Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Российский экономический университет имени Г.В. Плеханова»
Высшая школа кибертехнологий, математики и статистики
Кафедра Прикладной информатики и информационной безопасности

ОТЧЕТ

по выполнению лабораторных работ 6.1-6.7 (ТОП-AI уровень).

«Автоматы, внешняя память, геометрия, нечёткие множества, теория игр, машинное обучение и нейронные сети»

Направление 09.03.03 Прикладная информатика

Профиль: Инжиниринг предприятий и информационных систем
Прикладная информатика в экономике

Уровень: Бакалавриат

Дисциплина: Структуры данных и алгоритмов

Выполнили: студенты гр. ПИ05/25б
1 курс, ИЦЭиИТ
Ахтямов Николай Сергеевич_____
(подпись)
Проверил:
Попов А.А.
(должность, ФИО преподавателя)
_____
(оценка), подпись)
_____
(дата)

Москва, 2025

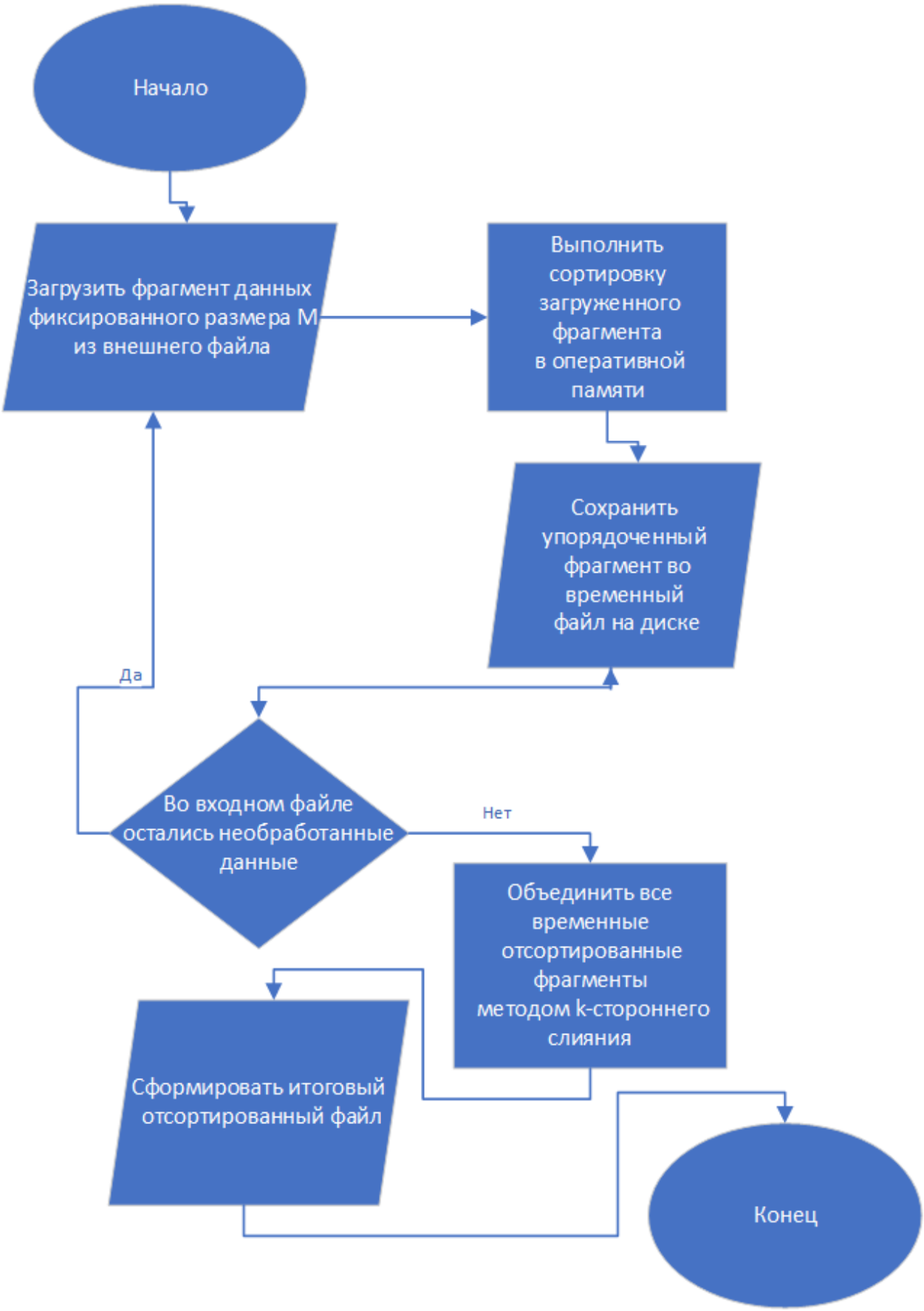
Предметная область: Алгоритмы во внешней памяти

