Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Лабораторная работа № 3
по дисциплине «Анализ алгоритмов»

Тема Графовые модели алгоритмов

Студент Доколин Г. А.

Группа ИУ7-52Б

Преподаватели Волкова Л. Л.

Москва, 2025

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Целью данной работы является выделение и анализ участков программ, которые могут выполняться параллельно, на основе графовых моделей алгоритмов.

Для достижения поставленной цели необходимо решить следующие задачи:

1) разработать два алгоритма согласно индивидуальному варианту — рекурсивный и итеративный;

2) описать реализацию каждого алгоритма с использованием четырёх графовых моделей: графа управления, информационного графа, операционной истории и информационной истории;

3) определить участки программ, которые могут быть выполнены параллельно, либо обосновать их отсутствие.

Таким образом, работа направлена на исследование возможностей параллельного исполнения фрагментов программ посредством анализа их графовых моделей.

1 Аналитическая часть

1.1 Рекурсия

Рекурсия — функция, которая вызывает сама себя. [1] Хвостовая рекурсия — рекурсия, при которой рекурсивный вызов является последней операцией перед возвратом из функции. [2]

1.2 Граф

Пусть V — непустое множество, $V^{(2)}$ — множество всех его двухэлементных подмножеств. Пара (V, E), где E — произвольное подмножество множества $V^{(2)}$, называется **графом** (неориентированным графом). Элементы множества V называются **вершинами** графа, а элементы множества E — **рёбрами**. Итак, граф — это конечное множество V вершин и множество E рёбер, $E \subseteq V^{(2)}$ [3].

Если направление рёбер не указано, то граф называется **неориентированным**. Если направление рёбер указано, то граф называется **ориентированным**, а сами рёбра принято называть **дугами** [3].

1.3 Виды графов

1.3.1 Граф управления

Граф управления — это ориентированный граф, вершины которого соответствуют базовым блокам программы, а дуги показывают возможные переходы управления между ними. Граф управления используется для анализа структуры программы, оптимизации и построения пути выполнения алгоритма.

1.3.2 Информационный граф

Информационный граф — это граф, в котором вершины представляют данные, а рёбра отражают отношения или зависимости между ними. Такой граф описывает потоки данных в системе и помогает анализировать, как информация перемещается и преобразуется между различными компонентами программы.

1.3.3 Граф операционной истории

Граф операционной истории — это граф, отображающий последовательность выполнения операций программы. Его вершины соответствуют операциям или командам, а рёбра отражают причинно-следственные связи между ними. Такой граф позволяет анализировать порядок выполнения действий и выявлять зависимости между операциями.

1.3.4 Граф информационной истории

Граф информационной истории — это граф, описывающий эволюцию данных в процессе выполнения программы. Его вершины представляют состояния данных после каждой операции, а рёбра отражают, как одно состояние информации переходит в другое. Граф информационной истории используется для отслеживания изменений данных и анализа корректности преобразований.

Вывод

В аналитической части были рассмотрены основные понятия, необходимые для анализа алгоритмов. Были даны определения рекурсии и хвостовой рекурсии, а также определён неориентированный и ориентированный граф с пояснением структуры вершин и рёбер.

Кроме того, рассмотрены виды графов, применяемые при анализе программ: граф управления для построения потоков выполнения, информационный граф для анализа передачи данных, граф операционной истории для отслеживания последовательности выполнения операций и граф информационной истории для анализа эволюции данных в процессе работы программы.

Данное теоретическое основание позволяет проводить систематический анализ алгоритмов и оценивать структуру и поведение программ.

2 Конструкторская часть

В данной главе представлены требования к разрабатываемому программному обеспечению, а также приведены схемы алгоритмов.

2.1 Требования к программе

Для разрабатываемой программы определены следующие задачи.

1) Реализовать интерфейс выбора операций для пользователя.

2) Обеспечить возможность работы программы в двух режимах: одиночное выполнение и массовое измерение времени выполнения.

3) В рамках режима одиночного запуска необходимо предусмотреть:

   — ввод последовательности, оканчивающейся нулём;

   — проверку корректности введённых данных.

4) Для режима массового измерения времени требуется фиксировать затраченное процессорное время и выводить результаты в табличном виде.

2.2 Разработка алгоритмов

На рисунке 2.1 представлена схема рекурсивного алгоритма вывода элементов последовательности с нечётными номерами. Сам алгоритм называется `recursive_print`.

