**Реферат**

Отчёт 24 c., 17 рис., 4 табл., 24 источн.

**Ключевые слова**: *неадаптивные сортировки; адаптивные сортировки; сортирующие сети; сортировка; слой сети; глубина сети; размер сети; операция сравнения-обмена; сортировка Бэтчера; компараторы, сети АКС, разделители, сортировка «зиг-загом»*.

В отчете представлены результаты курсовой работы на тему «Исследование неадаптивных сортировок», выполненной на основе учебного плана подготовки бакалавров по направлению 09.03.04 «Программная инженерия» и утвержденной академическим руководителем темы курсового проекта.

**Объект исследования** – сортировки.

**Предмет исследования** – неадаптивные сортировки.

**Цели исследования** – сравнение алгоритмов адаптивных и неадаптивных сортировок.

**Задачи исследования:**

* изучение наиболее известных неадаптивных сортировок;
* реализация самых известных неадаптивных сортировок;
* разработка программы с открытым исходным кодом, позволяющая провести эксперимент сравнения времени работы алгоритмов адаптивных и неадаптивных сортировок;
* провести эксперимент, позволяющий сравнить время работы адаптивных и неадаптивных сортировок.

**Методы исследования**:

* изучение монографий, публикаций и статей;
* сравнительный анализ.

**Научная новизна** работы состоит в следующем:

* написана открытая библиотека, содержащая в себе реализации самых известных неадаптивных сортировок;
* получен сравнительный анализа времён работы адаптивных и неадаптивных сортировок.

**Достоверность научных** **результатов** подтверждена результатами экспериментальных исследований с разработанной программой, содержащей в себе имплементации неадаптивных сортировок.

**Практическая значимость**. Результаты работы могут быть использованы при создании промышленных систем, а также в существующих системах с целью оптимизации или решения задач, связанных с сортировкой объектов.

**Результаты работы**:

* изучены самые известные алгоритмы неадаптивных сортировок;
* разработана программа для сравнения времени работы адаптивных и неадаптивных сортировок;
* проведён вычислительный эксперимент, позволяющий получить сравнение времён работ адаптивных и неадаптивных сортировок на различных объёмах входных данных;
* реализована открытая библиотека, содержащая в себе реализации самых известных неадаптивных сортировок.

Оглавление

[**1.** **Введение** 5](#_Toc70031207)

[**2.** **Неадаптивные сортировки** 5](#_Toc70031208)

[**2.1.** **Сортирующие сети** 5](#_Toc70031209)

[**2.2.** **Сортировка «пузырьком»** 6](#_Toc70031210)

[**2.3.** **Битонная сортировка Бэтчера** 7](#_Toc70031211)

[**2.4.** **Разделители** 10](#_Toc70031212)

[**2.5.** **Сети АКС** 10](#_Toc70031213)

[**2.6.** **Сортировка зиг-загом** 12](#_Toc70031214)

[**2.7.** **Сортировка произвольного объёма данных** 15](#_Toc70031215)

[**3.** **Адаптивные сортировки** 15](#_Toc70031216)

[**3.1.** **Сортировка слиянием** 15](#_Toc70031217)

[**3.2.** **Быстрая сортировка Хоара** 16](#_Toc70031218)

[**3.3.1** **Пирамидальная сортировка** 17](#_Toc70031219)

[**4.** **Экспериментальный комплекс** 18](#_Toc70031220)

[**4.1** **Подготовка к проведению экспериментов** 18](#_Toc70031221)

[**4.2** **Предварительная оценка времён работ алгоритмов** 19](#_Toc70031222)

[**4.3** **Описание первого эксперимента** 20](#_Toc70031223)

[**4.4** **Результаты первого эксперимента** 20](#_Toc70031224)

[**4.5** **Описание второго эксперимента** 20](#_Toc70031225)

[**4.6** **Результаты второго эксперимента** 21](#_Toc70031226)

[**5.** **Выводы** 21](#_Toc70031227)

[**6.** **Направления дальнейшей работы** 22](#_Toc70031228)

[**7.** **Термины и сокращения** 22](#_Toc70031229)

[**8.** **Список использованных источников** 23](#_Toc70031230)

# **Введение**

Задача сортировки вставала перед людьми ещё задолго до зарождения математики и информатики. Действительно, чтобы распределить ресурсы внутри племени или построить устойчивое сооружение, необходимо расположить объекты от меньшего к большему. Таким образом древнейшие предки (и даже некоторые животные, обладающие абстрактным мышлением) решали задачу сортировки, не зная чисел. И даже сейчас любой человек почти ежедневно сталкивается с необходимостью упорядочивания объектов -начиная от времени на часах и заканчивая укладкой продуктов в холодильник.

Задача сортировки объектов - центральная задача теории алгоритмов, которая широко применяется во всех сферах программирования, в частности - в промышленной разработке ПО.

Ввиду повсеместной применимости любые оптимизации и улучшения существующих алгоритмов сортировки имеют не только теоретическую, но и практическую ценность. Необходимость в сортировке данных возникает во всех научных и практических сферах, начиная от маркетинга (отсортировать продукты по релевантности для потребителя) и заканчивая высшей математикой (аппроксимационные алгоритмы для вычисления функций).

Неадаптивные сортировки позволяют эффективно работать с данными вне зависимости от их вида, перестановок и используемой ЭВМ, поэтому открывают широкие перспективы для внедрения.

Однако, на данный момент, этот вид сортировок изучен слабо и предпочтения отдаются адаптивным сортировкам.

Данная работа посвящена изучению самых популярных и используемых на данный момент неадаптивных сортировок с целью сравнить их с наиболее эффективными адаптивными сортировками.

# **Неадаптивные сортировки**

В данной главе будут рассмотрены самые известные неадаптивные сортировки, а также связанные с ними термины и понятия.

Предполагается, что в рамках данного реферата все данные, подающиеся на вход сортировкам, являются упорядочиваемыми, то есть они поддерживают операцию сравнения между собой.

* 1. **Сортирующие сети**

Понятие сортирующих сетей тесно связано с неадаптивными сортировками. Именно с помощью данной абстракции можно представить любую неадаптивную сортировку для фиксированного количества данных. Сеть сортировки состоит из «нитей» с данными и компараторов, соединяющих пары «нитей» и выполняющих операцию сравнения-обмена для них. Сеть называется сортирующей тогда и только тогда, когда она корректно сортирует все возможные входные данные. Предполагается, что данные по «нитям» идут одновременно. Когда на пути «нити» встречается компаратор, он выполняет операцию сравнения-обмена между ними, то есть, пусть на верхней нити лежало значение *x*, а на нижней значение *y*, тогда после выполнения компаратора на верхней будет идти значение *min(x, y)*, а на нижней *max(x, y)*. Соответственно такая пара значений сортируется между собой.

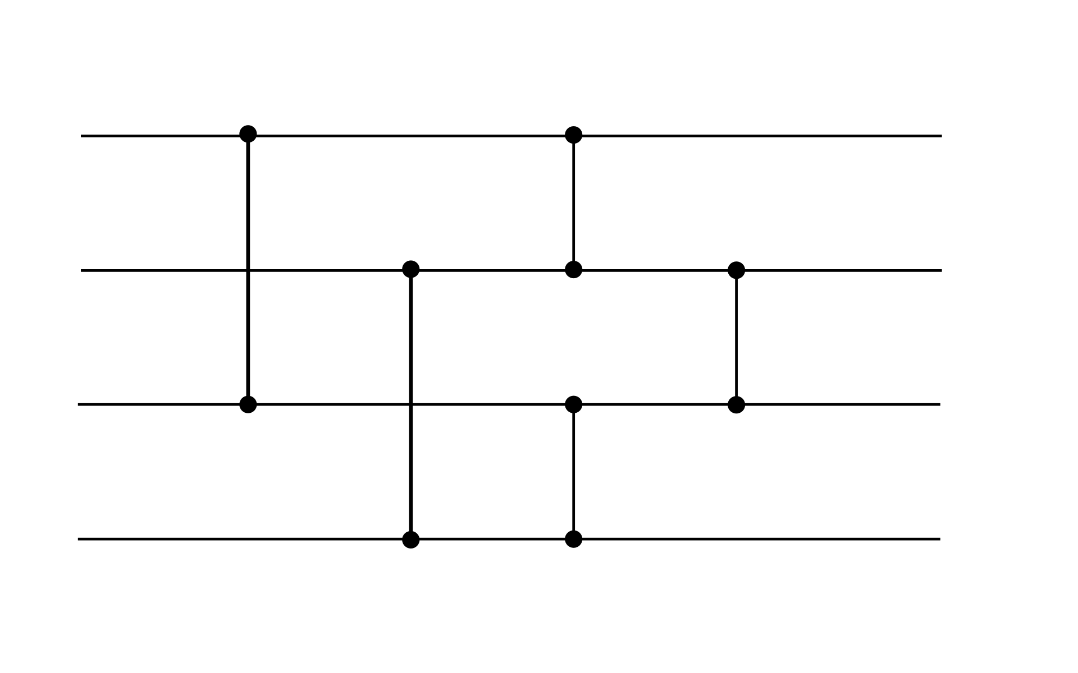


Рисунок 1. Простейшая сортирующая сеть для 4 элементов

Сортирующую сеть можно охарактеризовать при помощи двух параметров [3]:

* размер сортирующей сети, то есть, общее количество компараторов в сети;
* глубина сортирующей сети. Чтобы понять данную характеристику, будет необходимо ввести ряд понятий. Глубина «нити» - изначально все «нити» имеют глубину ноль. Пусть компаратор соединяет две «нити», глубина которых на данный момент *a* и *b*, тогда после исполнения компаратора «нити» будут иметь глубину *max(a, b) + 1*. Так как сортирующие сети являются ацикличными, то глубина «нити» на каждом этапе определяется однозначно. Подразумевается, что компараторы, после выполнения которых «нити» имеют одну и ту же глубину, «лежат» в одном слое. Тогда, глубина сортирующей сети – это максимум из глубин всех «нитей» после окончания работы сортирующей сети или общее количество слоёв компараторов. Менее формально, можно определить глубину сортирующей сети, как максимальное количество компараторов, которые может встретить на своём пути в сети какое-либо входное значение. Таким образом, если предположить, что у нас есть вычислительная машина, умеющая параллельно делать любое количество операций сравнения-обмена, то глубиной сети задаётся количество временных шагов, необходимых для её выполнения [5].