Алгоритм №1. External Merge Sort

1. **Назначение и область применения алгоритма**

External Merge Sort предназначен для сортировки наборов данных, объём которых превышает возможности оперативной памяти. Алгоритм основан на разбиении исходных данных на небольшие блоки, которые поочерёдно загружаются в оперативную память и сортируются. После этого отсортированные блоки сохраняются во внешней памяти и объединяются между собой с помощью операции слияния. Такой подход обеспечивает эффективную сортировку больших файлов и снижает количество обращений к внешним устройствам хранения данных.

2. **Блок-схема алгоритма**


Начало
Загрузить фрагмент данных фиксированного размера M из внешнего файла
Выполнить сортировку загруженного фрагмента в оперативной памяти
Сохранить упорядоченный фрагмент во временный файл на диске
Да
Во входном файле остались необработанные данные
Нет
Объединить все временные отсортированные фрагменты методом k-стороннего слияния
Сформировать итоговый отсортированный файл
Конец

3. **Пошаговый анализ работы алгоритма.**

1) Инициализация алгоритма – производится запуск алгоритма, подготавливаются структуры данных и открывается файл с исходными неотсортированными данными, расположенный во внешней памяти.
2) Загрузка очередного блока данных – из внешнего файла считывается часть данных фиксированного размера *M*, объём которой не превышает доступную оперативную память.
3) Внутренняя сортировка **блока** – загруженный блок полностью размещается в RAM и сортируется с использованием внутреннего алгоритма сортировки.
4) Сохранение отсортированного блока – результат сортировки записывается во временный файл на внешнем носителе, формируя один из отсортированных фрагментов данных.
5) Проверка завершения разбиения – определяется, остались ли во входном файле необработанные элементы.

- Если данные ещё присутствуют**,** выполняется возврат к шагу 2 для обработки следующего блока.
- Если данные закончились**,** алгоритм переходит к этапу объединения.

6) Многопутевое слияние отсортированных фрагментов – все временные файлы открываются одновременно, и из них поочерёдно выбираются минимальные элементы, формируя единую упорядоченную последовательность.
7) Формирование выходного файла – полученные в процессе слияния элементы записываются в результирующий файл в отсортированном виде.
8) Завершение работы алгоритма – закрываются все файлы, временные ресурсы освобождаются, алгоритм завершает выполнение.

4. **Программный код на языке Python.**

```
import heapq
import os
def external_merge_sort(input_file, output_file, block_size):
```

```
# Список временных файлов
temp_files = []
# ===== Фаза 1: чтение блоков и сортировка =====
with open(input_file, "r") as fin:
    idx = 0
    while True:
        block = []

        # Считывание блока размера block_size
        for _ in range(block_size):
            line = fin.readline()
            if not line:
                break
            block.append(int(line.strip()))
        if not block:
            break

        # Сортировка блока в RAM
        block.sort()

        # Запись отсортированного блока во временный файл
        temp_name = f"run_{idx}.txt"
        idx += 1
        with open(temp_name, "w") as fout:
            for x in block:
                fout.write(f"{x}\n")

        temp_files.append(temp_name)

# ===== Фаза 2: k-стороннее слияние =====
files = [open(name, "r") for name in temp_files]
heap = []

# Инициализация кучи первыми элементами файлов
for i, f in enumerate(files):
    line = f.readline()
    if line:
        heapq.heappush(heap, (int(line.strip()), i))

# Формирование итогового отсортированного файла
with open(output_file, "w") as out:
    while heap:
        value, idx = heapq.heappop(heap)
        out.write(f"{value}\n")
```

```
            line = files[idx].readline()
            if line:
                heapq.heappush(heap, (int(line.strip()), idx))

    # Завершение: закрытие и удаление временных файлов
    for f in files:
        f.close()
    for name in temp_files:
        os.remove(name)
```