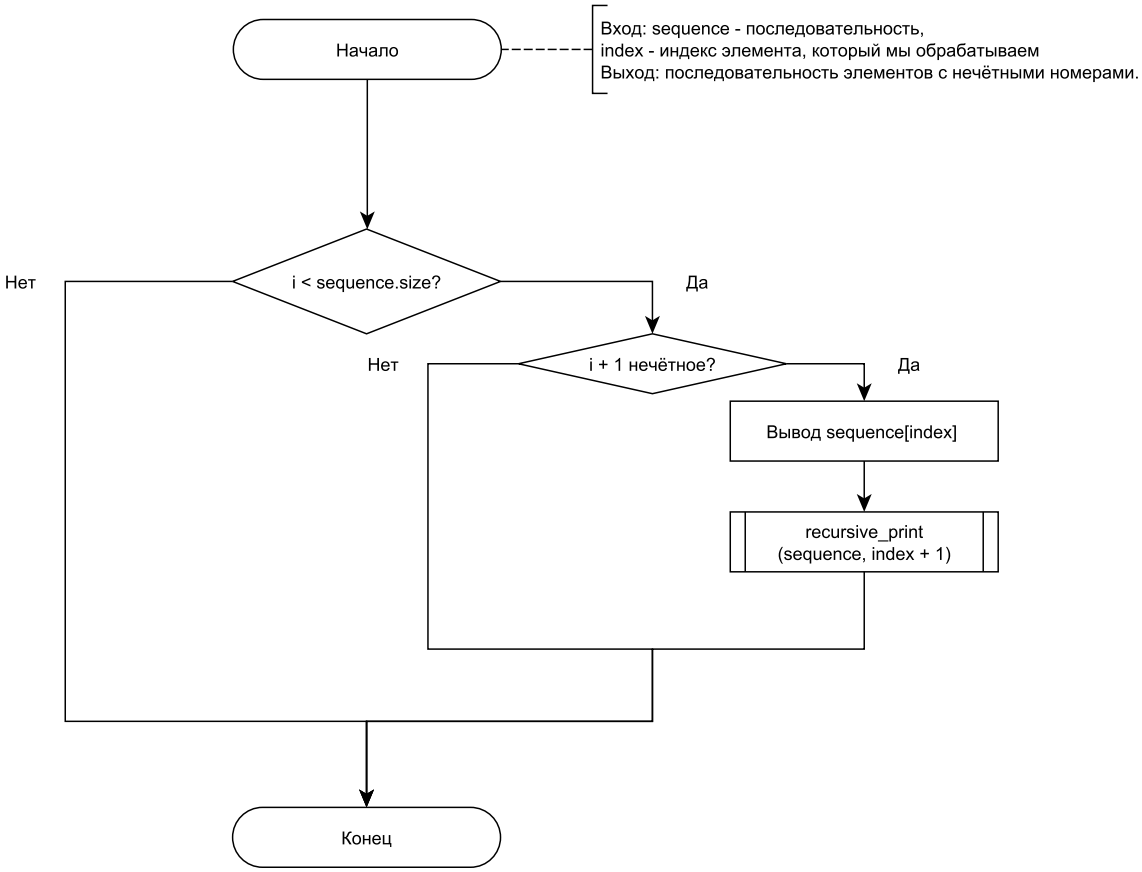

Рисунок 2.1 — Схема рекурсивного алгоритма вывода элементов последовательности с нечётными номерами

На рисунке 2.2 представлена схема итеративного алгоритма вывода элементов последовательности с нечётными номерами.

