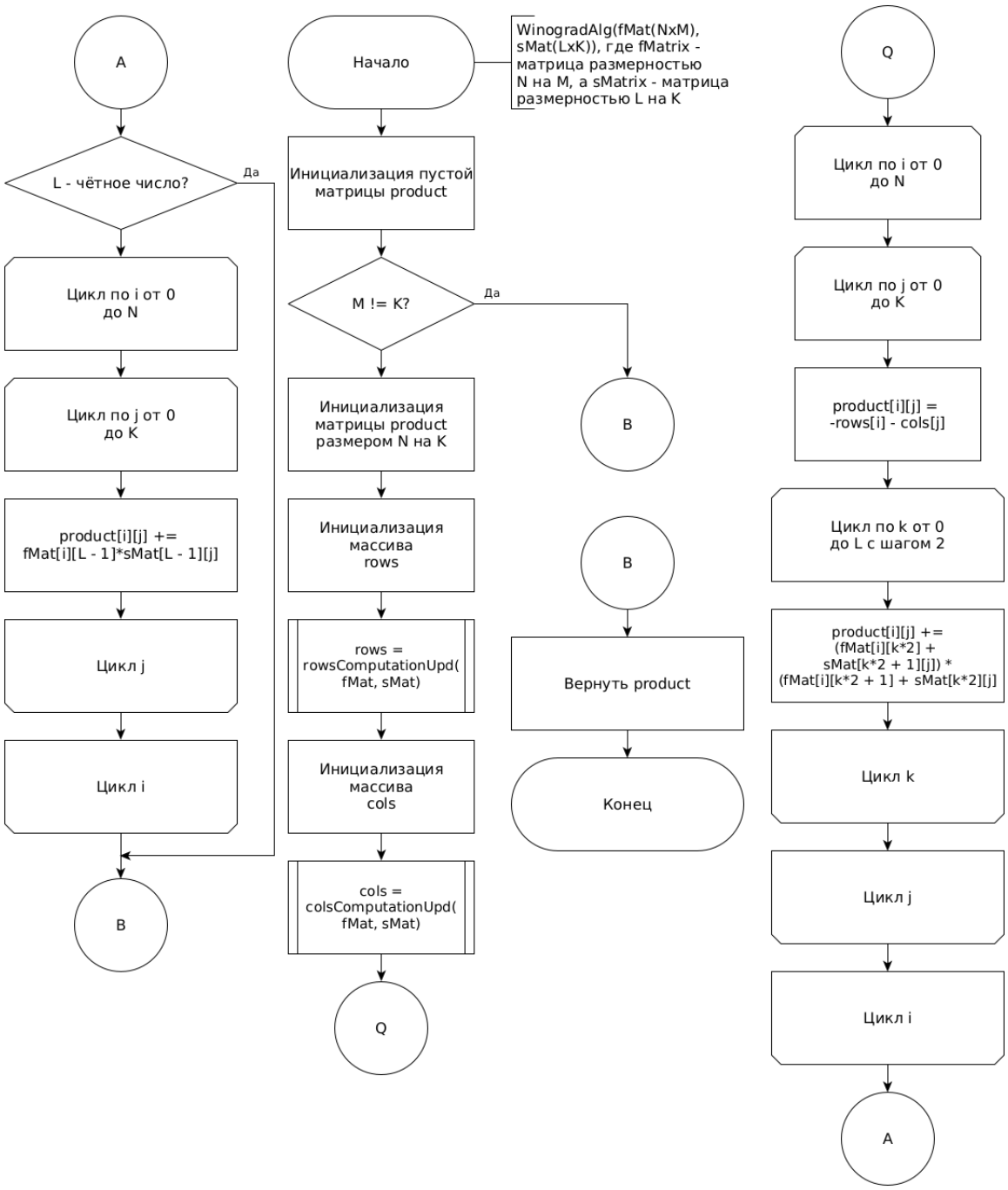

Рис. 2.1: Блок-схема алгоритма Копперсмита-Винограда.