5. Временная сложность алгоритма

Алгоритм External Merge Sort состоит из двух основных этапов: сортировки блоков во внешней памяти и их последующего слияния.

На первом этапе входные данные объёма N разбиваются на блоки размера M, каждый из которых сортируется в оперативной памяти. Временная сложность этого этапа составляет $O(N \cdot \log M)$, так как сортировка выполняется для каждого блока отдельно.

На втором этапе выполняется k-стороннее слияние отсортированных блоков, где k – количество временных файлов. С использованием приоритетной очереди операция выбора минимального элемента выполняется за $O(\log k)$, а всего обрабатывается N элементов. Следовательно, временная сложность этапа слияния равна $O(N \cdot \log k)$.

Таким образом, общая временная сложность алгоритма External Merge Sort оценивается как:

$O(N \cdot \log M + N \cdot \log k)$, что делает данный алгоритм эффективным для сортировки больших объёмов данных, не помещающихся в оперативную память.

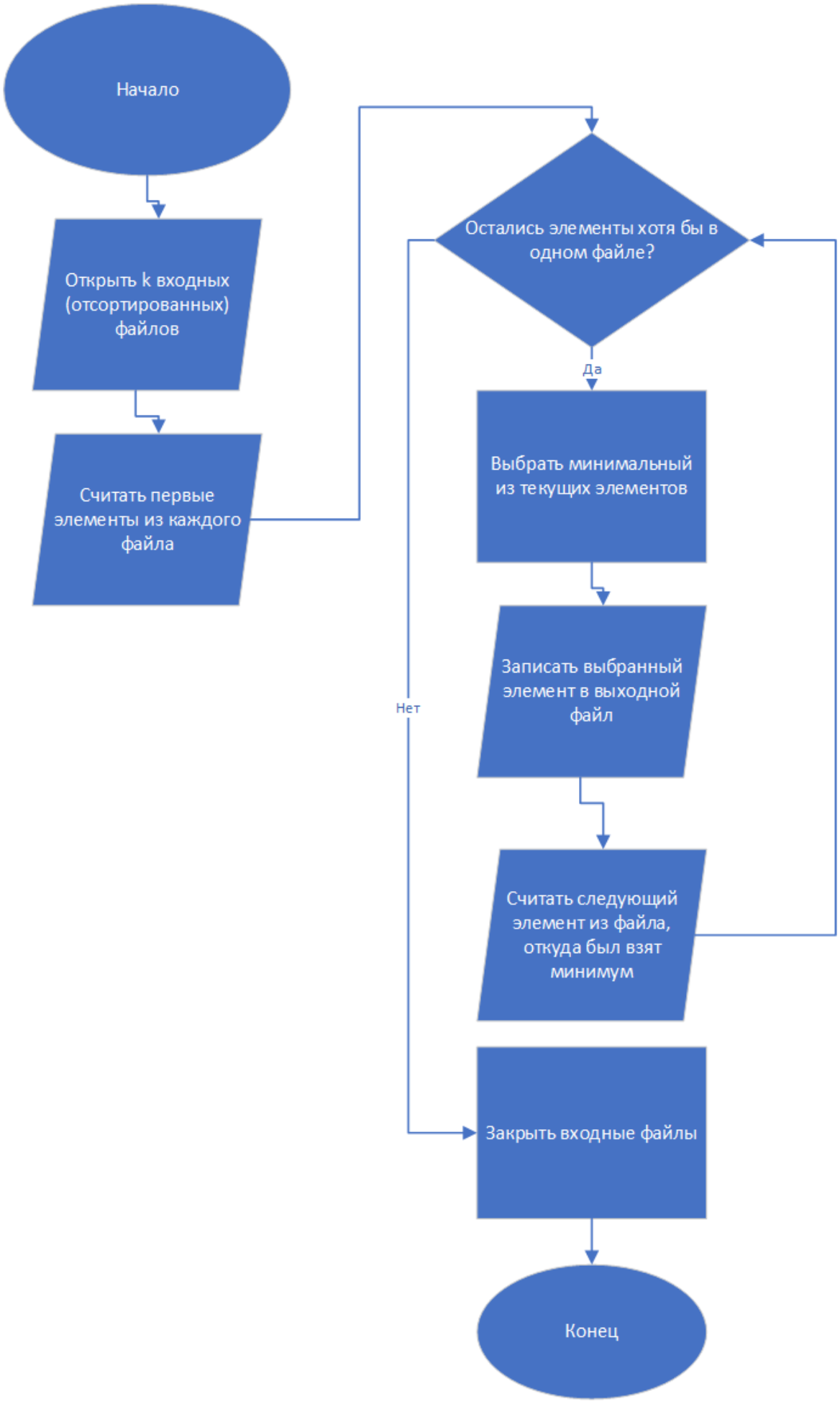
Алгоритм №2. k-way External Merge (многопутевое слияние)