Например: сортирующая сеть на рисунке 1 имеет размер 5 и глубину сети 3.

* 1. **Сортировка «пузырьком»**

Простейшим примером неадаптивной сортировки является сортировка «пузырьком». Её алгоритм заключается в проходе по входным данным *N – 1* раз (где *N* – количество входных данных), каждый раз последовательно применяя к ним операцию сравнения-обмена. Таким образом, после первого прохода максимум будет установлен в позицию *N*, после второго прохода второй максимум будет установлен в позицию *N – 1* и так далее. Сортирующая сеть, соответствующая сортировке «пузырьком» для 6 элементов, представлена на рисунке 2.



Рисунок 2. Сортирующая сеть, соответствующая сортировке «пузырьком» для 6 элементов

Данная сортировка имеет асимптотику *O(n^2)* [1].

* 1. **Битонная сортировка Бэтчера**

Эта сортировка была предложена Кеном Бэтчером в 1968 году. Это одна из самых первых сортировок с возможностью распараллеливания операций и одна из самых первых сортирующих сетей для любого количества входных данных. Также стоит заметить, что несмотря на достаточно большое количество исследований, проведённых уже после открытия данной сортировки, сортировка Бэтчера всё ещё является самым применяемым алгоритмом неадаптивной сортировки в силу простоты распараллеливания сортировки и очень маленькой константы в асимптотике. Также сети слияния Бэтчера являются оптимальными [3], так что любая сеть сортировки с меньшим размером и глубиной может быть построена только в рамках подхода отличного от сортировки слиянием.

Для начала рассмотрим все необходимые понятия для того, чтобы построить сортирующую сеть Бэтчера, основанную на битонических последовательностях. В рамках данного описания алгоритма будет использоваться 0-1 принцип [17], который гласит, что сортирующая сеть корректна, если она корректно сортирует все возможные последовательности из 0 и 1. Исходя из описанного выше принципа будут рассмотрены только битонические последвательности состоящие из 0 и 1, называющиеся нуль-единичные битонические последовательности.

Нуль-единичная битоническая последовательность – последовательность вида или , где и . Запись вида обозначает a подряд идущих нулей и аналогична для 1.

Принцип сортировки Бэтчера основывается на том, что битоническую последовательность можно достаточно просто получить из двух отсортированных последовательностей (будет показано далее). Получается, что если мы сможем эффективно сортировать такую последовательность, то по аналогии с сортировкой слиянием (см. п. 3.1) мы сможем разбить наш массив размера N на N / 2 частей длины 2, отсортировать их, а далее подниматься на уровни выше объединяя меньшие отсортированные последовательности в битонические и применять операцию сортировки. В конечном итоге, будет получен отсортированный массив. В рамках данной сортировки мы предполагаем, что размер массива с входными данными – всегда , так что при разбиении на меньшие части мы можем не бояться получения нечётных чисел. Способ, позволяющий сортировать массивы любых размеров будет описан в конце данной главы.

Построим сортирующую сеть, способную отсортировать произвольную нуль-единичную битоническую последовательность – битонический сортировщик. Его строение основывается на применении к битонической последовательности некоторого набора полуфильтров. Полуфильтр представляет из себя некоторую последовательность компараторов, которые выстраиваются следующим образом: пусть в рассматриваемой последовательности N элементов, тогда мы соединяем 0 и N / 2, 1 и N / 2 + 1 … N / 2 – 1 и N – 1. В данном случае очевидно, что все такие компараторы лежат в одном слое и могут быть выполнены параллельно. Такой полуфильтр разделит нашу исходную битоническую последовательность на две последовательности равной длины, одна из которых будет являться битонической, а другая однородной. Под однородностью подразумевается последовательность состоящая только из 0 или только из 1. Тогда мы можем к каждой полученной части снова применить полуфильтр вдвое меньшего размера, который разделит однородную последовательность на две однородные, а битоническую на битоническую и однородную и так далее. В итоге применяя полуфильтр к битонической последовательности длины 2 мы получим две однородные последовательности (рисунок 3). В итоге мы получаем отсортированную последовательность.

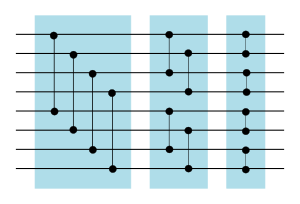


Рисунок 3. Битонический сортировщик для 8 элементов.

Теперь покажем, как при помощи битонического сортировщика объединить две отсортированные последовательности в одну отсортированную последовательность (аналогичное слиянию в сортировке слияниями). Построим так называемую объединяющую сеть, которая и будет справляться с данной задачей. На вход подаются две последовательности вида и . Заметим, что если к первой последовательности дописать перевёрнутую вторую, то получится последовательность вида , а это есть ни что иное, как битоническая последовательность, которую мы уже научились сортировать. Таким образом объединяющая сеть является ничем иным, как битонический сортировщик, с единственным отличием в первом полуфильтре, так как мы должны перевернуть вторую последовательность (рисунок 4).

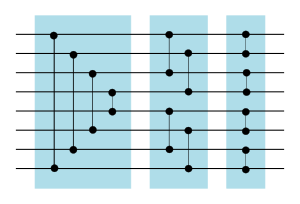


Рисунок 4. Объединяющая сеть для 8 элементов.

Чтобы построить сортирующую сеть Бэтчера необходимо вспомнить о принципах работа обычной сортировки слияниями. Пусть количество элементов на входе в алгоритм – N. На первом шаге отсортируем пары элементов, получив N / 2 отсортированных пар. Далее при помощи объединяющих сетей для 2 последовательностей из 2 элементов каждая получим N / 4 отсортированных последовательностей длины 4 и так далее до N (рисунок 5).

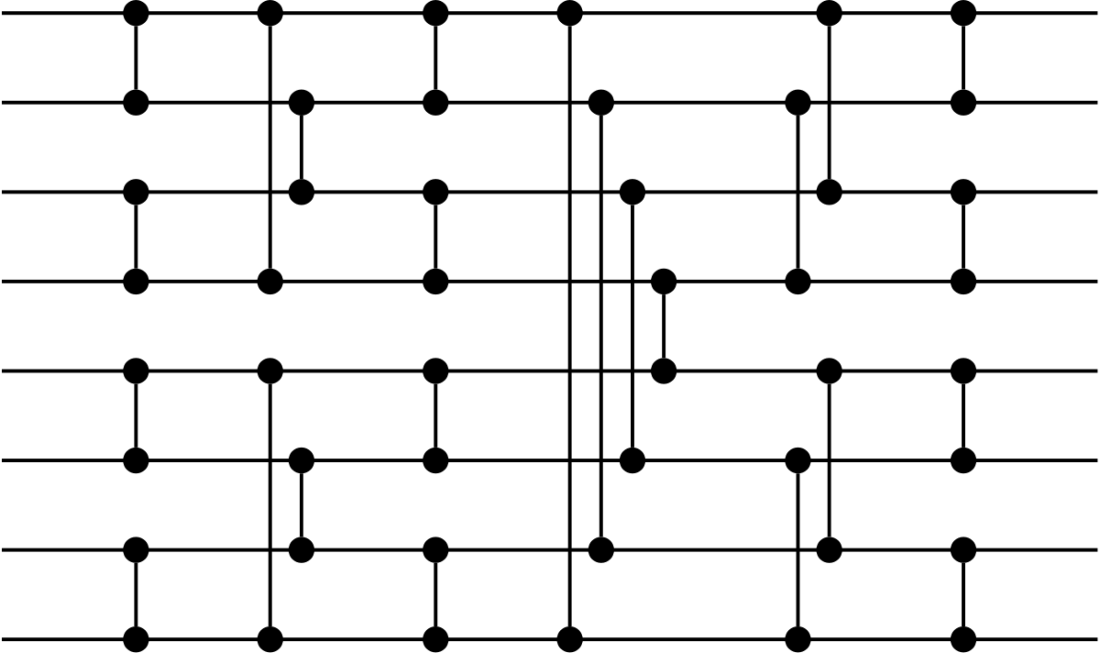


Рисунок 5. Сортирующая сеть Бэтчера для 8 элементов.

Таким образом получаем неадаптивную сортировку, работающую за [3]. Сортирующая сеть, соответствующая сортировке Бэтчера имеет размер и глубину . Алгоритм Бэтчера можно выполнять параллельно, получив итоговую асимптотику , это достигается путём распараллеливания выполнения компараторов, лежащих в одном слое.

* 1. **Разделители**

Одним из важнейших терминов для описания дальнейших алгоритмов сортирующих сетей является разделитель - это сортирующая сеть, берущая на вход 2 последовательности A и B длины N сравниваемых элементов и возвращающая на выходе изменённые A и B такие что, в А содержится некоторое количество наименьших элементов из , а в Б – некоторое количество наибольших элементов из .

Идеальный разделитель – разделитель, возвращающий в последовательности А N минимальных элементов из , а в B N максимальных элементов из .

-разделитель – разделитель, для любого помещающий не более из k максимальных элементов из в А и не более, чем из k минимальных элементов из в B. Это вероятностная сортирующая сеть. Пусть U и V – это первая и вторая часть элементов, к которым необходимо применить -разделитель, тогда его можно описать достаточно просто – пусть мы будем формировать двудольный экспандер на множестве вершин (U, V), тогда случайно сгенерировав r наибольших паросочетаний, и добавив их в граф, с высокой вероятностью рёбра данного графа, взятые в произвольном порядке, и будут формировать сортирующую сеть -разделителя. В данном случае r – это функция от , заданная следующим образом [20].

