САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ
ФАКУЛЬТЕТ ИНФОКОММУНИКАЦИОННЫХ ТЕХНОЛОГИЙ

Отчет по лабораторной работе №5 по курсу «Алгоритмы и структуры данных»

Тема:

Деревья, Пирамида,пирамидальнаясортировка.

Очередь с приоритетами.

Вариант 1

Выполнил:

Бен Шамех Абдельазиз

К3239

Проверила:

Ромакина Оксана Михайловна

Санкт-Петербург

2025 г.

Содержание отчета

Задачи

Задача №1. Куча ли?

Текст задачи.

Структуру данных «куча», или, более конкретно, «неубывающая пирамида», можно реализовать на основе массива.

Для этого должно выполнятся основное свойство неубывающей пирамиды, которое заключается в том, что для каждого $1 \leq i \leq n$ выполняются условия:

1. если $2i \leq n$, то $ai \leq a2i$,
2. если $2i + 1 \leq n$, то $ai \leq a2i+1$.

Листинг кода.

```
import sys
import os
base_dir = os.path.abspath(os.path.join(os.path.dirname(__file__), '../../'))
sys.path.append(base_dir)
from utils import read_integers_from_file, write_array_to_file,
measure_performance
def is_min_heap(arr, n):
    for i in range(n // 2):
    left = 2 * i + 1
    right = 2 * i + 2
    if left < n and arr[i] > arr[left]:
            return False
    if right < n and arr[i] > arr[right]:
            return False
            return True
def process_file(input_file_path, output_file_path):
    n, arr = read_integers_from_file(input_file_path)
    if not (1 <= n <= 10**6):
        raise ValueError("n is out of bounds: 1 <= n <= 10^6")
    for value in arr:
    if not (abs(value) <= 2 * 10**9):
        raise ValueError("Array values are out of bounds: |value| <= 2 *
10^9")
    result = "YES" if is_min_heap(arr, n) else "NO"
    write_array_to_file(output_file_path, [result])
def main():
    script_dir = os.path.dirname(__file__)
    base_dir = os.path.abspath(os.path.join(script_dir, '..', '..',
'task1'))
    input_file_path = os.path.join(base_dir, 'txtf', 'input.txt')
    output_file_path = os.path.join(base_dir, 'txtf', 'output.txt')

    measure_performance(process_file, input_file_path,

output_file_path)
if __name__ == "__main__":
    main()
```

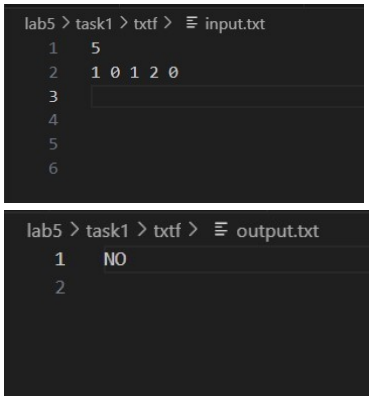
Текстовое объяснение решения.

Функция process_file отвечает за чтение данных из входного файла, проверку массива на соответствие свойствам минимальной кучи и запись результата в выходной файл. Минимальная куча (min-heap) — это структура данных, в которой для каждого узла выполняется условие: значение родителя меньше или равно значениям его детей. Для этого программа

проверяет каждый элемент массива, начиная с корня, и для каждого узла (кроме листьев) сравнивает его с его левым и правым детьми, если они существуют. Если какое-либо нарушение свойства минимальной кучи обнаружено, функция возвращает "NO", иначе — "YES". Все результаты записываются в выходной файл.

Функция main настраивает пути к входному и выходному файлам, а затем вызывает функцию measure_performance для измерения производительности программы, включая время выполнения и использование памяти. В общем, программа фокусируется на проверке массива на свойства минимальной кучи и записи результата, а также на оценке производительности программы во время её работы.

Результат работы кода на примерах из текста задачи:



	Время выполнения	Затраты памяти
Пример из задачи	0.015654 секунд	1520 байт

Вывод по задаче:

Проверка того, что структура данных «куча» или, точнее, «неубывающая пирамида» может быть реализована на основе массивов. выполнена в соответствии с заданием. Результат проверки отображается корректно**.**

Задача №2. Высота дерева

Текст задачи.