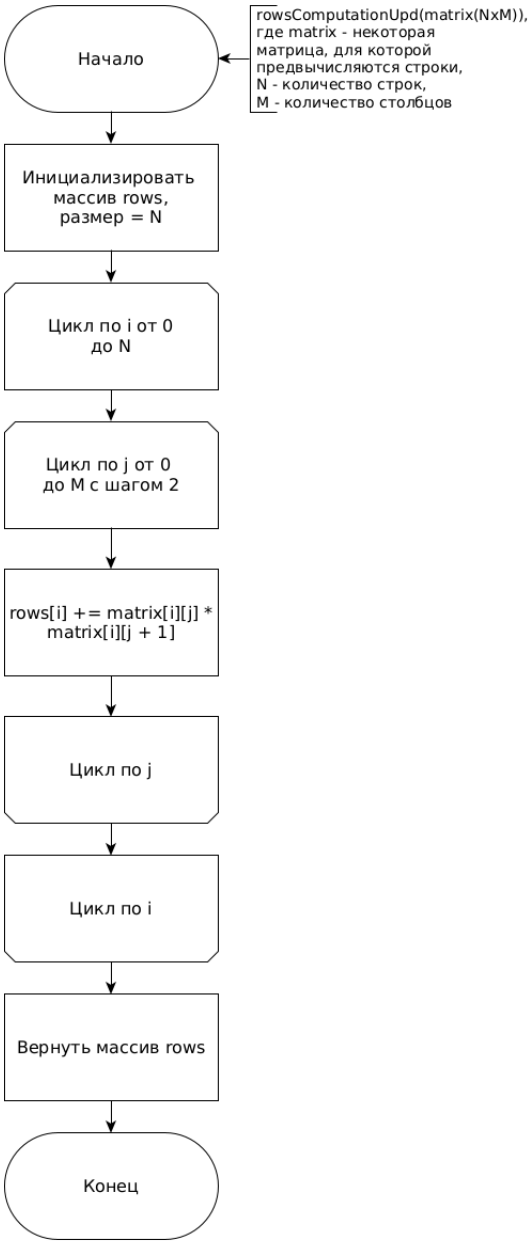

Рис. 2.2: Блок-схема предрасчёта строк для алгоритма Копперсмита-Винограда.

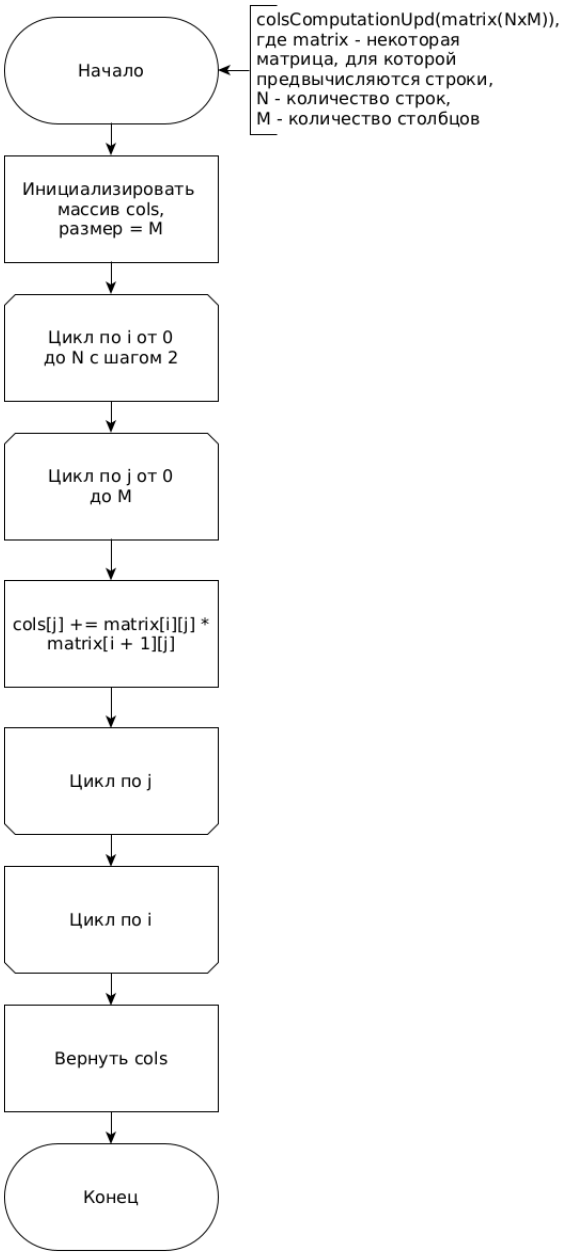

Рис. 2.3: Блок-схема предрасчёта столбцов для алгоритма Копперсмита-Винограда.