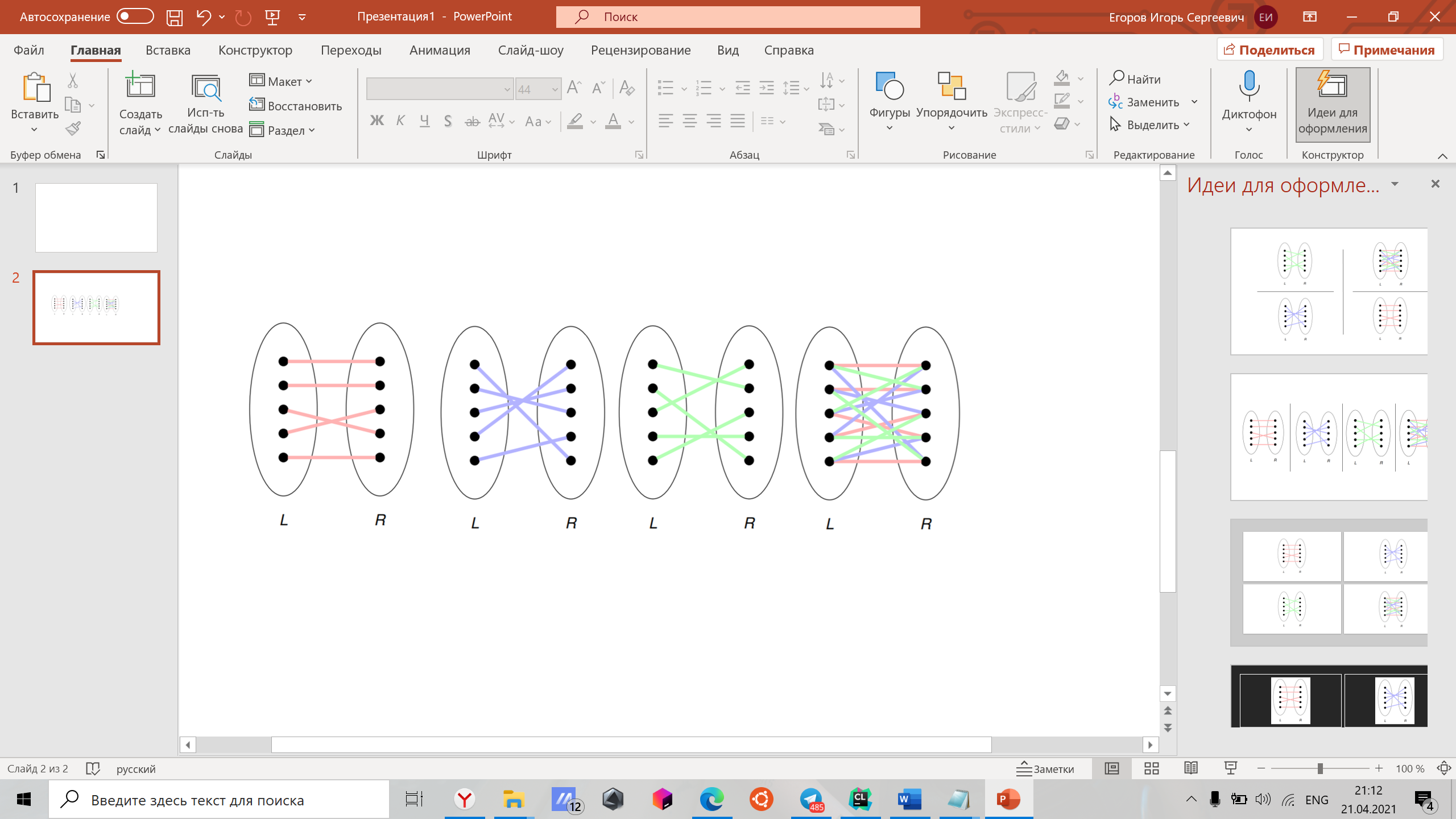


Рисунок 6. Пример формирования экспандера

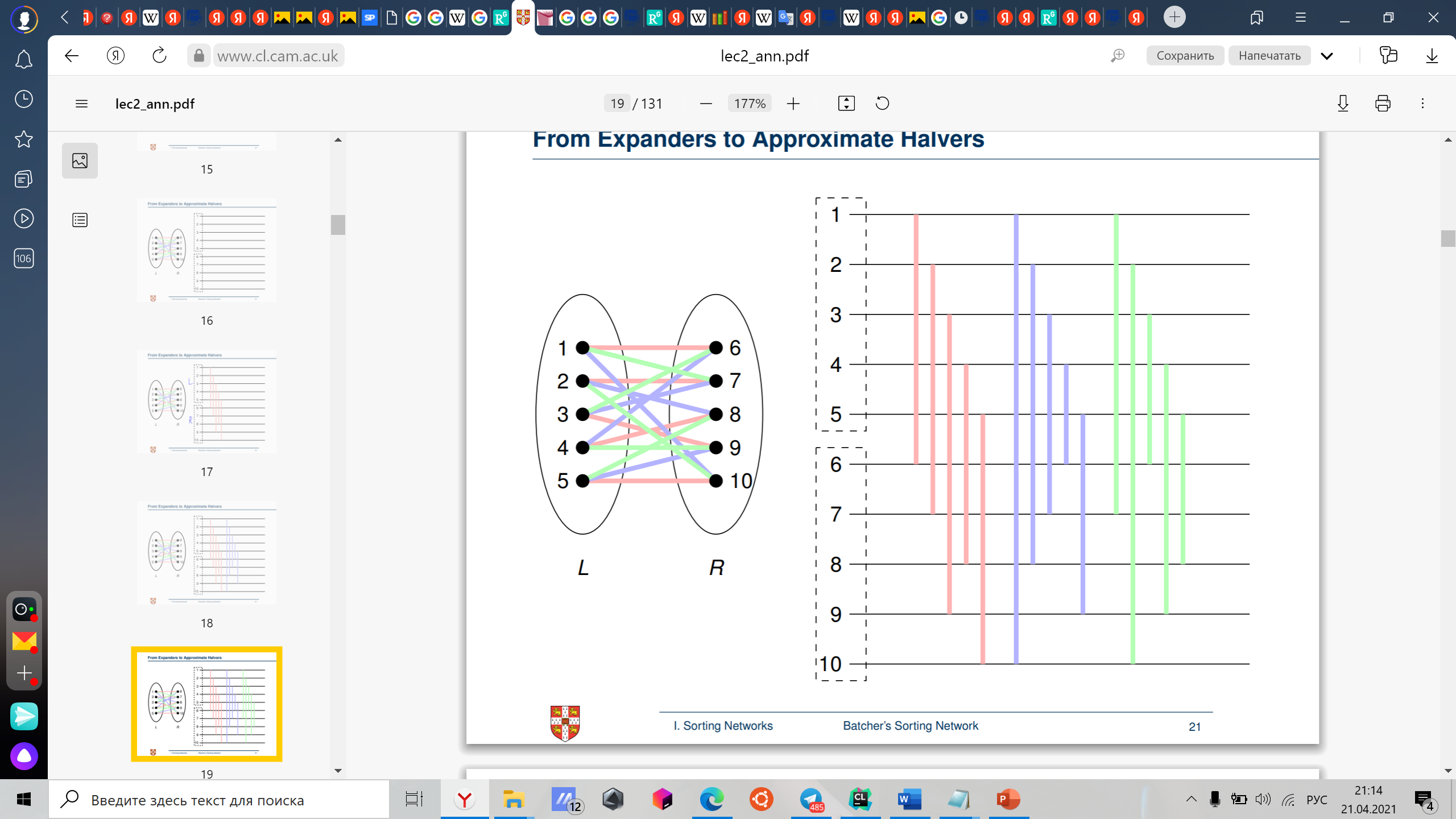


Рисунок 7. Пример формирования -разделителя на основе рёбер экспандера

* 1. **Сети АКС**

В течение долгого времени оставался открытым вопрос о существовании сортирующих сетей глубиной [3]. Однако, в 1983 году Miklós Ajtai, Janos Komlos и Endre Szemerédi смогли ответить на данный вопрос утвердительно. Получившаяся сортирующая сеть получила название в честь своих разработчиков (A- Ajtai, K- Komlos, C- Szemerédi). Они сумели разработать сортирующую сеть глубины и размером . Но, к сожалению, скрытые в О-обозначении константы получаются слишком большими, поэтому считается, что данная сеть не имеет никакой практической пользы.

Рассмотрим алгоритм сортировки сетями АКС.

Введём понятие -близкой сортировки – данный метод очень похож на быструю сортировку (см. п. 3.2). На вход подаётся некоторая последовательность элементов. Он формулируется достаточно просто: сначала применим -разделитель для всей последовательности. Далее поделим последовательность пополам и применим к каждой половинке -разделитель. После этого разобьём каждую половинку ещё пополам и применим -разделитель к каждой четвертинке и так далее, пока размеры разбиения не станут единичными.

По аналогии с сортировкой Бэтчера в рамках данной сортировки мы предполагаем, что размер массива с входными данными – всегда , так что при разбиении на меньшие части мы можем не бояться получения нечётных чисел.

Алгоритм сортировки можно описать достаточно компактно: пусть дана последовательность из N элементов на вход, произвольно пронумеруем их от 1 до N. Тогда для каждого i, такого что будет дано некоторое разбиение элементов на множества. Такие разбиения будут заданы явно и не будут зависеть от значений элементов. Для каждого i мы выполняем -близкой сортировки отдельно и одновременно для каждого множества из i-ого разбиения.

Опишем разбиения на множества: для описания формата разбиений будем использовать двоичное дерево (у каждой вершины 2 потомка, если это не лист). В некоторое время t разбиения наших элементы будут представимы в виде дерева, изображённого на рисунке 8. Каждой вершине соответствует натуральный интервал элементов: для корня – все элементы, для закрашенной вершины – (N / 4 + 1, 2 \* N / 4). Эти интервалы интерпретируются, как отрезки элементов, принадлежащие узлу. Всего в дереве во время t – t уровней. Например, на рисунке 8 t = 3.