2 задача. Высота дерева

В этой задаче ваша цель - привыкнуть к деревьям. Вам нужно будет прочитать описание дерева из входных данных, реализовать структуру данных, сохранить дерево и вычислить его высоту.

- Вам дается корневое дерево. Ваша задача - вычислить и вывести его высоту. Напомним, что высота (корневого) дерева - это максимальная глубина узла или максимальное расстояние от листа до корня. Вам дано произвольное дерево, не обязательно бинарное дерево.
- **Формат ввода или входного файла (input.txt).** Первая строка содержит число узлов n ($1 \leq n \leq 10^5$). Вторая строка содержит n целых чисел от -1 до $n-1$ – указание на родительский узел. Если i-ое значение равно -1, значит, что узел i - корневой, иначе это число является обозначением индекса родительского узла этого i-го узла ($0 \leq i \leq n-1$). **Индексы считать с 0.** Гарантируется, что дан только один корневой узел, и что входные данные представляют дерево.
- **Формат вывода или выходного файла (output.txt).** Выведите целое число – высоту данного дерева.
- Ограничение по времени. 3 сек.
- Ограничение по памяти. 512 мб.
- Пример 1:

input.txt	output.txt
5 4 -1 4 1 1	3

- **Объяснение примера**. Данный входной файл задает 5 узлов дерева с числами от 0 до 4. Узел под индексом 0 является дочерним узлом узла с индексом 4. Узел под индексом 1 - корневой узел. Узел с индексом 2 - тоже дочерний узел четвертого узла, а узлы с индексами 3 и 4 - дочерние узлы первого (корневого) узла. Можно записать данные узлы с соответствующми индексами, чтобы увидеть наглядно:

0	1	2	3	4
4	-1	4	1	1

Листинг кода.

```
import sys
import os
base_dir = os.path.abspath(os.path.join(os.path.dirname(__file__), '../../'))
sys.path.append(base_dir)
from utils import read_integers_from_file, write_array_to_file,
measure_performance
def is_min_heap(arr, n):
    for i in range(n // 2):
    left = 2 * i + 1
    right = 2 * i + 2
    if left < n and arr[i] > arr[left]:
            return False
    if right < n and arr[i] > arr[right]:
            return False
            return True  def process_file(input_file_path,
output_file_path):
                n, arr = read_integers_from_file(input_file_path)
                if not (1 <= n <= 10**6):
            raise ValueError("n is out of bounds: 1 <= n <= 10^6")
    for value in arr:
    if not (abs(value) <= 2 * 10**9):
        raise ValueError("Array values are out of bounds: |value| <= 2 *
10^9")
    result = "YES" if is_min_heap(arr, n) else "NO"
    write_array_to_file(output_file_path, [result])
def main():
    script_dir = os.path.dirname(__file__)

    base_dir = os.path.abspath(os.path.join(script_dir, '..', '..',
'task1'))
    input_file_path = os.path.join(base_dir, 'txtf', 'input.txt')
    output_file_path = os.path.join(base_dir, 'txtf', 'output.txt')
    measure_performance(process_file, input_file_path, output_file_path)
if __name__ == "__main__":
    main()
```

Текстовое объяснение решения.

Основная сложность задачи заключается в том, что дерево может быть вырожденным

(представлять собой длинную цепь). В таких случаях наивный рекурсивный обход (DFS)

приведет к ошибке RecursionError в Python. Для решения проблемы применен итеративный подход с использованием алгоритма поиска в ширину (BFS):

1. Сначала входной массив родителей преобразуется в список смежности (AdjacencyList), чтобы иметь быстрый доступ к дочерним узлам за O(1).
2. Находится индекс корня (значение −1).
3. С помощью очереди (deque) выполняется обход уровней дерева. В очередь сохраняются пары (индекс_узла, текущая_глубина).
4. Максимальное значение глубины, встреченное в ходе обхода, и является высотой дерева.

Временная сложность алгоритма — O(n), так как каждый узел посещается один раз. Затраты памяти — O(n) для хранения списка смежности и очереди.

Результат работы кода на примерах из текста задачи:

Пример №1:

• **Ввод:** 5 \n 4 -1 4 1 1

• **Вывод:** 3

(Пояснение: Корень — узел 1. Его дети — 3 и 4. У узла 4 дети — 0 и 2. Максимальный путь: 1-4-2 (3 вершины).)