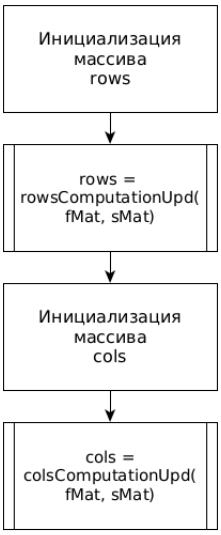

Рис. 2.4: Часть блок-схемы, в которой происходит вызов подпрограмм предрасчёта строк и столбцов.

2.2 Параллельная реализация алгоритма Копперсмита-Винограда

Для распараллеливания алгоритма требуется определить, какие из частей алгоритма могут быть выполнены вне зависимости друг от друга.

Часть блок-схемы, отвечающая за предрасчёт строк и столбцов (рисунок 2.4), однозначно может быть распараллелен. Но до завершения двух этих подпрограмм к выполнению следующих этапов приступать нельзя, так как вычисленные массивы будут использоваться далее. В данной части будет использоваться полное выделенное количество потоков последовательно - первоначально для вычисления строк, а далее - для вычисления столбцов.

Непосредственное вычисление значения каждого элемента матрицы, показанное на рисунке 2.5, может быть также поэлементно распараллелено. Однако тут важно заметить, что часть данного этапа, показанная на рисунке 2.6, и часть, показанная на рисунке 2.7, могут быть выполнены как раздельно, так и совместно при обработке каждого из элементов. Важно заметить, что, при раздельном выполнении, потребуется создавать новые потоки, что является достаточно затратным по времени действием.

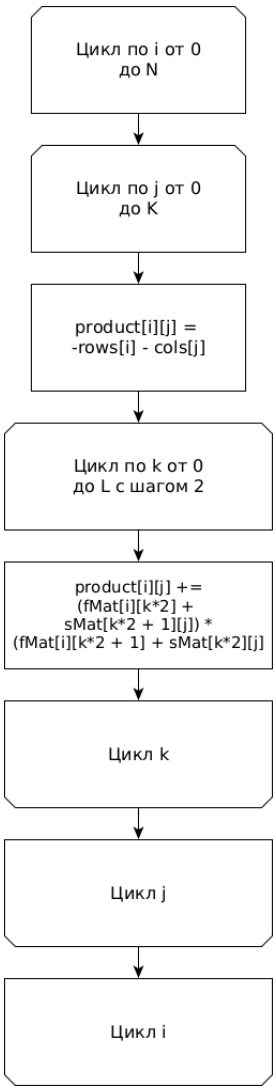

Рис. 2.5: Часть блок-схемы, в которой происходит расчёт значения каждого элемента матрицы.

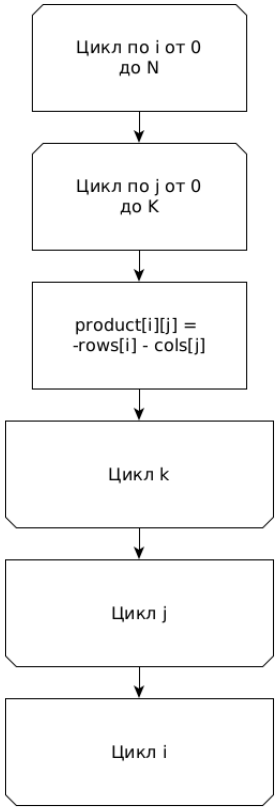

Рис. 2.6: Первая часть рассматриваемого этапа.

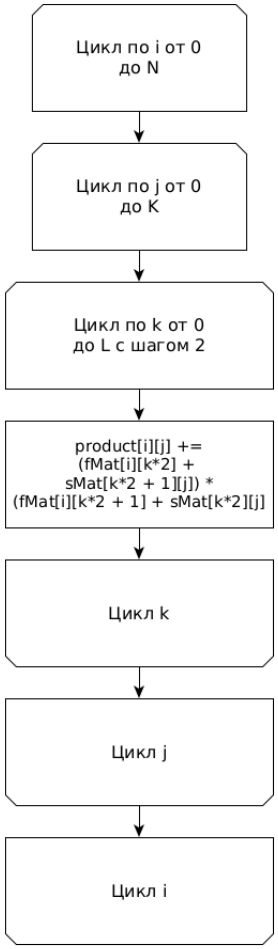

Рис. 2.7: Вторая часть рассматриваемого этапа.