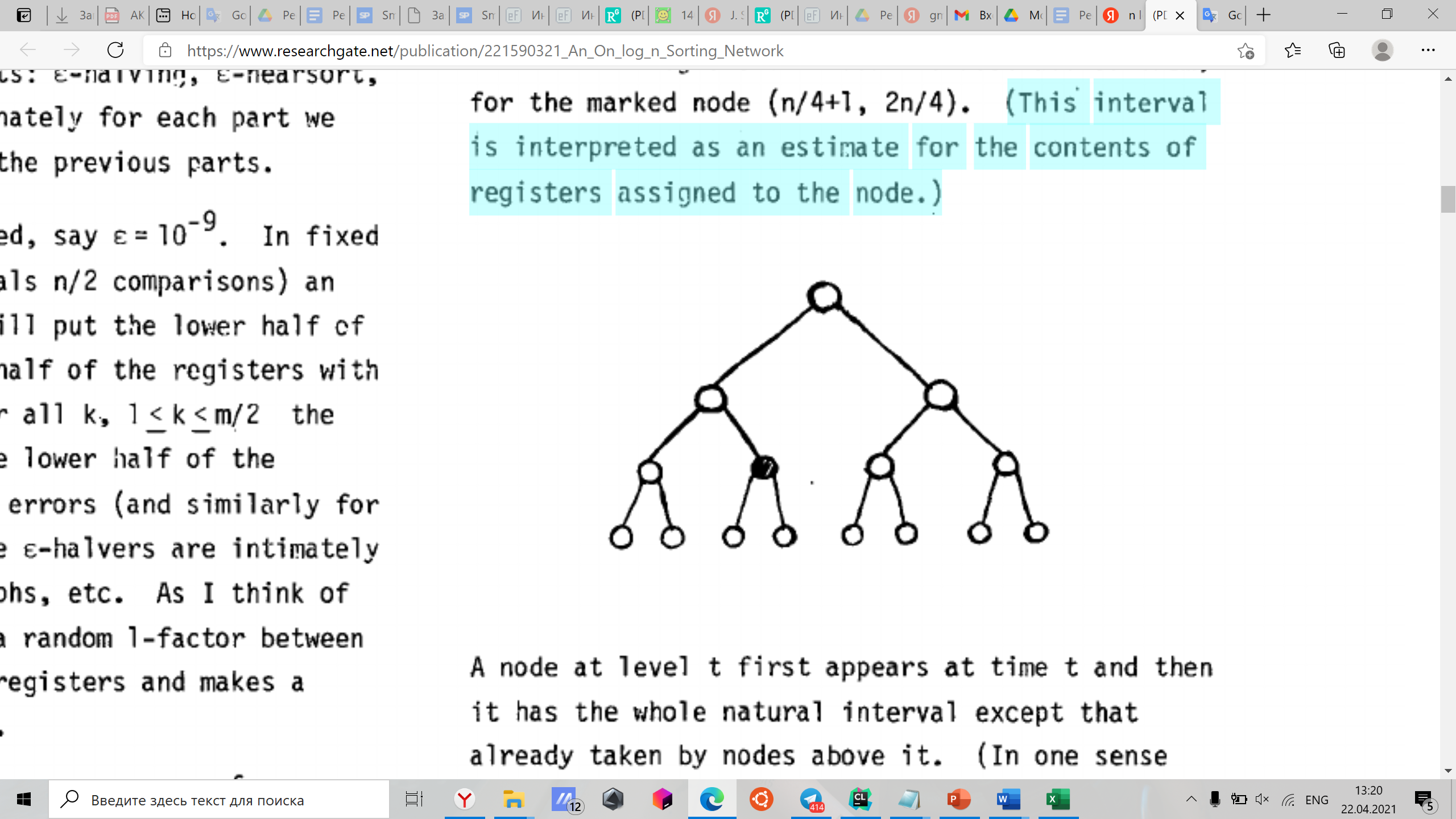


Рисунок 8. Представление бинарного дерева при t = 3

Предположим, что для времени t мы для каждой вершины определили отрезки элементов, за которые отвечает каждая вершина. Теперь разобьём всё наше дерево на тройки (треугольники), в которые войдут вершины, лежащие на чётных уровнях и их дети, и к каждой такой тройке применим -близкую сортировку. Назовём такой шаг «зиг». После этого разобьём наше дерево на тройки (треугольники), в которые войдут вершины, лежащие на нечётных уровнях и их дети, и к каждой такой тройке применим -близкую сортировку. Назовём такой шаг «заг» (рисунок 9). Выполним для времени t шаг «зиг-заг-зиг» (мы называем весь шаг «зиг-заг-зиг» одной единицей времени).

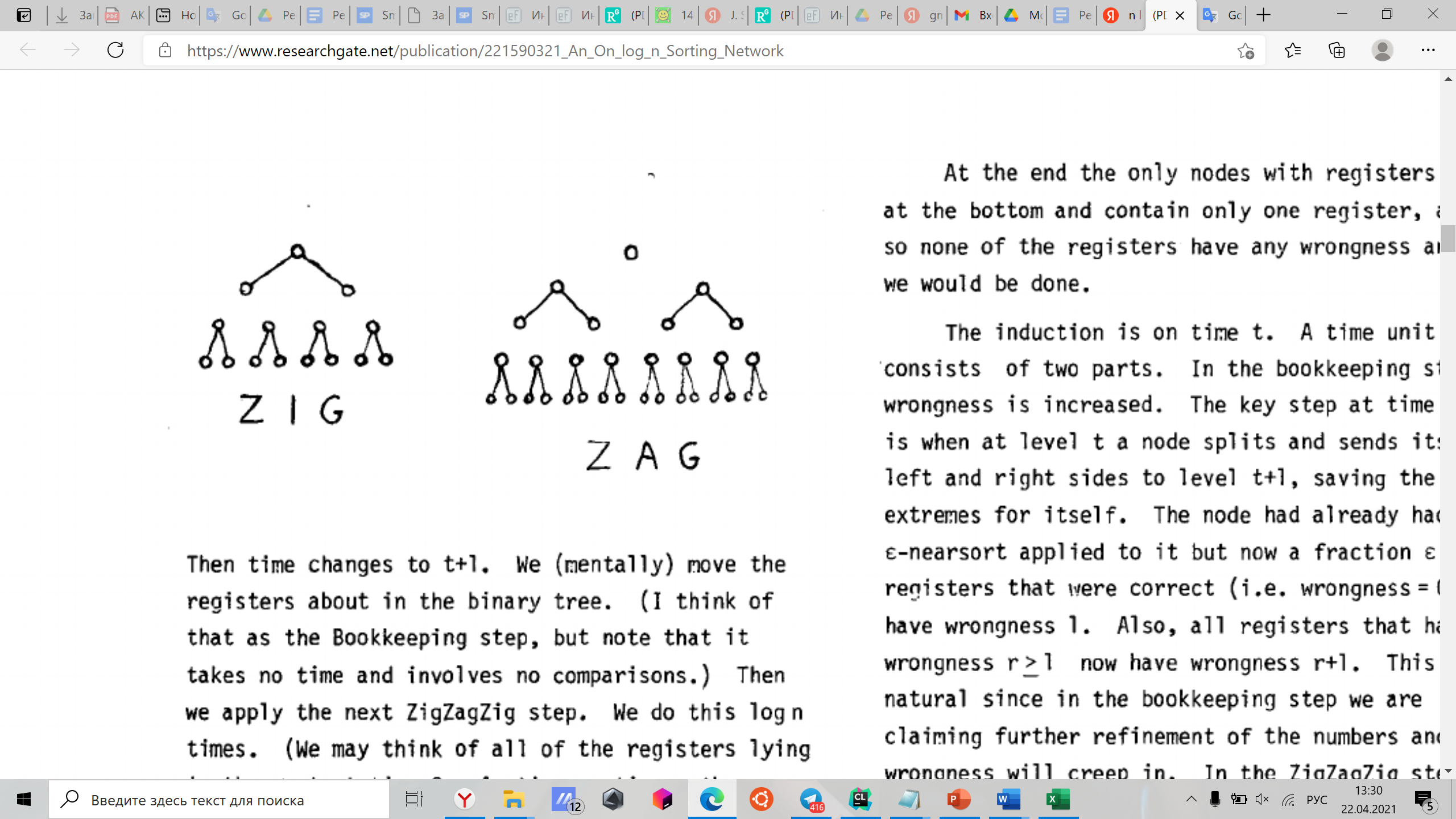


Рисунок 9. Шаги «зиг» и «заг»

Когда время изменится на t + 1, мы добавим к нашему дереву ещё один нижний слой и сместим наши отрезки в нижние уровни, не забыв разбить их на зоны ответственности на отрезки, и снова применим шаг «зиг-заг-зиг». Мы делаем так раз, после чего можно утверждать, что входная последовательность будет отсортирована [11].

Такая сортирующая сеть будет иметь глубину O (log N) [11], однако, в ходе выполнения данной сортировки будет накоплена достаточно большая константа, которая возникает из -разделителя.

Это константа будет настолько огромна, что Дональд Кнут сказал: “Метод Батчера будет работать лучше, пока N не превышает общий объём памяти компьютеров на всей земле!”. А Ричард Липтон назвал эту сортировку «Галактикой»: “Сортирующая сеть АКС — это галактика: необходимо, чтобы размер входных данных был около , чтобы она наконец работала быстрее, чем сортировка Батчера”.

* 1. **Сортировка «зиг-загом»**

В 2014 году Михаэлем Годричем была открыта новая сортировка, имеющая размер сортирующей сети и глубину [18].

Данная сортировка является вариацией неадаптивной сортировки Шелла [23], при этом, это первый вариант такой сортировки являющийся детерминированным и работающий за . Отличие от предыдущих вариантов состоит в том, что приращения на каждой фазе алгоритма не фиксированы – они варьируются в зависимости от диапазонов, разделяемых на каждой из log(n) фаз.

В основе данного алгоритма, как и в основе сетей АКС, лежит повторение -разделителей для разных диапазонов сортируемых объектов.

В рамках данной сортировки мы предполагаем, что размер массива с входными данными – всегда , так что при разбиении на меньшие части мы можем не бояться получения нечётных чисел. Способ, позволяющий сортировать массивы любых размеров будет описан в конце данной главы.

Введём несколько новых понятий, необходимых для описания сортировки:

1. Смягчение определения -разделителей: ()-разделитель – это тот же самый -разделитель, за исключением того, что , для .
2. -attenuator – это не зависимая от значений входных данных процедура, которая принимает на вход пару последовательностей A, B размером N, таких что из наибольших элементов из лежат в А, и из наименьших элементов из лежат в B и возвращает последовательность операций сравнения-обмена, такие что не более, чем из наибольших элементов из останутся в А, и не более, чем из наименьших элементов из останутся в B для , , и .

-attenuator построен на -разделителях и имеет следующий псевдокод (рисунок. 10).