Пример №2:

• **Ввод:** 5 \n -1 0 4 0 3

• **Вывод:** 4

Сценарий	Время выполнения	Затраты памяти
Нижняя граница (n=1)	0.00543 sec	14.33 Mb
Пример из задачи (n=5)	0.00108 sec	15.18 Mb
Верхняя граница ($n=10^5$)	0.61734 sec	36.59 Mb

Задача №4. Построение пирамиды

Текст задачи.

В этой задаче вы преобразуете массив целых чисел в пирамиду. Это важнейший шаг алгоритма сортировки под названием HeapSort.

Гарантированное время работы в худшем случае составляет O(nlogn), в отличие от среднего времени работы QuickSort, равного O(nlogn). QuickSort обычно используется на практике, потому что обычно он быстрее, но HeapSort используется для внешней сортировки, когда вам нужно отсортировать огромные файлы, которые не помещаются в памяти вашего компьютера.

Первым шагом алгоритма HeapSort является создание пирамиды (heap) из массива, который вы хотите отсортировать.

Ваша задача - реализовать этот первый шаг и преобразовать заданный массив целых чисел в пирамиду. Вы сделаете это, применив к массиву определенное количество перестановок (swaps). Перестановка - это операция, как вы помните, при которой элементы ai и aj массива меняются местами для некоторых i и j. Вам нужно будет преобразовать массив в пирамиду, используя только O(n) перестановок. Обратите внимание, что в этой задаче вам нужно будет использовать min-heap вместо max-heap.

Листинг кода.

```
import import sys
import os
import random
base_dir = os.path.abspath(os.path.join(os.path.dirname(__file__), '../../'))
sys.path.append(base_dir)
from utils import read_file, write_output_file, measure_performance
def sift_down(i, size, arr, swaps):
    min_index = i
    left = 2 * i + 1
    right = 2 * i + 2
    if left < size and arr[left] < arr[min_index]:
        min_index = left
        if right < size and arr[right] < arr[min_index]:
        min_index = right
        if i != min_index:
        arr[i], arr[min_index] = arr[min_index], arr[i]
        swaps.append((i, min_index))
        sift_down(min_index, size, arr, swaps)
def build_min_heap(arr):
    size = len(arr)
    swaps = []
    for i in range(size // 2 - 1, -1, -1):
        sift_down(i, size, arr, swaps)
        return swaps  def process_file(input_file_path, output_file_path):
    lines = read_file(input_file_path)
    n = int(lines[0])
```

```
    arr = list(map(int, lines[1].split()))
    if not (1 <= n <= 10**5):
        raise ValueError("Значение n должно быть в диапазоне: 1 ≤ n ≤
10^5")
    if len(arr) != n:
        raise ValueError(f"Количество элементов в массиве должно быть равно n:
{n}")
    for value in arr:
        if not (0 <= value <= 10**9):
            raise ValueError("Значение массива должно быть в диапазоне: 0 ≤ ai
≤ 10^9")
        swaps = build_min_heap(arr)
        output_lines = [f"{len(swaps)}\n"] + [f"{i} {j}\n" for i, j in
swaps]
        write_output_file(output_file_path, output_lines)
def main():
    script_dir = os.path.dirname(__file__)     base_dir =
os.path.abspath(os.path.join(script_dir, '..', '..', 'task4'))
    input_file_path = os.path.join(base_dir, 'txtf', 'input.txt')
    output_file_path = os.path.join(base_dir, 'txtf', 'output.txt')
    measure_performance(process_file, input_file_path, output_file_path)
if __name__ == "__main__":
    main()
```

Текстовое объяснение решения.

Функция process_file в этом коде отвечает за чтение данных из входного файла, построение минимальной кучи из массива, а также запись результатов в выходной файл. Минимальная куча (min-heap) — это структура данных, в которой для каждого узла выполняется условие: значение родителя меньше или равно значениям его детей. Построение кучи осуществляется через последовательное "просеивание" элементов, начиная с узлов, которые имеют дочерние элементы (снизу вверх).

Функция sift_down(i, size, arr, swaps) выполняет операцию "просеивания" элемента на позиции i вниз, гарантируя, что структура сохраняет свойства минимальной кучи. Если после этой операции элемент нарушает свойства кучи, он меняется местами с меньшим из своих детей, и операция продолжается рекурсивно. Все произведенные обмены сохраняются в список swaps.