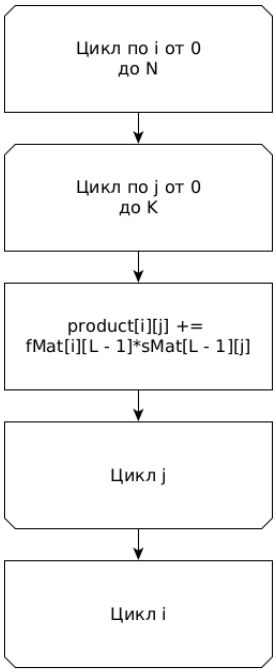

Рис. 2.8: Довычисление значений итоговой матрицы при нечётной размерности оной.

Довычисление значений в случае нечётной размерности итоговой матрицы, показанное на рисунке 2.8 в каждой итерации цикла требует обращения к матрице, что при распараллеливании приведёт к большому числу блокирований разделяемой памяти и будет неэффективно. Поэтому данный этап останется без изменений.

2.3 Схема параллельной реализации алгоритма Копперсмита-Винограда с разделением этапа вычисления элементов матрицы

На рисунке ?? предоставлена схема с параллельным выполнением двух частей этапа вычисления элементов матрицы.

2.4 Схема параллельной реализации алгоритма Копперсмита-Винограда без разделения этапа вычисления элементов матрицы

На рисунке ?? предоставлена схема с последовательным выполнением двух частей этапа вычисления элементов матрицы.

Вывод

На основе теоретических данных, полученных из аналитического раздела, была построена схема алгоритма Копперсмита-Винограда. Были предложены 2 схемы параллельной реализации рассматриваемого алгоритма.

3 | Технологическая часть

3.1 Требования к программному обеспечению

- входные данные - две матрицы размерностью MxN и KxL;
- выходные данные - результат умножения двух переданных матриц.

3.2 Средства реализации программного обеспечения

При написании программного продукта был использован язык программирования Nim [?].

Данный выбор обусловлен следующими факторами:

- Компилируемость языка в C, C++, Objective C и JavaScript;
- Синтаксис языка близок к синтаксису ЯП Python;

Для тестирования производительности реализаций алгоритмов использовалась библиотека times.

При написании программного продукта использовалась среда разработки IntelliJ IDEA.

Данный выбор обусловлен тем, что данная среда разработки имеет плагин поддержки языка программирования Nim.