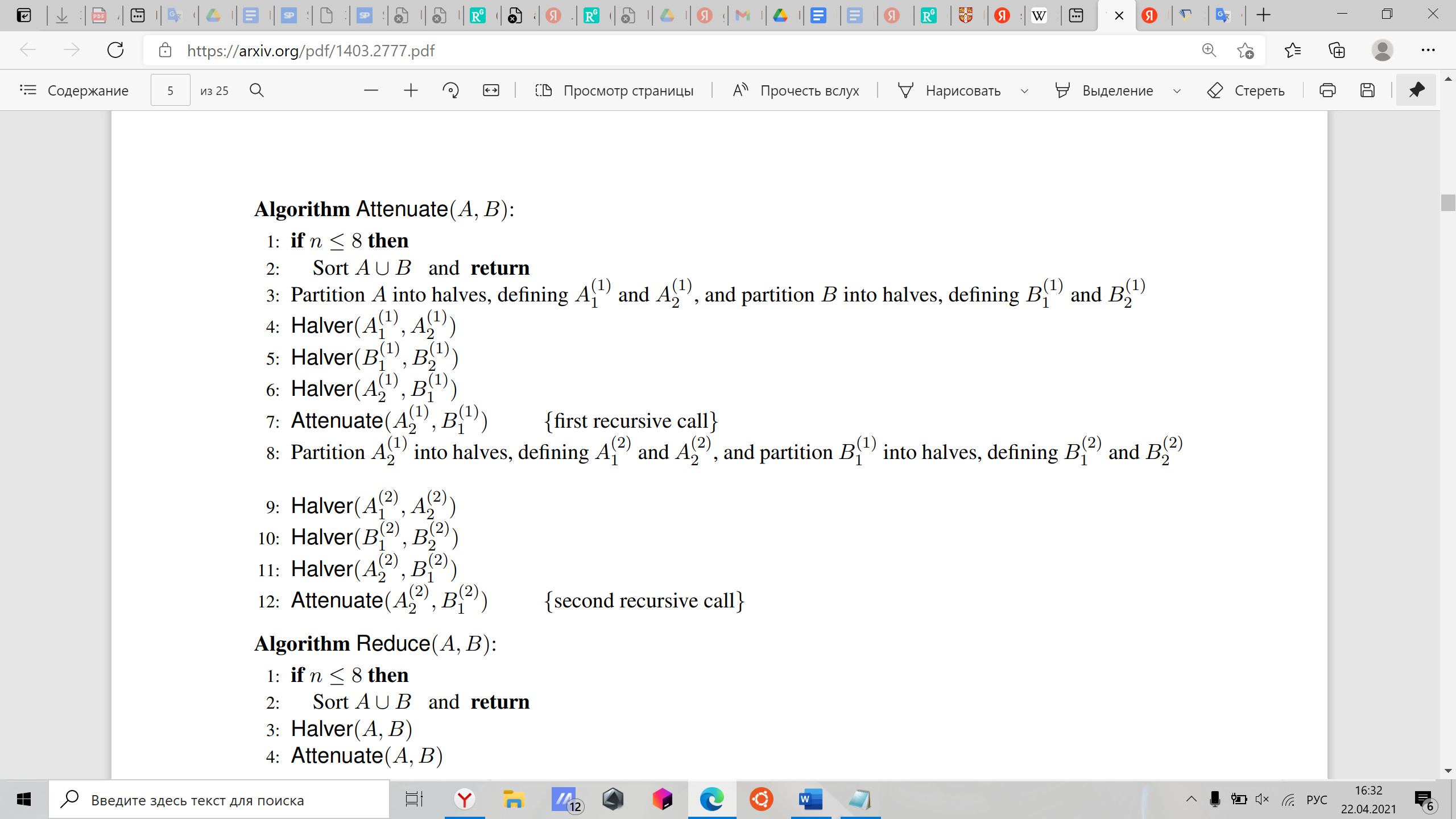


Рисунок 10. Псевдокод attenuator [18]

Также на рисунке 11 представлены потоки данных в attenuator.



Рисунок 11. Потоки данных в attenuator

Имя «зиг-заг» сортировка получила из-за применения в себе процедур, которые можно назвать «зиг-загами». Первым местом такой процедуры является внешний цикл, где мы делаем разбиение исходной последовательности на подпоследовательности в стиле подъёмов в сортировке Шелла (далее будем называть это фазой «внешнего зига»). За ним следует проход в стиле спуска в сортировке Шелла по полученным под подпоследовательностям (далее будем называть это фазой «внешнего зага»). Второе место применения такого типа процедур находится внутри таких циклов, где для последовательных подпоследовательностей мы генерируем набор операций сравнений-обмена после того, как поменяем местами элементы в этих двух подмассивах. Такой шаг мы называем «внутренний зиг-заг». Далее приведём псевдокод данной сортировки.

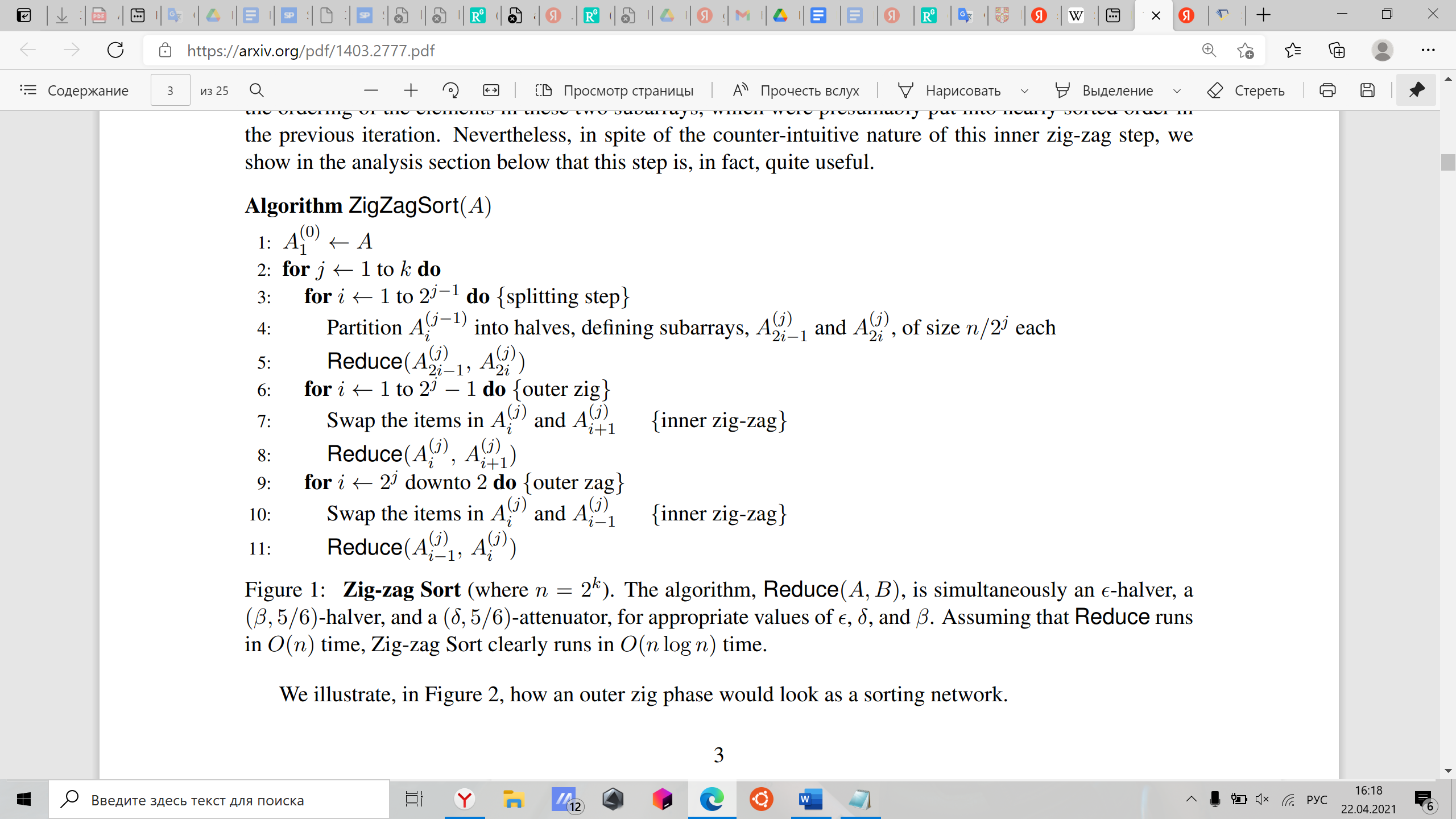


Рисунок 12. Псевдокод алгоритма сортировки зиг-загом [18].

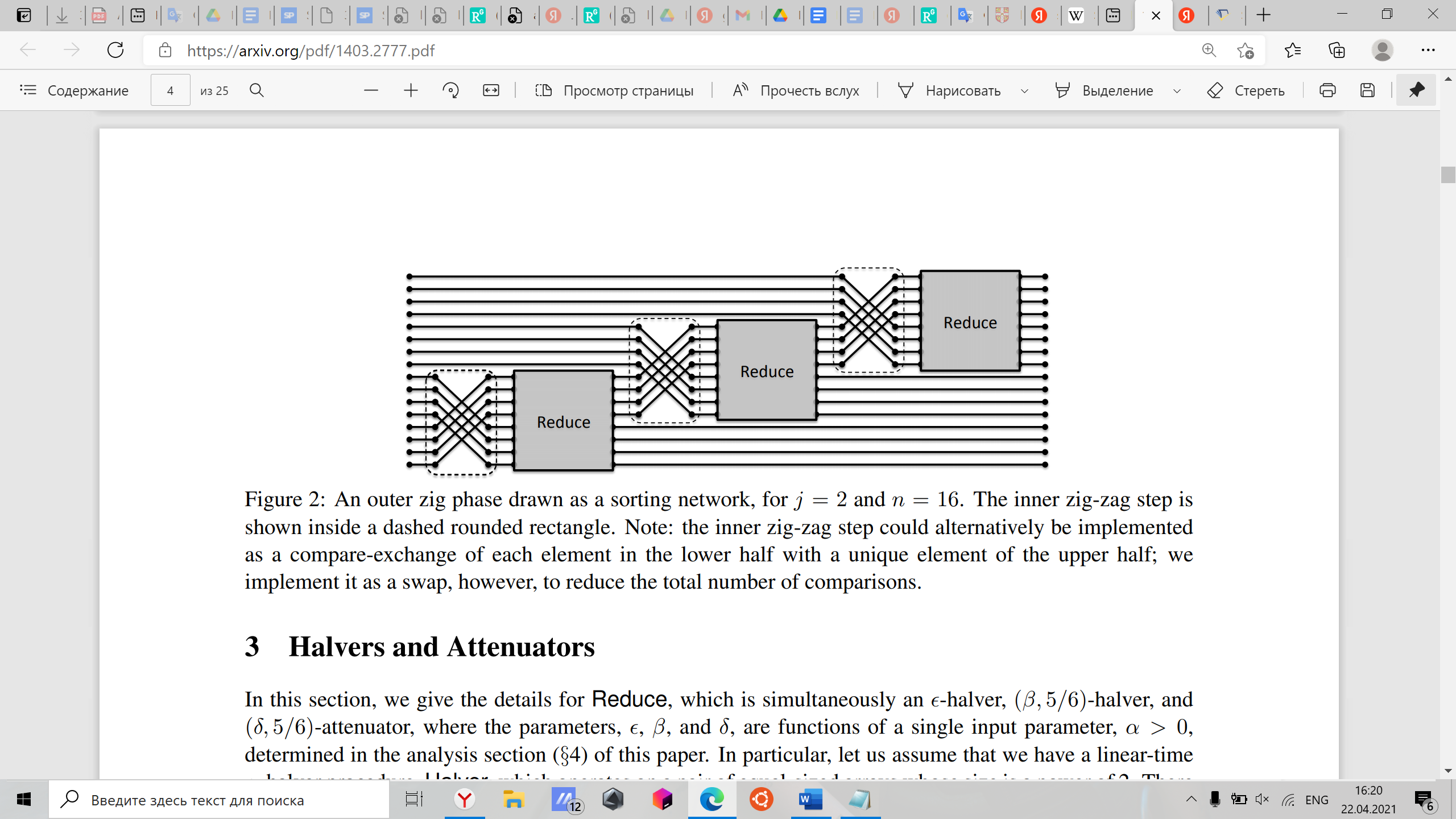


Рисунок 13. Сортирующая сеть, представляющая собой фазу «внешнего зига». На данном рисунке фаза «внутреннего зиг-зага» представлена в виде прямоугольников с границей контуром.