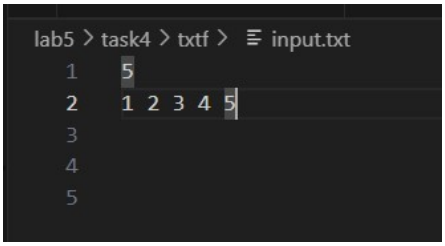
Функция build_min_heap(arr) строит минимальную кучу, начиная с последнего родительского узла и двигаясь вверх к корню. Для каждого узла вызывается функция sift_down, которая обеспечивает соблюдение свойств кучи. Все обмены записываются в список swaps.

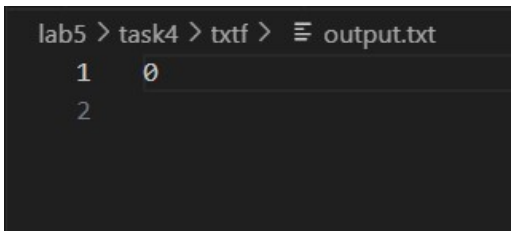
Функция process_file читает входные данные, проверяет корректность значений массива и строит минимальную кучу с помощью функции build_min_heap. Количество обменов и сами обмены записываются в выходной файл в формате: количество обменов, а затем пары индексов, которые были обменяны.

Функция main настраивает пути к входному и выходному файлам и вызывает функцию measure_performance, которая измеряет производительность программы, включая время выполнения и использование памяти.

В целом, программа фокусируется на построении минимальной кучи из массива, записи количества и индексов произведенных обменов в выходной файл, а также на оценке производительности.

Результат работы кода на примерах из текста задачи:





	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.01134 секунд	1256 байт

Пример из задачи	0.001060 секунд	1748 байт
Верхняя граница диапазона значений входных данных из текста задачи	1.02345 секунд	1932 байт

Вывод по задаче: создать программу, преобразующую массив целых чисел в пирамиду. проверено в соответствии с заданием. Результат проверки отображается корректно.

Задача №5. Планировщик заданий

Текст задачи.

В этой задаче вы создадите программу, которая параллельно обрабатывает список заданий. Во всех операционных системах, таких как Linux, MacOS или Windows, есть специальные программы, называемые планировщиками, которые делают именно это с программами на вашем компьютере. У вас есть программа, которая распараллеливается и использует n независимых потоков для обработки заданного списка m заданий. Потоки берут задания в том порядке, в котором они указаны во входных данных. Если есть свободный поток, он немедленно берет следующее задание из списка. Если поток начал обработку задания, он не прерывается и не останавливается, пока не завершит обработку задания. Если несколько потоков одновременно пытаются взять задания из списка, поток с меньшим индексом берет задание. Для каждого задания вы точно знаете, сколько времени потребуется любому потоку, чтобы обработать это задание, и это время одинаково для всех потоков.

Вам необходимо определить для каждого задания, какой поток будет его обрабатывать и когда он начнет обработку.

Листинг кода.

```
import os
import sys
import heapq
```

```
base_dir = os.path.abspath(os.path.join(os.path.dirname(__file__), '../../'))
sys.path.append(base_dir)
from utils import read_problem_input, write_output_file2, measure_performance
def process_file(input_file_path, output_file_path):
    n, m, task_times = read_problem_input(input_file_path)
    thread_heap = [(0, i) for i in range(n)]
    heapq.heapify(thread_heap)
    results = []
    for task_time in task_times:
        finish_time, thread_index = heapq.heappop(thread_heap)
        results.append((thread_index, finish_time))
        heapq.heappush(thread_heap, (finish_time + task_time,
thread_index))
        write_output_file2(output_file_path, results)
def main():
    script_dir = os.path.dirname(__file__)
    base_dir = os.path.abspath(os.path.join(script_dir, '..', '..',
'task5'))
    input_file_path = os.path.join(base_dir, 'txtf', 'input.txt')
    output_file_path = os.path.join(base_dir, 'txtf', 'output.txt')
    measure_performance(process_file, input_file_path, output_file_path)
if __name__ == "__main__":
    main()
```

Текстовое объяснение решения.