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
```

```
8           while (j < arr.size)
9           {
10              if (arr[j] < arr[min])
11                  min = j
12              j++
13          }
14          if (min != i)
15              arr[i] = arr[min].also { arr[min] = arr[i] }
16          i++
17      }
18  }
```

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.

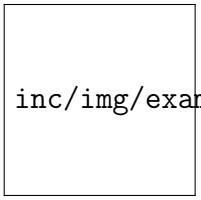

Рис. 4.1: Пример работы ПО.

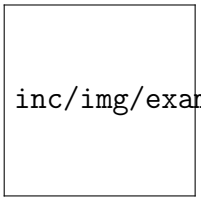

Рис. 4.2: Пример работы ПО.

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 таблицах 4.1 - 4.3. На рисунках 4.3 и 4.5 приведены графики зависимостей времени работы алгоритмов от количества элементов в случаях передачи упорядоченного, неупорядоченного и упорядоченного в обратном порядке массива. В таблице СП - Сортировка Пузырьком, СВст - Сортировка Вставками, СВыб - Сортировка выбором.

Таблица 4.1: Замеры времени для упорядоченных массивов

Размер массива	СП	СВст	СВыб
100	1818544	1630961	1826308
200	2182084	1647635	2144446
300	2836575	1640207	2530081
400	3666706	1640022	2924786
500	4561938	1645407	3594866
1000	12282615	1654109	8634889
1500	25296835	1662390	17374004
2000	43069012	1700609	29104145
3000	91881622	1810398	61997393
4000	165458964	1823571	105764286
5000	248940636	1766772	164210924

Таблица 4.2: Замеры времени для упорядоченных в обратном порядке массивов

Размер массива	СП	СВст	СВыб
100	1902381	1770645	1712426
200	2690147	2035881	1948340
300	3973067	2809635	2299634
400	6108805	3078558	2837866
500	8344909	3876786	3483199
1000	28119897	10741734	9179446
1500	59508253	21659401	17961391
2000	102249552	37367527	30242327
3000	218854972	78513340	65218978
4000	379629421	131840967	114500414
5000	592686305	197947307	173933001

Таблица 4.3: Замеры времени для неупорядоченных массивов

Размер массива	СП	СВст	СВыб
100	1592950	1498909	1460379
200	2200640	1562519	1680307
300	3280409	1808384	2038358
400	4728961	2184299	2558087
500	6573267	2566161	3166091
1000	21195842	5769483	8411547
1500	45146080	11233626	17161009
2000	79173129	18556919	28673064
3000	166840980	38942468	64444295
4000	280142359	66181509	102043152
5000	449032194	99391239	172713187

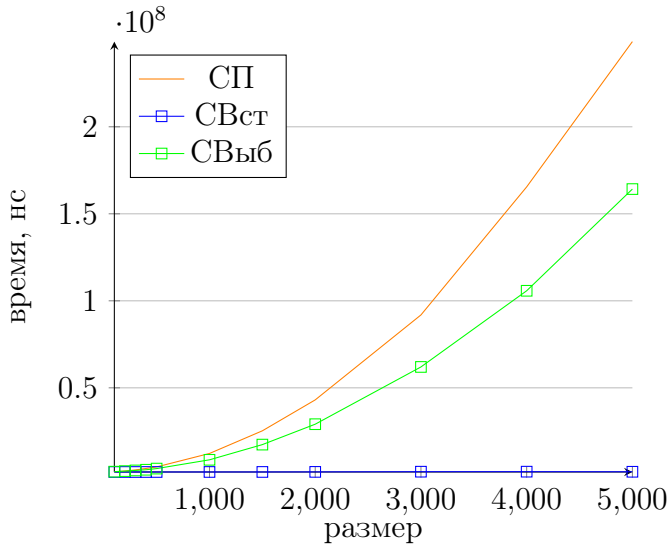

Рис. 4.3: Зависимость времени работы от размера отсортированных массивов

Вывод

При сравнении результатов замеров по времени видно, что выводы в каждом из случаев неоднозначны.

В случае уже отсортированных массивов, однозначно лучше всех показывает себя сортировка вставками. Связано это с особенностями реализации алгоритма, зависимость в текущих обстоятельствах будет, теоретически, линейной. Поэтому целесообразно указать на факт того, что сортировка пузырьком уже на 3000 элементов будет работать медленнее, чем сортировка выбором на $\approx 50.8\%$, при этом с увеличением количества элементов массива, это значение будет расти.