Функция Reduce является одновременно -разделителем, ()-разделителем и ()- attenuator для определённых . Далее будет представлен псевдокод метода Reduce.

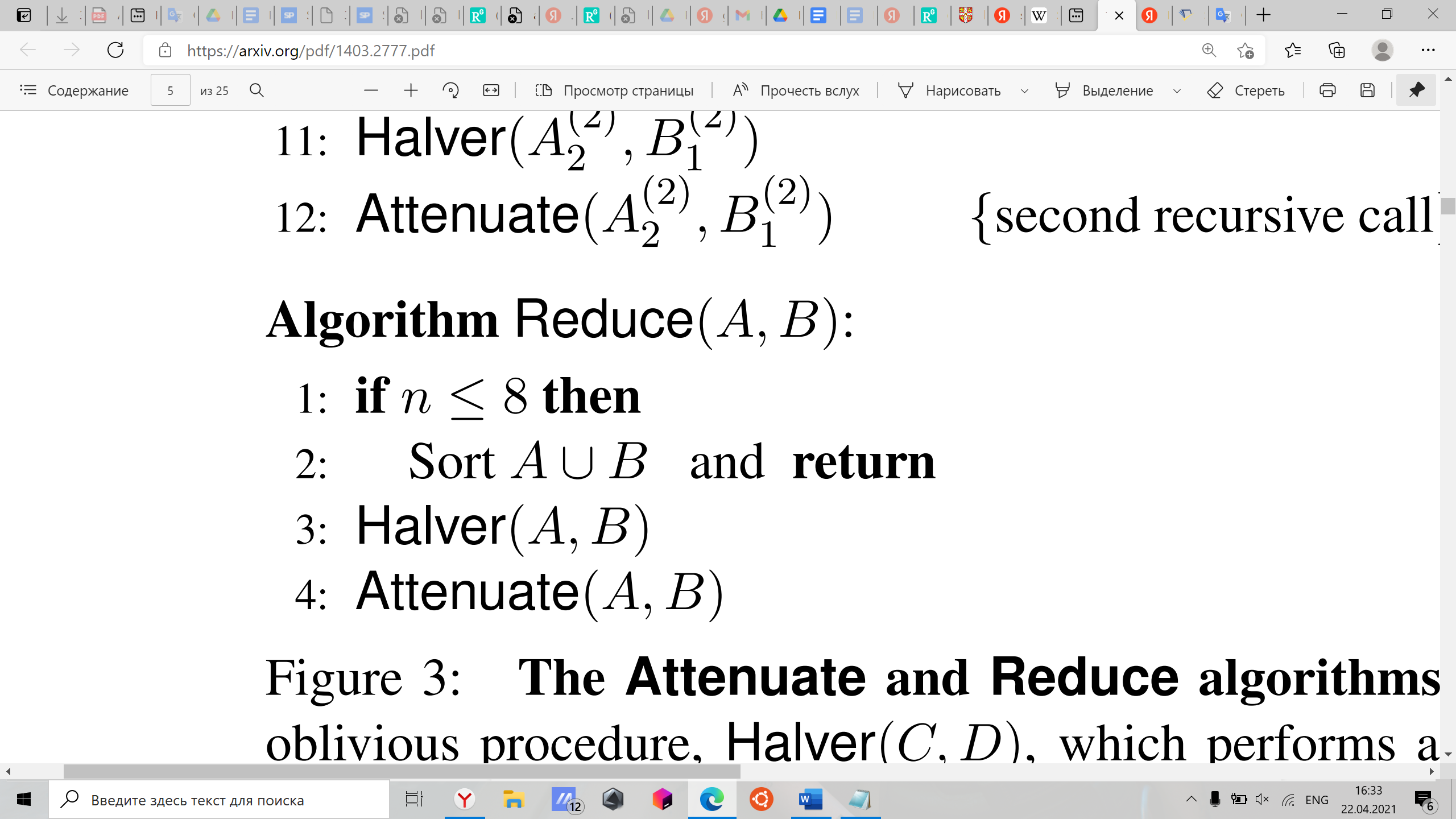


Рисунок 14. Псевдокод метода Reduce.

Утверждается, что данный алгоритм работает при для -разделителем, и константа времени его работы для данного равна 2700 [18].

* 1. **Сортировка произвольного объёма данных**

Как было замечено выше, сортировка «зиг-загом» и сортировка Бэтчера работают корректно для объёма входных данных равному какой-либо степени двойки.

Основным способом сортировать данные произвольного объёма является дополнение этих данных до объёма ближайшей степени двойки с конца элементами большими любых элементов исходных данных. Это гарантирует, что ни одна операция сравнения-обмена с такими фиктивными данными выполнена не будет и эти элементы так и останутся в конце входных данных. При обработке финальных результатов их можно просто не учитывать. Этот же метод можно реализовать без использования дополнительной памяти просто не выполняя любую операцию сравнения обмена, если номер какого-либо элемента для данной операции выходит за границы исходных данных. Это аналогично сравнению с фиктивными элементами, так как сама операция обмена никогда не будет выполнена.

1. **Адаптивные сортировки**

На данный момент адаптивные сортировки являются самым популярным и широко известным методом сортировки объектов. Именно поэтому в рамках данной работы будет выполнено сравнение неадаптивных сортировок, описанных выше, с самыми популярными и эффективными адаптивными сортировками.

* 1. **Сортировка слиянием**

Данная сортировка была разработана в 1945 году Джоном фон Нейманом, пионером теории алгоритмов и программирования. Идея сортировки слиянием заключается в использовании рекурсивного метода «разделяй и властвуй». Разделив массив данных на меньшие части и отсортировав их, можно эффективно слить два отсортированных подмассива в один [1]. Базой рекурсии могут выступать подмассивы размера 1 - они уже отсортированы по определению. В сортировке слиянием массив делится на две равных части по индексам *[1, n/2]* и *[n/2+1, n]*, где *n* – количество входных данных, а дальше запускается от них рекурсивно.

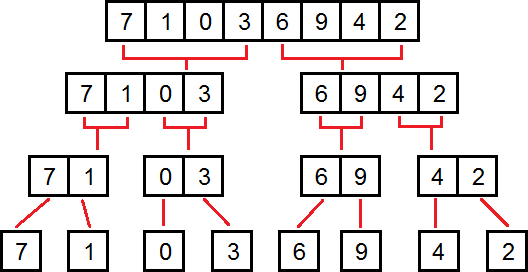


Рисунок 15. Демонстрация разделения массива на подмассивы при сортировке слиянием

Асимптотика данной сортировки - *O(n\*log(n))* [2], что ставит её в ряд с быстрейшими сортировками из основанных на сравнении.

* 1. **Быстрая сортировка Хоара**

Быстрая сортировка была разработана Чарльзом Энтони Ричардом Хоаром в 1962 году.

Алгоритм быстрой сортировки состоит из двух этапов - разделение (partition) и сортировка (sort).

Разделение принимает массив и по некоторому правилу выбирает опорный элемент (pivot), относительного которого размещает элементы массива слева или справа от опорного. Таким образом, чтобы отсортировать весь массив, необходимо отсортировать левую и правую часть, при этом дальнейшее слияние не требуется.

Шаг сортировки же просто вызывает partition рекурсивно от левой и правой частей, полученных не предыдущем шаге, либо выходит из рекурсии, если достигнут подмассив размера 1 (заведомо отсортированный).

Разбиение Хоара заключается в том, что в массиве с уже имеющимся значением опорного элемента заводятся два итератора -- один в начале массива, второй -- в конце. Первый итератор инкрементируется, пока не встретит элемент, больший опорного. Второй декрементируется, пока не встретит элемент меньше опорного. Далее значения по итераторам меняются местами. Ход итераторов продолжается, пока они не окажутся в одной и той же ячейке массива.

Таким образом после разбиения Хоара мы получим массив, содержащий сначала часть элементов, меньших опороного, далее - опорный элемент, занимающий своё место, как если бы массив был отсортирован, и за ним - часть элементов, больших опорного.

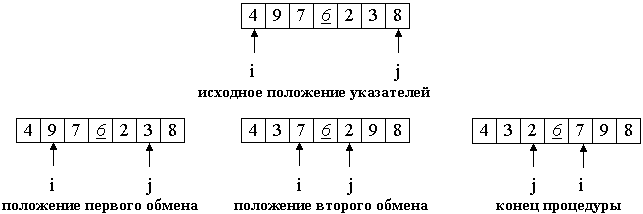


Рисунок 16. Демонстрация разделения Хоара

При этом количество уровней разделения массива в среднем будет равно *O(log(n))*, что приводит нас к общей асимптотике в *O(n \* log(n))*[1].

За счёт малой асимптотики, локальности, а также возможности введения дополнительных оптимизаций быстрая сортировка является самой популярной для использования в обобщённых библиотеках для языков программирования. В частности, она реализована в методе sort() из библиотеки <algorithm> языка С++. Можно утверждать, что быстрая сортировка - основной способ сортировки данных в программных продуктах.