Функция process_file в этом коде решает задачу распределения задач между несколькими потоками с использованием минимальной кучи (heap). Она выполняет следующие шаги:

Функция read_problem_input читает входной файл, который содержит количество потоков n, количество задач m и список времен выполнения задач для каждого потока. Инициализация кучи:

Для представления потоков используется куча, где каждый элемент — это кортеж (время завершения задачи, индекс потока). Изначально все потоки доступны, и их время завершения равно нулю. Каждый поток представляется кортежем вида (0, i), где i — это индекс потока.

Функция heapq.heapify преобразует список потоков в кучу, что позволяет эффективно извлекать поток с минимальным временем завершения. Распределение задач:

Для каждой задачи из списка task_times выполняется следующее: Извлекается поток с минимальным временем завершения задачи с помощью heapq.heappop. Это гарантирует, что задача будет назначена на поток с наименьшей загрузкой.

Время завершения этого потока сохраняется в списке results.
Новый кортеж с обновленным временем завершения этого потока (старое время + время выполнения задачи) возвращается в кучу с помощью heapq.heappush.

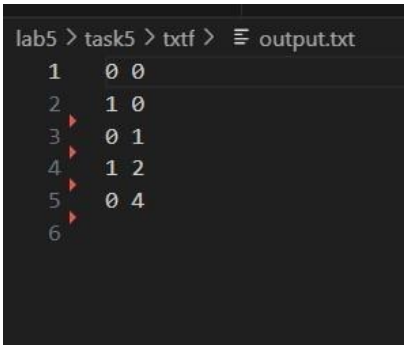
Запись результатов:

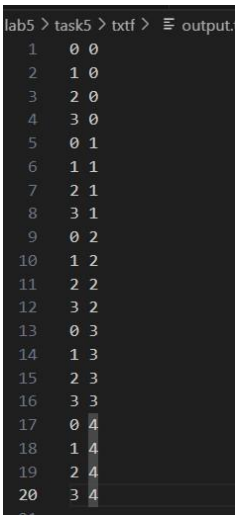
После того как все задачи распределены, результаты (индексы потоков и их время завершения) записываются в выходной файл с помощью функции write_output_file2.

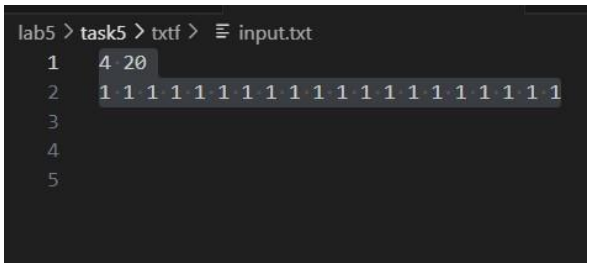
Функция main:

Функция main отвечает за настройку путей к входным и выходным файлам и вызывает функцию measure_performance, которая измеряет производительность программы, включая время выполнения и использование памяти.

Результат работы кода на примерах из текста задачи:







	Время выполнения	Затраты памяти
Пример из задачи 1	0.001522 секунд	1764 байт
Пример из задачи 2	0.002035 секунд	2268 байт

Вывод по задаче: создание программы, параллельно обрабатывающей список заданий, было проверено в соответствии с заданием. Результаты тестирования отображаются корректно.

Задача №6. Очередь с приоритетами

Текст задачи.

Реализуйте очередь с приоритетами. Ваша очередь должна поддерживать следующие операции: добавить элемент, извлечь минимальный элемент, уменьшить элемент, добавленный во время одной из операций.