1. **Назначение и область применения алгоритма**

k-way External Merge предназначен для объединения k заранее отсортированных файлов (серий), расположенных во внешней памяти, в один общий отсортированный выходной файл. Алгоритм используется как основной этап внешней сортировки: после того как данные разбиты на отсортированные блоки (runs), требуется собрать их в один упорядоченный файл.

Главная цель – выполнять слияние, минимизируя операции чтения/записи за счёт последовательного чтения из файлов и поэлементного формирования результата.

2. **Блок-схема алгоритма**


Начало
Открыть k входных (отсортированных) файлов
Считать первые элементы из каждого файла
Остались элементы хотя бы в одном файле?
Да
Нет
Выбрать минимальный из текущих элементов
Записать выбранный элемент в выходной файл
Считать следующий элемент из файла, откуда был взят минимум
Закрыть входные файлы
Конец

3. **Пошаговый анализ работы алгоритма.**

1) Начало – подготовка ресурсов и открытие выходного файла для записи результата.
2) Открытие k файлов – открываются все входные отсортированные файлы (серии), которые нужно слить.
3) Инициализация текущих значений – из каждого файла считывается первый элемент (или фиксируется, что файл пуст).
4) Проверка наличия данных – определяется, остались ли элементы хотя бы в одном файле.
5) Выбор минимального элемента – среди текущих элементов всех файлов выбирается наименьший.
6) Запись в результат – выбранный минимальный элемент добавляется в выходной файл.
7) Обновление источника – из того файла, откуда был взят минимум, считывается следующий элемент и заменяет текущий.
8) Повторение цикла – шаги 4–7 повторяются, пока все входные файлы не будут полностью обработаны.
9) Конец – файлы закрываются, выходной файл содержит полностью отсортированную последовательность.

4. **Пошаговый код на языке Python.**

```
import heapq


def k_way_merge(input_files, output_file):
    # input_files: список путей к отсортированным файлам
    # output_file: путь к результирующему файлу

    files = [open(name, "r") for name in input_files]
    heap = []

    # Считываем первый элемент каждого файла и кладём в кучу
    for i, f in enumerate(files):
        line = f.readline()
        if line:
            heapq.heappush(heap, (int(line.strip()), i))
```

```
# Пока есть элементы – выбираем минимальный и дописываем результат
with open(output_file, "w") as out:
    while heap:
        value, idx = heapq.heappop(heap)
        out.write(f"{value}\n")

        # Берём следующий элемент из того же файла
        nxt = files[idx].readline()
        if nxt:
            heapq.heappush(heap, (int(nxt.strip()), idx))

# Закрываем входные файлы
for f in files:
    f.close()
```