* + 1. **Пирамидальная сортировка**

Пирамидальная сортировка была разработана Дж. Уильямсом в 1964 году.

Создание пирамидальной сортировки было тесно связано с развитием контейнеров для хранения однородных данных в оперативной памяти. По большому счёту она оказалась скорее приложением структуры, поддерживающей извлечение минимума, к задаче сортировки.

Пирамида (или куча) - структура данных, имеющая вид двоичного дерева, для вершин которого выполняется следующее свойство: значение ключа в любой вершине не меньше ключей в её потомках. При этом все листья такого дерева должны находиться на уровнях, отличающихся не более чем на 1. Таким образом достигается гарантированная глубина дерева в *O(log(n))* [2].

Тогда, если мы сможем построить пирамиду на данных в массиве, который требуется отсортировать, а далее будем последовательно извлекать наименьший элемент и отправлять его в начало массива, при этом восстанавливая свойства пирамиды, то мы сможем извлечь все элементы и расположить их в порядке возрастания, что и будет являться решением задачи сортировки.

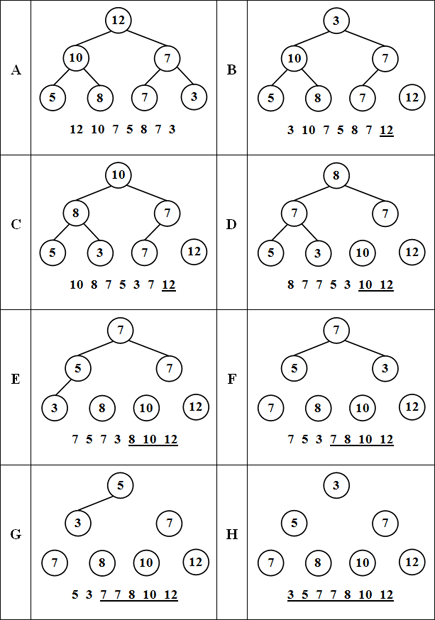


Рисунок 17. Демонстрация работы пирамидальной сортировки

Асимптотика работы пирамидальной сортировки равна *O(n \* log(n))*[2].

1. **Экспериментальный комплекс**

Цель эксперимента – сравнить по времени работы алгоритмы адаптивных и неадаптивных сортировок.

* 1. **Подготовка к проведению экспериментов**

В рамках эксперимента были реализованы следующие сортировки на языке С++:

* адаптивные сортировки: быстрая сортировка Хоара, сортировка слияниями, пирамидальная сортировка;
* неадаптивные сортировки: сортировка пузырьком, сортировка Бэтчера, сортировка «зиг-загом».

Далее все адаптивные и неадаптивные сортировки были помещены в соответствующие библиотеки (adaptive\_sorts и nonadaptive\_sorts).

Проверка на корректность реализаций сортировок осуществлялась путём тестирования их на наборах тестов из:

1. Всех возможных перестановок псевдослучайно сгенерированных массивов размером от 2 до 12 элементов типа *int* при помощи генератора *mt19937* [20] из библиотеки random;
2. На сгенерированном наборе из 1000 тестов, содержащих в себе псевдослучайно сгенерированные массивы при помощи генератора *mt19937* размером от 1000 до 10000000.

Затем была подготовлена программа, позволяющая при помощи библиотеки *chrono* [21] замерять время выполнения определённого участка кода.

Все материалы, подготовленные в рамках эксперимента доступны в github репозитории [22].

Эксперимент проводился на стационарной машине со следующими характеристиками: AMD Ryzen 9 4900HS (8 ядер, 16 потоков). 8 GB RAM.

Были последовательно сгенерированы псевдослучайные массивы чисел типа *int* размеров 10, 100, 1000, 10000, 100000, 1000000, 10000000, 100000000. Назовём данные массивы тестовыми в рамках экспериментов.

После этого были проведены два независимых эксперимента.

* 1. **Предварительная оценка времён работ алгоритмов**

Так как все реализованные сортировки были протестированы на наборах тестов только на корректность, хотелось бы также проверить, что они удовлетворяют описанной асимптотике. Для этого посчитаем ожидаемое количество операций для каждого алгоритма с учётом констант и после этого поделим общее количество операций на среднюю скорость моей стационарной машины для языка С++, которая составляет в среднем около 10^8 операций в секунду.

* для быстрой сортировки сортировки слиянием, пирамидальной сортировкой ожидаемое количество операций совпадает с О-нотацией для данных сортировок и равно *n \* log(n);*
* для сортировки пузырьком ожидаемое количество операций совпадает с О-нотацией и равно *n \* n*;
* для сортировки Бэтчера ожидаемое количество операций равняется количеству компараторов в сети .
* для сортировки «зиг-загом» доказана константа равная 2700 для самого быстрого из известных -разделителей, однако, мной была реализована чуть менее оптимальная версия -разделителя, которая в среднем работает хуже, чем самая оптимальная версия в 2 раза [19], поэтому итоговое ожидаемое количество операций .

Итого, посчитав для каждого теста ожидаемое время работы и поделив его на 10^8 получаем следующую таблицу:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **10** | **100** | **1000** | **10000** | **100000** | **1000000** | **10000000** | **100000000** |
| **Быстрая сортировка** | 0 | 0 | 0 | 0,001 | 0,017 | 0,199 | 2,325 | 26,575 |
| **Сортировка слияниями** | 0 | 0 | 0 | 0,001 | 0,017 | 0,199 | 2,325 | 26,575 |
| **Пирамидальная сортировка** | 0 | 0 | 0 | 0,001 | 0,017 | 0,199 | 2,325 | 26,575 |
| **Сортировка "Пузырьком"** | 0 | 0 | 0,01 | 1 | 100 | 10000 | 1000000 | 100000000 |
| **Сортировка Бэтчера** | 0 | 0 | 0 | 0,004 | 0,069 | 0,993 | 13,518 | 176,563 |
| **Сортировка зиг-загом** | 0,003 | 0,048 | 0,553 | 12,386 | 120,324 | 1132,462 | 21743,27 | 195689,45 |

Таблица 1. Ожидаемое время работы сортировок в секундах

* 1. **Описание первого эксперимента**

В данном эксперименте в неадаптивных сортировках операция сравнения-обмена применялась сразу к массиву данных, а не генерировала пару индексов для сортировки массива вне рассматриваемой сортировки. В данном эксперименте было замеряно время исполнения неадаптивных и адаптивных сортировок на тестовых массивах.

* 1. **Результаты первого эксперимента**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **10** | **100** | **1000** | **10000** | **100000** | **1000000** | **10000000** | **100000000** |
| **Быстрая сортировка** | 0 | 0 | 0 | 0,002 | 0,026 | 0,32 | 3,884 | 43,873 |
| **Сортировка слияниями** | 0 | 0 | 0 | 0,003 | 0,043 | 0,487 | 4,197 | 45,766 |
| **Пирамидальная сортировка** | 0 | 0 | 0 | 0,002 | 0,036 | 0,464 | 4,29 | 46,812 |
| **Сортировка "Пузырьком"** | 0 | 0 | 0,002 | 0,188 | 18,882 | - | - | - |
| **Сортировка Бэтчера** | 0 | 0 | 0 | 0,005 | 0,075 | 0,985 | 12,223 | 168,231 |
| **Сортировка зиг-загом** | 0 | 0,038 | 0,489 | 11,884 | 124,77 | 1206,507 | - | - |

Таблица 2. Измеренное время работы сортировок во время первого эксперимента в секундах (под – указано время, которое не удалось замерить в связи слишком долгой работой сортировки)

Если сравнить ожидаемое время работы и полученное, то можно сделать следующие выводы: в среднем имплементации всех адаптивных сортировок и сортировки пузырьком работают в два раза хуже ожидаемого времени. Сортировки Бэтчера и сортировка «зиг-загом» работают примерно, как и ожидалось.

* 1. **Описание второго эксперимента**

В данном эксперименте по исполнении очередной неадаптивной сортировки для массивов длины N получался файл с парами индексов, необходимыми для операций сравнения-обмена для сортировки любых массивов размера N. В данном эксперименте было замеряно время работы неадаптивных сортировок для генерации необходимых пар, а после для сортировки массива по сгенерированным парам. Чтобы минимизировать влияние времени записи в файл и выгрузки из него, все инструкции сначала записывались в заранее подготовленный массив длины 10^8, после заполнения данного массива командами / исполнения всех команд из файла запись времени приостанавливалась, данные из массива выгружались в файл или загружались из него. После чего продолжалась работа сортировки.

* 1. **Результаты второго эксперимента**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **10** | **100** | **1000** | **10000** | **100000** | **1000000** | **10000000** | **100000000** |
| **Сортировка "Пузырьком"** | 0 | 0 | 0,004 | 0,302 | 29,511 | - | - | - |
| **Сортировка Бэтчера** | 0 | 0 | 0 | 0,006 | 0,079 | 1,026 | 13,276 | 173,087 |
| **Сортировка зиг-загом** | 0 | 0,02 | 0,401 | 12,076 | 125,392 | 1210,714 | - | - |

Таблица 3. Измеренное время работы сортировок для генерации последовательностей операций сравнений-обмена (пар индексов, к которым необходимо применить данную операцию), задающих сортирующую сеть (под – указано время, которое не удалось замерить в связи слишком долгой работой сортировки)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **10** | **100** | **1000** | **10000** | **100000** | **1000000** | **10000000** | **100000000** |
| **Сортировка "Пузырьком"** | 0 | 0 | 0,005 | 0,298 | 29,834 | - | - | - |
| **Сортировка Бэтчера** | 0 | 0 | 0 | 0,005 | 0,08 | 1,072 | 14,794 | 176,064 |
| **Сортировка зиг-загом** | 0 | 0,002 | 0,052 | 1,236 | 17,772 | 218,382 | - | - |

Таблица 4. Время работы сортировок для применения к тестовым массивам, операций сравнения-обмена, сгенерированных соответствующими сортировками (под – указано время, которое не удалось замерить в связи слишком долгой работой сортировки)

Как видно из таблицы 3, время генерации последовательностей операций сравнения-обмена немного больше, чем применения операций сравнения-обмена сразу к массивам (таблица 1). Аналогичная ситуация и для применения сгенерированных операций к массивам (таблица 4). Это связано с тем, что операция записи очередной пары в какой-либо массив для её хранения немного дороже операции непосредственного применения операции сравнения-обмена к паре элементов массива. Аналогична ситуация для вытаскивания из массивов пары индексов для того, чтобы провести с ними операцию сравнения обмена.