В случае упорядоченного в обратном порядке массива, что можно назвать наихудшим случаем, лучше всех показывает себя сортировка выбором. При этом, разница между временем работы сортировки выбором и сортировки вставками на 4000 элементов массива будет равна $\approx 15\%$. Но наибольшая разница, которая заключается между временем работы сортировки выбором и сортировки пузырьком на 5000 элементов: $\approx 340\%$.

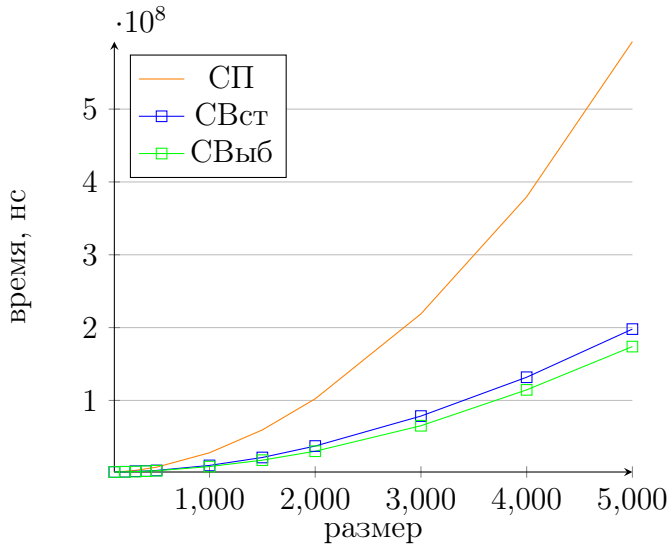

Рис. 4.4: Зависимость времени работы от размера отсортированных в обратном порядке массивов

Ситуация несколько изменится при рассмотрении скорости работы алгоритмов на массивах чисел, сгенерированных случайно. Здесь можно будет увидеть преобладание алгоритма сортировки выбором над алгоритмом сортировки вставками. На значениях величины массива, равных 4000, разница в скорости обработки будет составлять $\approx 65\%$. Наибольшая выявленная разница в скорости работы алгоритма вставками и алгоритма сортировки пузырьком будет составлять $\approx 260\%$.

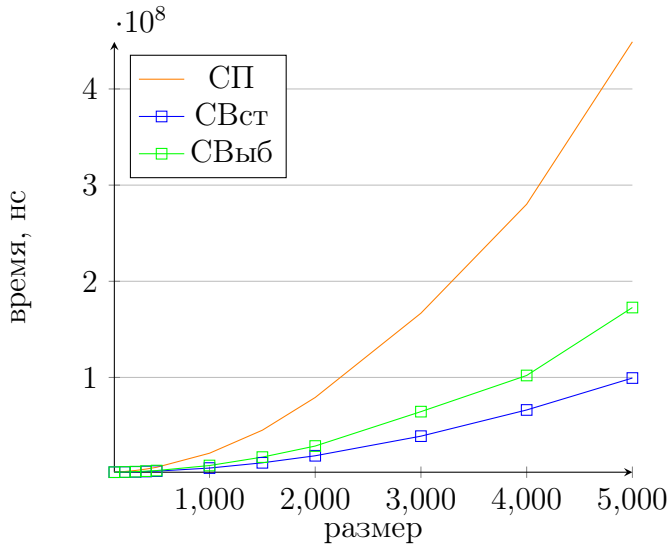

Рис. 4.5: Зависимость времени работы от размера массивов, заданных случайным образом

Заключение

В ходе выполнения лабораторной работы:

1. были изучены алгоритмы сортировки пузырьком, вставками и выбором;
2. были реализованы алгоритмы сортировки пузырьком, вставками и выбором;
3. был проведён сравнительный анализа трудоёмкости алгоритмов на основе теоретических расчетов и выбранной модели вычислений;
4. был проведён сравнительный анализ алгоритмов на основе экспериментальных данных;
5. был подготовлен отчёт по лабораторной работе;
6. были получены практические навыки реализации алгоритмов на ЯП Kotlin.

Исследования показали, что хуже всего себя показала реализация алгоритма сортировки пузырьком.

При сортировке уже отсортированного массива "пузырёк"будет работать на $\approx 66\%$ медленнее при сортировке 5000 элементов, чем алгоритм сортировки выбором. Но наилучшим вариантом в данном случае будет выбор алгоритма сортировки вставками, так как возрастание скорости его работы в данном случае линейно.

В случае отсортированного в обратном порядке массива лучше всего себя показывает сортировка вставками, она в ≈ 1.38 раз быстрее алгоритма сортировки выбором. Наибольшая разница, которая заключается между временем работы сортировки выбором и сортировки пузырьком на 5000 элементов: $\approx 340\%$.

Но в случае неупорядоченных массивов лучше всего себя показывает сортировка вставками, она в ≈ 1.74 раза быстрее, чем сортировка выбором уже на 5000 элементов. Наибольшая выявленная разница в скорости работы алгоритма вставками и алгоритма сортировки пузырьком будет составлять $\approx 260\%$

Литература

[1] Coppersmith D., Winograd S. Matrix multiplication via arithmetic progressions // Journal of Symbolic Computation. 1990. no. 9. P. 251–280.

[2] Погорелов Дмитрий Александрович Таразанов Артемий Михайлович Волкова Лилия Леонидовна. Оптимизация классического алгоритма Винограда для перемножения матриц // Журнал №1. 2019. Т. 49.

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