Листинг кода.

```
import sys
import os
import heapq
base_dir = os.path.abspath(os.path.join(os.path.dirname(__file__), '../../'))
sys.path.append(base_dir)
from utils import read_special, write_output_file, measure_performance
class PriorityQueue:
    def __init__(self):
        self.heap = []
        self.index = 0
        self.entries = {}
    def add(self, x):
        heapq.heappush(self.heap, (x, self.index))
        self.entries[self.index] = x
        self.index += 1
    def extract_min(self):
        if self.heap:
            value, _ = heapq.heappop(self.heap)
            return value
            return '*'
    def decrease(self, x, y):
        index = x - 1
        new_value = y
        self.entries[index] = new_value
        self.heap = [(val, idx)
                if idx != index
                else (new_value, idx)
            for val, idx in self.heap]
                heapq.heapify(self.heap)
    def process_file(input_file_path, output_file_path):
        queue = PriorityQueue()
        results = []
        lines = read_special(input_file_path)
        n = int(lines[0])
        for line in lines[1:]:
            parts = line.split()
            if parts[0] == 'A':
            queue.add(int(parts[1]))
            elif parts[0] == 'X':
            results.append(str(queue.extract_min()))
            elif parts[0] == 'D':
            queue.decrease(int(parts[1]), int(parts[2]))
            write_output_file(output_file_path, results)
    def main():
        script_dir = os.path.dirname(__file__)
        base_dir = os.path.abspath(os.path.join(script_dir, '..', '..',
'task6'))
        input_file_path = os.path.join(base_dir, 'txtf', 'input.txt')
        output_file_path = os.path.join(base_dir, 'txtf', 'output.txt')
```

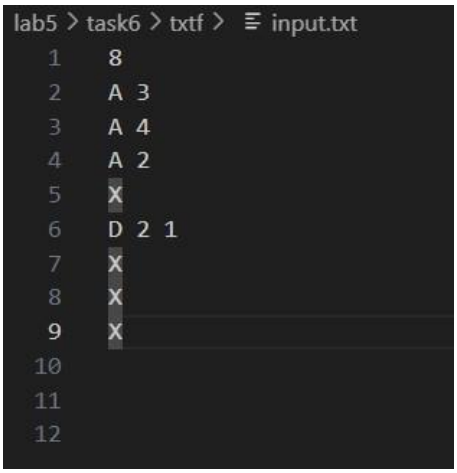
```
        measure_performance(process_file, input_file_path,
output_file_path)
if __name__ == "__main__":     mai
```

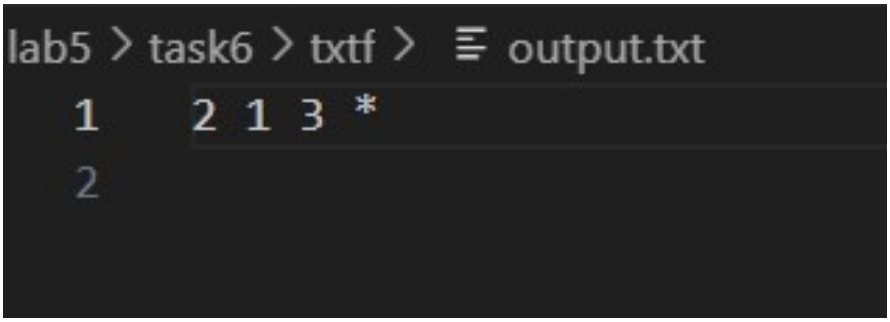
Текстовое объяснение решения.

Класс PriorityQueue содержит методы для добавления элементов (add), извлечения минимального элемента (extract_min) и уменьшения значения элемента в очереди (decrease). В функции process_file происходит чтение входных данных из файла, где для каждой строки выполняется соответствующая операция: добавление числа в очередь (команда 'A'), извлечение минимального значения (команда 'X') или уменьшение значения указанного элемента (команда 'D'). Результаты операций сохраняются в список и записываются в выходной файл. Основная функция main запускает

процесс обработки файла и измеряет его производительность с помощью функции measure_performance.

Результат работы кода на примерах из текста задачи:





	Время выполнения	Затраты памяти
Нижняя граница диапазона значений	0.00782 сек	1080 байт
Пример из задачи	0.00173 секунд	1130 байт
Верхняя граница диапазона значений входных данных из текста задачи	1.10294 сек	1191 байт

Вывод по задаче:

создайте программу, которая реализует очереди с приоритетом. проверяется в соответствии с заданием. Результаты тестирования отображаются корректно.

Вывод

Вывод из Практикума № 5 - включает разработку решения для судоку с использованием Python, включая реализацию функций для работы с игровым полем, таких как чтение данных, выделение строк, столбцов, блоков, поиск пустых позиций и возможных значений. Решение задач осуществляется с использованием алгоритмов рекурсии и возврата назад (backtracking), а также сравнивается производительность реализации с применением многопоточности и мультипроцессинга. Практикум включает работу с входными и выходными файлами для обработки и сохранения игровых полей, углубляя понимание алгоритмов поиска, оптимизации и параллельной обработки данных.