Особенно интересным в данном эксперименте является то, что для сортировки «зиг-загом» затраченное время на то, чтобы применить сгенерированные последовательности операций сравнения-обмена к массивам гораздо меньше затраченного времени на генерацию данных последовательностей. Это объясняется тем, что в сортировке «зиг-загом» помимо операций сравнения-обмена присутствует огромное количество иных действий – разбиение массива на подмассивы, генерация рандомных последовательностей, огромное количество рекурсий. В связи с этим генерация последовательностей операций срванения-обмена занимает лишь меньшую часть от всего времени работы сортировки.

1. **Выводы**

Были изучены самые известны алгоритмы неадаптивных сортировок;

Была разработана программа для сравнения времён работ адаптивных и неадаптивных сортировок;

Был проведён вычислительный эксперимент, позволяющий получить сравнение времён работ адаптивных и неадаптивных сортировок на различных объёмах входных данных;

Была реализована открытая библиотека, содержащая в себе реализации самых популярных неадаптивных сортировок [22].

По итогу изученного материала можно резюмировать следующее: в настоящее время исследованием неадаптивных сортировок занимается очень небольшая группа учёных, также данные алгоритмы сортировок являются достаточно неизвестными в силу того, что их изучение не входит в основные программы дисциплин университетов. Поэтому их развитие происходит крайне медленно, о чём можно судить даже по динамике публикуемых статей по данной тематике. В связи с этим, данный раздел сортировок открывает достаточно широкий простор возможных исследований, связанных с неадаптивными сортировками.

Как видно из продемонстрированных экспериментов, неадаптивные сортировки уступают в скорости адаптивным сортировкам, поэтому предпочтения отдаются именно последним.

1. **Направления дальнейшей работы**

Реализация распараллеливания сортировки Батчера и проведение на ней эксперимента.

Более детальное изучение существующих оптимизаций сортирующих сетей АКС с целью разработки оптимизации, позволяющей сильно сократить константу.

Также, в конце работы над курсовым проектом, была предпринята попытка разработать неадаптивную сортировку на основе структуры данных «куча». Далее планируется доведение данной разработки до конца.

Публикация ряда статей по изученным материалам на интернет-ресурсе habr.com.

1. **Термины и сокращения**

Задача сортировки – состоит в упорядочивании набора однотипных объектов по некоторому правилу (например: по возрастанию или убыванию).

Сортировка – алгоритм, предлагающий решение задачи сортировки.

Неадаптивная сортировка – сортировка, которая для проведения следующих шагов сравнения не требует информации о результатах предыдущего шага, то есть неадаптивные сортировки линейны во время исполнения – они не имеют доступа к исходным данным, только к их количеству, поэтому не могут делать выводы на основании ключей входных данных, а значит конструкции ветвления в данных сортировках исключены. Именно поэтому неадаптивные сортировки способны генерировать операции сравнения-обмена в некотором порядке, набор которых будет сортировать любые последовательности объектов фиксированной длины (той же самой, на которой и была запущена сортировка).

Адаптивная сортировка – сортировка, использующая для оптимизации дальнейших шагов информацию о данных, полученную на предыдущих шагах.

Операция сравнения-обмена – это функция, принимающая на вход пару индексов, и, в случае нарушения порядка в этих индексах (больший элемент стоит в меньшем индексе), меняющая элементы местами.

Компаратор – простейшее сравнивающее устройство, принимающее на вход два элемента и применяющее к ним операцию сравнения-обмена.

Двудольный граф – граф, множество вершин которого можно разбить на два множества таким образом, что любое ребро графа соединяет вершину из одного множества с какой-то вершиной из другого множества. Иными словами, не существует рёбер между вершинами, расположенными в одном множестве.

Паросочетание – набор попарно несмежных рёбер графа, то есть рёбер, не имеющих общих вершин.

Наибольшее паросочетание – максимальный по мощности набор их всех возможных паросочетаний графа. То есть к нему невозможно добавить ребро, которое бы являлось несмежным ко всем рёбрам паросочетания.

Экспандер – конечный ненаправленный мультиграф (разрешается присутствие кратных рёбер), в котором любое подмножество вершин, не являющихся «сильно большим», имеет «сильную связность». У этого понятия могут быть различные формализации.

1. **Список использованных источников**
2. Алгоритмы. Построение и анализ, 2-е изд. / Томас Кормен, Чарльз Лейзерсон, Рональд Ривест, Клиффорд Штайн.; Пер. с англ. – М.: ООО «И.Д. Вильямс», 2005. – 1296 c.
3. Структуры данных и алгоритмы / Альфред В. Ахо, Джон Э. Хопкрофт, Джеффри Д. Ульман.; пер. с англ. СПб : ООО «Диалектика», 2019 – 400 c.
4. Роберт Седжвик. Фундаментальные алгоритмы на С++. [Электронный ресурс] // Internet archive [Официальный сайт]. URL: <https://archive.org/details/B-001-001-736/page/n3/mode/2up> (дата обращения 20.10.2020)
5. Дябилкин Д.А. Сортирующие сети. [Электронный ресурс] // Казанский федеральный университет [Официальный сайт]. URL: <https://dspace.kpfu.ru/xmlui/bitstream/handle/net/20359/09_104_001110.pdf> (дата обращения: 29.12.2020)
6. [Cormen, Thomas H.](https://en.wikipedia.org/wiki/Thomas_H._Cormen) [Leiserson, Charles E.](https://en.wikipedia.org/wiki/Charles_E._Leiserson) [Rivest, Ronald L.](https://en.wikipedia.org/wiki/Ron_Rivest) (1990). [*Introduction to Algorithms*](https://en.wikipedia.org/wiki/Introduction_to_Algorithms) (1st ed.). MIT Press and McGraw-Hill. [ISBN](https://en.wikipedia.org/wiki/ISBN_(identifier)) [0-262-03141-8](https://en.wikipedia.org/wiki/Special:BookSources/0-262-03141-8). -C. 636-637
7. Michel Goemans. Batcher’s Algorithm. [Electronic resource] // MIT Mathematics [Official website]. URL: <http://math.mit.edu/~shor/18.310/batcher.pdf> (accessed: 24.03.2021)
8. Vaˇsek Chv´atal. Lecture notes on the AKS sorting network [Electronic resource] // AiTS [Official website]. URL: <https://users.encs.concordia.ca/~chvatal/notes/aks.pdf> (accessed: 12.03.2021)
9. Joel I. Seiferas. Sorting Networks of Logarithmic Depth, Further Simplified [Electronic resource] // ResearchGate [Official website]. URL: https://www.researchgate.net/publication/263074945\_Sorting\_Networks\_of\_Logarithmic\_Depth\_Further\_Simplified (accessed: 12.04.2021)
10. A glimpse at the AKS network [Electronic resource] // University of Cambridge [Official website]. URL: <https://www.cl.cam.ac.uk/teaching/1415/AdvAlgo/lec2_ann.pdf> (accessed: 20.03.2021)
11. Четно-нечетная сортировка слиянием Бэтчера [Электронный ресурс] // Habr [Официальный сайт]. URL: <https://habr.com/ru/post/261777/> (дата обращения: 29.12.2020)
12. Miklós Ajtai, Janos Komlos, Endre Szemerédi. An O (n log n) Sorting Network [Electronic resource] // ResearchGate [Official website]. URL: <https://www.researchgate.net/publication/221590321_An_On_log_n_Sorting_Network> (accessed: 10.02.2021)
13. Sorting network [Electronic resource] // Wikipedia [Official website]. URL: https://en.wikipedia.org/wiki/Sorting\_network (accessed: 05.01.2021).
14. Сортирующие сети [Электронный ресурс] // Университет ИТМО [Официальный сайт]. URL: <https://neerc.ifmo.ru/wiki/index.php?title=Сортирующие_сети> (дата обращения: 21.12.2020)
15. Сеть Бэтчера [Электронный ресурс] // Университет ИТМО [Официальный сайт]. URL: https://neerc.ifmo.ru/wiki/index.php?title=Сеть\_Бетчера (дата обращения: 24.02.2021)
16. Параллельная сортировка Бэтчера [Электронный ресурс] // Мегапредмет [Официальный сайт]. URL: <https://megapredmet.ru/1-46160.html> (дата обращения: 15.04.2021)
17. 0-1 принцип [Электронный ресурс] // Университет ИТМО [Официальный сайт]. URL: https://neerc.ifmo.ru/wiki/index.php?title=0-1\_принцип (дата обращения: 22.02.2021)
18. The 0-1-principe [Electronic resource] // Innovative Hochschule Flensburg [Official website]. URL: <https://www.inf.hs-flensburg.de/lang/algorithmen/sortieren/networks/nulleinsen.htm> (accessed: 07.02.2021)
19. Michael T. Goodrich Zig-zag Sort: A Simple Deterministic Data-Oblivious Sorting Algorithm Running in O(n log n) Time [Electronic resource] // arXiv [Official website]. URL: https://arxiv.org/pdf/1403.2777.pdf (accessed: 30.01.2021)
20. Miklós Ajtai, Janos Komlos, Endre Szemerédi. Halvers and expanders [Electronic resource] // ResearchGate [Official website]. URL: https://www.researchgate.net/publication/3513473\_Halvers\_and\_expanders\_switching (accessed: 30.01.2021)
21. std::mt19937 [Electronic resource] // Cplusplus [Official website]. URL: <https://www.cplusplus.com/reference/random/mt19937/> (accessed: 11.04.2021)
22. Date and time utilities [Electronic resource] // CppReference [Official website]. URL: https://en.cppreference.com/w/cpp/chrono (accessed: 11.04.2021)
23. non-adaptive-sorts-research [Experiment repository] // Github [Official website]. URL: <https://github.com/Igor-SeVeR/non-adaptive-sorts-research> (accessed: 20.04.2021)
24. Robert Sedgewick. Analysis of Shellsort and Related Algorithms [Electronic resource] // Semantic scholar [Official website]. URL: <http://thomas.baudel.name/Visualisation/VisuTri/Docs/shellsort.pdf> (accessed: 11.04.2021)
25. Параллельное программирование на OpenMP [Электронный ресурс] // Новосибирский государственный университет [Официальный сайт]. URL: http://ccfit.nsu.ru/arom/data/openmp.pdf (дата обращения: 19.01.2021)