5. Временная сложность алгоритма

Пусть N – общее количество элементов во всех входных файлах, а k – число файлов (серий). Для выбора минимального текущего элемента удобно использовать структуру данных “куча” (heap), где операции извлечения минимума и добавления нового элемента выполняются за **O(log k)**. Поскольку таких операций выполняется N раз, общая временная сложность алгоритма составляет: **O(N · log k)**.

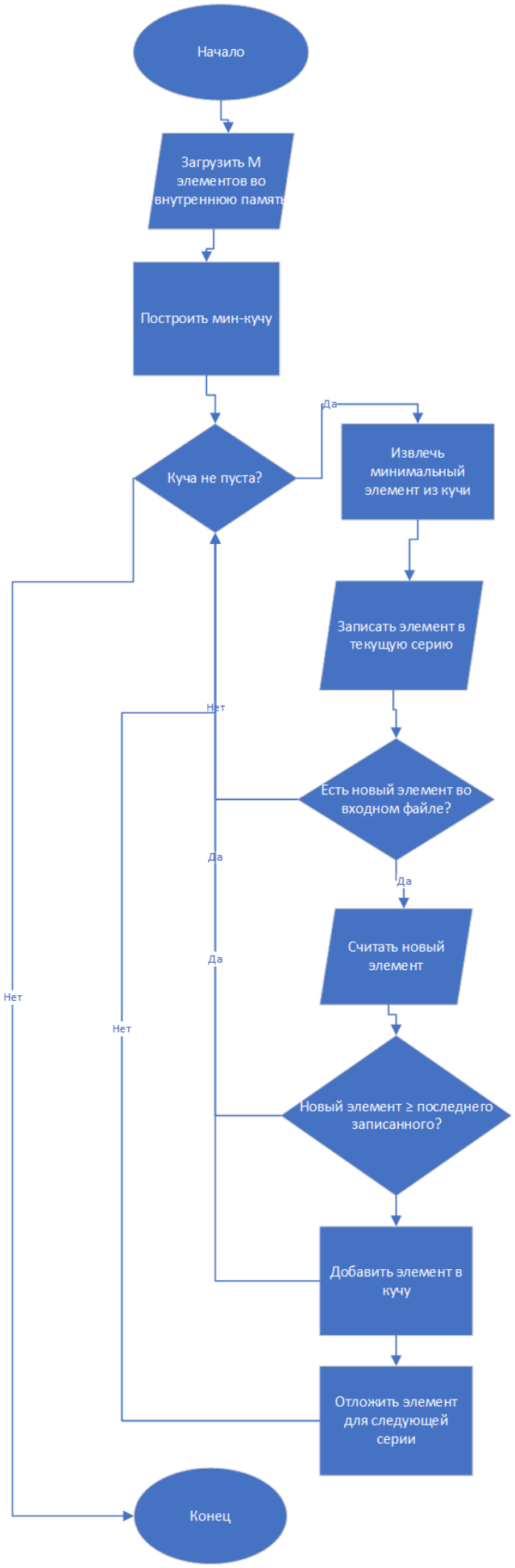
Алгоритм №3. Replacement Selection

1. **Назначение и область применения алгоритма**

Replacement Selection предназначен для формирования длинных отсортированных серий (runs) во внешней памяти при ограниченном объёме оперативной памяти. Алгоритм используется на этапе предварительной обработки данных перед внешней сортировкой и позволяет получать серии, средняя длина которых превышает размер оперативной памяти.

Основная идея алгоритма заключается в использовании структуры данных (обычно кучи) для выбора минимального элемента и динамической замены его новыми элементами, поступающими из внешней памяти.

