Министерство науки и высшего образования РФ
федеральное государственное автономное образовательное учреждение
высшего образования
«Омский государственный технический университет»
Факультет информационных технологий и компьютерных систем
Кафедра «Прикладная математика и фундаментальная информатика»

Отчет по лабораторным работам

по дисциплине **Алгоритмы и структуры данных**

Студента Гресь Владимир Игоревич

Курс 2 Группа ФИТм-241

Направление 02.04.02 Фундаментальная информатика и информационные технологии

Руководитель доц.,к.н. ФЕДОТОВА И.В.

Выполнил _____
дата, подпись студента

Проверил _____
дата, подпись руководителя

Омск 2025

Глава 1. Отчет

1.1 Задание 1. Сортировка стихов

В данном задании реализована программа для чтения стихов из файла с особой разметкой и сортировки их по возрастанию размера (длины).

Листинг 1.1: Парсинг и сортировка стихов

```
def parse_poetry_file(file_path):
    result = []
    current_title = None
    reading_text = False
    current_text = []
    try:
        with open(file_path, "r", encoding="utf-8") as file:
            for line in file:
                line = line.strip()
                if "|name_start|" in line:
                    title_part = line.split("|name_start|")[1]
                    if "|name_close|" in title_part:
                        current_title = title_part.split("|
name_close|")[0].strip()
                    else:
                        current_title = title_part.strip()
                elif "|name_close|" in line and current_title is
 not None:
                    current_title = current_title.split("|
name_close|")[0].strip()
                elif "|text_start|" in line:
                    reading_text = True
                    if line.strip() != "|text_start|":
                        current_text.append(line.split("|
text_start|")[1])
                elif "|text_close|" in line and reading_text:
                    reading_text = False
                    if line.strip() != "|text_close|":
                        current_text.append(line.split("|
text_close|")[0])
                    full_text = "\n".join(current_text)
                    if current_title:
```

```
                        result.append(
                            {"title": current_title, "length":
    len(full_text)}
                        )

                    current_title = None
                    current_text = []
                elif reading_text:
                    current_text.append(line)
        return result

    except Exception as e:
        print(f"error: {e}")
        return []


def bubble_sort(poems):
    n = len(poems)
    sorted_poems = poems.copy()
    for i in range(n):
        # если за проход не было обменов, массив отсортирован
        swapped = False
        for j in range(0, n - i - 1):
            if sorted_poems[j]["length"] > sorted_poems[j + 1]["
    length"]:
                sorted_poems[j], sorted_poems[j + 1] = (
                    sorted_poems[j + 1],
                    sorted_poems[j],
                )
                swapped = True
        if not swapped:
            break

    return sorted_poems
```

Комментарий: Программа считывает стихи из файла, используя специальные маркеры для выделения названия и текста стиха. Сортировка производится с помощью пузырькового метода, который останавливается, если за проход не было перестановок (оптимизация). Сортировка происходит по длине текста стиха.

1.2 Задание 2. Калькулятор выражений

Реализация калькулятора, который вычисляет арифметические выражения, содержащие 4 основных действия и скобки.

Листинг 1.2: Вычисление арифметических выражений

```
def calculate(s: str) -> int:
    def update_stack(num, op):
        if op == "+":
            stack.append(num)
        elif op == "-":
            stack.append(-num)
        elif op == "*":
            stack.append(stack.pop() * num)
        elif op == "/":
            if num != 0:
                stack.append(int(stack.pop() / num))
            elif num == 0:
                raise ValueError("division by zero")

    balance = 0
    for char in s:
        if char == "(":
            balance += 1
        elif char == ")":
            balance -= 1
        if balance < 0:
            raise ValueError("unbalance")
    if balance != 0:
        raise ValueError("unbalance")

    stack = []
    num = 0
    op = "+"
    i = 0

    while i < len(s):
        char = s[i]
        if char.isdigit():
            num = num * 10 + int(char)
        elif char in "+-*/":
```

```
            update_stack(num, op)
            num = 0
            op = char
        elif char == "(":
            j = i + 1
            balance = 1
            while j < len(s):
                if s[j] == "(":
                    balance += 1
                elif s[j] == ")":
                    balance -= 1
                if balance == 0:
                    break
                j += 1
            num = calculate(s[i + 1 : j])
            i = j
        i += 1

    update_stack(num, op)
    return sum(stack)
```

Комментарий: Реализован рекурсивный калькулятор с использованием стека для вычисления выражений. Программа проверяет баланс скобок, обрабатывает операции с учетом приоритета и обрабатывает ошибки деления на ноль и несбалансированных скобок.

1.3 Задание 3. Замена элементов

Алгоритм заменяет каждый элемент массива на ближайший следующий элемент, который больше текущего (или на 0, если такого нет).

Листинг 1.3: Замена элементов на ближайший больший

```
def f(arr):
    result = [0] * len(arr)

    for i in range(len(arr)):
        for j in range(i + 1, len(arr)):
            if arr[j] > arr[i]:
                result[i] = arr[j]
```

```
                break

    return result


A = [1, 3, 2, 5, 3, 4]
result = f(A)
print(result)
```

Комментарий: Простая реализация алгоритма, который для каждого элемента ищет следующий больший. По умолчанию все элементы результата инициализируются нулями, и если больший элемент найден, то значение обновляется.

1.4 Задание 4. Поиск быстрейшего пути

Реализация алгоритма поиска наиболее быстрого пути между автобусными остановками.