2. **Блок-схема алгоритма**


Начало
Загрузить M элементов во внутреннюю память
Построить мин-кучу
Куча не пуста?
Да
Нет
Извлечь минимальный элемент из кучи
Записать элемент в текущую серию
Есть новый элемент во входном файле?
Да
Нет
Считать новый элемент
Новый элемент ≥ последнего записанного?
Да
Нет
Добавить элемент в кучу
Отложить элемент для следующей серии
Нет
Конец

3. **Пошаговый анализ работы алгоритма.**

1) Начало – инициализация алгоритма и открытие входного файла.
2) Загрузка данных – во внутреннюю память загружается *M* элементов, из которых формируется начальная мин-куча.
3) Проверка состояния кучи – если куча пуста, работа алгоритма завершается.
4) Извлечение минимума – из кучи извлекается минимальный элемент, который записывается в текущую отсортированную серию.
5) Чтение нового элемента – из входного файла считывается следующий элемент, если он существует.
6) Сравнение с последним элементом серии – если новый элемент не нарушает порядок сортировки, он добавляется в текущую кучу.
7) Отложенные элементы – элементы, которые меньше последнего записанного, откладываются для формирования следующей серии.
8) Формирование новой серии – после опустошения кучи начинается формирование следующей серии.
9) Конец – все данные разбиты на отсортированные серии во внешней памяти.

4. **Программный код на языке Python.**

```
import heapq

def replacement_selection(input_file, output_file, M):
    heap = []
    deferred = []

    with open(input_file, "r") as fin, open(output_file, "w") as out:
        # Загрузка первых M элементов
        for _ in range(M):
            line = fin.readline()
            if not line:
                break
            heapq.heappush(heap, int(line.strip()))

        last_output = None
```

```
while heap:
    # Извлечение минимального элемента
    smallest = heapq.heappop(heap)
    out.write(f"{smallest}\n")
    last_output = smallest

    line = fin.readline()
    if line:
        value = int(line.strip())
        if value >= last_output:
            heapq.heappush(heap, value)
        else:
            deferred.append(value)

    # Если куча пуста – начинаем новую серию
    if not heap and deferred:
        for x in deferred:
            heapq.heappush(heap, x)
        deferred.clear()
```

5. **Временная сложность алгоритма**

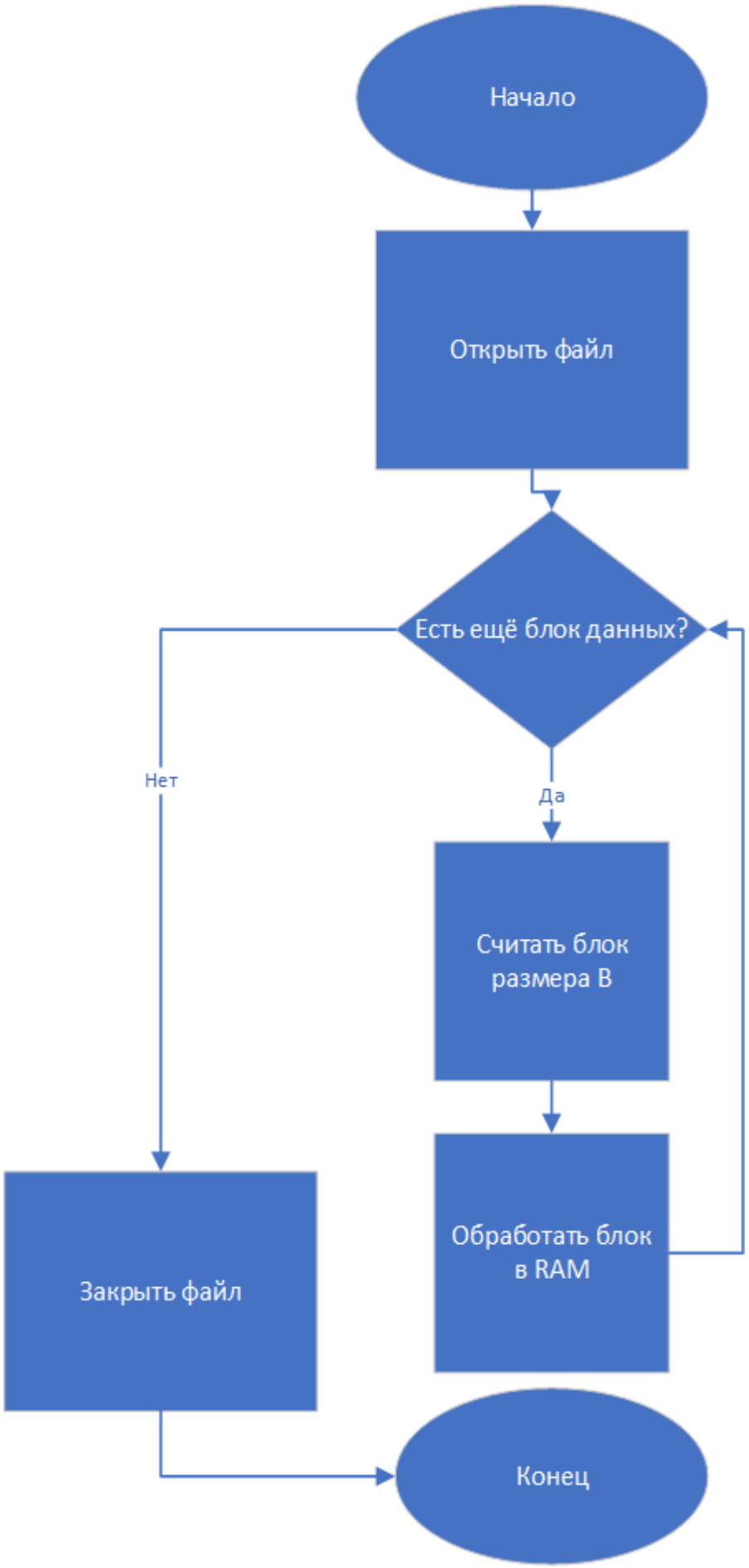
Пусть N – общее количество элементов, M – размер внутренней памяти. Операции вставки и извлечения из кучи выполняются за O(log M). Поскольку каждая операция применяется к каждому элементу входных данных, временная сложность алгоритма составляет O(N · log M).
С точки зрения внешней памяти алгоритм выполняет последовательное чтение и запись данных, поэтому количество операций ввода-вывода оценивается как O(N / B), где B – размер блока.

Алгоритм №4. Block Scan

1. **Назначение и область применения алгоритма**

Block Scan предназначен для обработки больших файлов, которые не помещаются в оперативную память. Данные читаются последовательно блоками размера B из внешней памяти (диск/SSD), каждый блок обрабатывается в RAM и затем читается следующий. Это базовый приём, который используется почти во всех алгоритмах внешней памяти (сортировка, поиск, агрегации).

2. **Блок-схема алгоритма**


Начало
Открыть файл
Есть ещё блок данных?
Нет
Да
Считать блок размера B
Обработать блок в RAM
Закрыть файл
Конец

3. **Пошаговый анализ работы алгоритма.**

1) Открыть входной файл.
2) Проверить, остались ли данные для чтения.
3) Считать следующий блок размера B во внутреннюю память.
4) Выполнить обработку блока (например, подсчёт суммы, поиск, фильтрация).
5) Повторять шаги 2–4, пока блоки не закончатся.
6) Закрыть файл и завершить алгоритм.

4. **Программный код на языке Python.**

```
def block_scan_sum(input_file, B):
    total = 0

    with open(input_file, "r") as f:
        while True:
            block = []
            for _ in range(B):
                line = f.readline()
                if not line:
                    break
                block.append(int(line.strip()))

            if not block:
                break

            # обработка блока в RAM
            total += sum(block)

    return total
```

5. **Временная сложность алгоритма**

В Время обработки: O(N) (каждый элемент читается и обрабатывается один раз).

I/O-сложность: чтение идёт блоками, поэтому число операций ввода примерно O(N / B).