Листинг 1.4: Поиск быстрейшего пути между остановками

```
from collections import defaultdict
import heapq


def fastest_path(routes, start, end):
    route_to_graph = defaultdict(lambda: defaultdict(list))

    for route_idx, route in enumerate(routes):
        for i in range(len(route) - 1):
            route_to_graph[route_idx][route[i]].append(route[i + 1])
            route_to_graph[route_idx][route[i + 1]].append(route[i])

    stop_to_routes = defaultdict(list)
    for route_idx, route in enumerate(routes):
        for stop in route:
            if route_idx not in stop_to_routes[stop]:
                stop_to_routes[stop].append(route_idx)
```

```
    # (время, остановка, текущий_маршрут)
    priority_queue = [(0, start, -1)]
    visited = set()

    while priority_queue:
        time, stop, current_route = heapq.heappop(priority_queue
)
        if (stop, current_route) in visited:
            continue

        visited.add((stop, current_route))

        if stop == end:
            return time
        # вниз
        for route in stop_to_routes[stop]:
            if route != current_route and (stop, route) not in
visited:
                route_switch_time = 3 if current_route != -1
else 0
                heapq.heappush(priority_queue, (time +
route_switch_time, stop, route))
        # вправо
        if current_route != -1:
            for next_stop in route_to_graph[current_route][stop
]:
                if (next_stop, current_route) not in visited:
                    heapq.heappush(priority_queue, (time + 1,
next_stop, current_route))

    return -1
```

Комментарий: Программа использует алгоритм Дейкстры с приоритетной очередью для поиска кратчайшего пути. Реализована модель, где меняться между маршрутами занимает в 3 раза больше времени, чем перемещение между остановками.

1.5 Задание 5. Задача о мышах

Алгоритм определяет порядок расположения мышей.

Листинг 1.5: Поиск расположения мышей

```
import random


def find_mice_arrangement(n, m, k, l, s):

    for _ in range(10000):
        mice = [0] * n + [1] * m
        random.shuffle(mice)
        mice_copy = mice.copy()

        start_position = 0
        while start_position < len(mice_copy) and mice_copy[
    start_position] != 0:
            start_position = (start_position + 1) % len(
    mice_copy)

        if start_position >= len(mice_copy):
            continue

        position = start_position

        while len(mice_copy) > (k + l):
            for _ in range(s - 1):
                position = (position + 1) % len(mice_copy)

            mice_copy.pop(position)

            if position == len(mice_copy):
                position = 0

        if mice_copy.count(0) == k and mice_copy.count(1) == l:
            return " ".join(["G" if mouse == 0 else "W" for
    mouse in mice])

    return "Не удалось найти решение"
```

Комментарий: Программа использует метод Монте-Карло для поиска решения, проводя множественные случайные эксперименты с различными начальными расположениями мышей и проверяя, соответствует ли результат требуемым условиям.

1.6 Задание 7. Максимальное пересечение множеств

Нахождение максимального размера пересечения пары множеств.

Листинг 1.6: Поиск максимального пересечения множеств

```
def peresech(set1, set2):
    if not set1 or not set2:
        return set()
    temp = {}
    for x in set1:
        temp[x] = True
    res = []
    for x in set2:
        if x in temp:
            res.append(x)
    return set(res)


def max_peresech_size(sets):
    mx = 0
    for i in range(len(sets)):
        for j in range(i + 1, len(sets)):
            temp = peresech(sets[i], sets[j])
            if len(temp) > mx:
                mx = len(temp)
    return mx
```

Комментарий: Алгоритм реализует ручное нахождение пересечения множеств и проверяет все возможные пары множеств, чтобы найти пару с максимальным размером пересечения.

1.7 Задание 8. Максимальный вес подстроки

Нахождение наибольшего веса среди подстрок разной длины.

Листинг 1.7: Нахождение максимального веса подстроки

```
def max_weight(s):
    n = len(s)
    max_weight_value = 0

    for L in range(1, n + 1):
        freq = {}
        for i in range(n - L + 1):
            substring = s[i : i + L]
            freq[substring] = freq.get(substring, 0) + 1
        if freq:
            max_freq = max(freq.values())
        else:
            max_freq = 0
        weight = max_freq * L
        max_weight_value = max(max_weight_value, weight)

    return max_weight_value
```

Комментарий: Программа перебирает все возможные длины подстрок, подсчитывает частоту каждой подстроки данной длины и вычисляет вес как произведение максимальной частоты на длину подстроки.

1.8 Задание 9. Максимальная сумма подматрицы

Алгоритм нахождения максимальной суммы подматрицы в заданной матрице.

Листинг 1.8: Поиск максимальной суммы подматрицы (алгоритм Кадане)

```
def max_sum_submatrix(matrix):
    if not matrix:
        return 0

    rows, cols = len(matrix), len(matrix[0])
```

```
    max_sum = -float("inf")

    for left in range(cols):
        temp = [0] * rows
        for right in range(left, cols):
            for i in range(rows):
                temp[i] += matrix[i][right]

            current_max = temp[0]
            global_max = temp[0]

            for num in temp[1:]:
                current_max = max(num, current_max + num)
                global_max = max(global_max, current_max)

            max_sum = max(max_sum, global_max)

    return max_sum
```

Комментарий: Программа использует модификацию алгоритма Кадане для 2D-массива. Для каждой пары столбцов вычисляется сумма по строкам, затем к получившемуся одномерному массиву применяется стандартный алгоритм Кадане для поиска максимальной подсуммы.