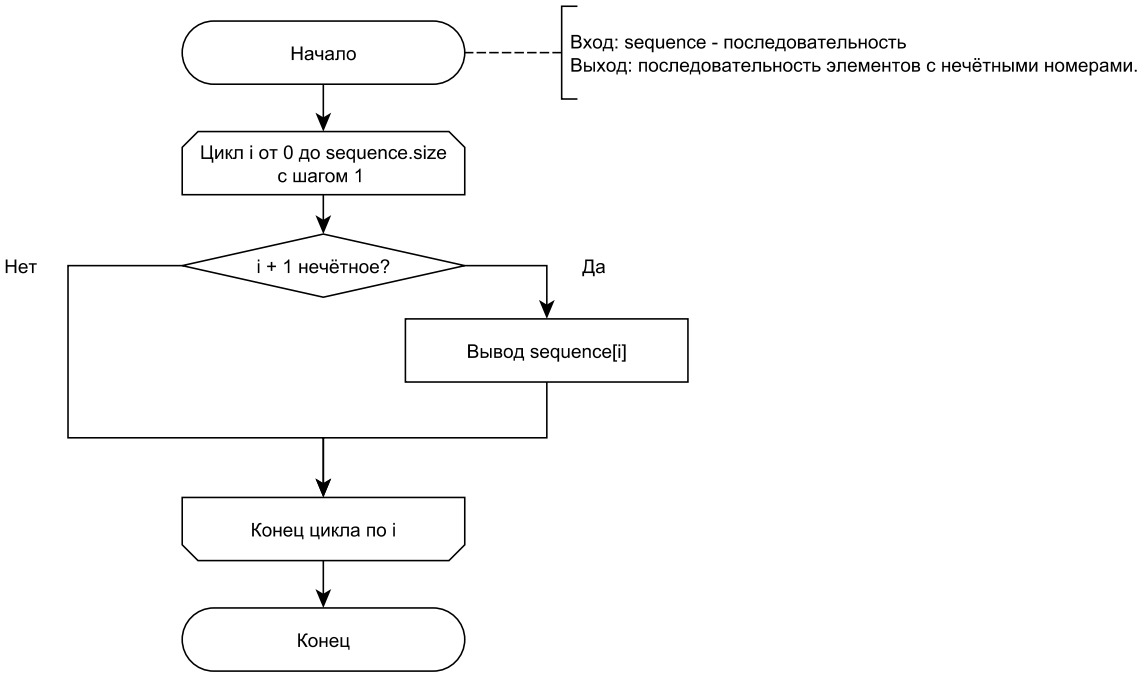

Рисунок 2.2 — Схема итеративного алгоритма вывода элементов последовательности с нечётными номерами

Вывод

В данной главе были сформулированы требования к разрабатываемому программному обеспечению, приведены схемы алгоритмов вывода элементов последовательности с нечётными номерами.

3 Технологическая часть

В данной части приведён выбор инструментов для разработки, представлены листинги реализованных алгоритмов, а также результаты функционального тестирования.

3.1 Средства реализации

Для разработки алгоритмов и программного обеспечения использовался язык программирования *C++* [4]. Этот язык обладает статической типизацией, что соответствует требованиям, предъявляемым к лабораторным работам по курсу анализа алгоритмов.

3.2 Реализации алгоритмов

В листинге 3.1 представлена реализация рекурсивного алгоритма вывода элементов последовательности с нечётными номерами.

Листинг 3.1 — Реализация рекурсивного алгоритма вывода элементов последовательности с нечётными номерами

```
void printOddPositionsRecursive(const vector<int> &sequence, int index)
    {
  if (index >= (int)sequence.size()) {
    return;
  }
  if ((index + 1) % 2 != 0) {
    cout << sequence[index] << " ";
  }
  printOddPositionsRecursive(sequence, index + 1);
}
```

Рекурсивная функция `printOddPositionsRecursive` выполняет обход элементов входной последовательности. На каждом шаге функция:

1) проверяет, не достигнут ли конец вектора (`index >= sequence.size()`);

2) если текущий индекс соответствует нечётной позиции, выводит значение элемента на экран;

3) вызывает саму себя для следующего индекса (`index + 1`).

Такой подход обеспечивает линейный проход по всем элементам последовательности. Главным отличием данного метода является использование стека вызовов: для каждого элемента создаётся новый кадр стека, что увеличивает расход памяти до $O(n)$.

В листинге 3.2 представлена реализация итеративного алгоритма вывода элементов последовательности с нечётными номерами.

Листинг 3.2 — Реализация итеративного алгоритма вывода элементов последовательности с нечётными номерами

```
void printOddPositionsIterative(const vector<int> &sequence) {
  for (size_t i = 0; i < sequence.size(); i++) {
    if ((i + 1) % 2 != 0) {
      cout << sequence[i] << " ";
    }
  }
  cout << endl;
}
```

В итеративной реализации используется цикл `for`, проходящий по всем элементам последовательности. Для каждой позиции вычисляется выражение `(i + 1) % 2 != 0`, определяющее нечётность индекса. Если условие выполняется, элемент выводится на экран.

В отличие от рекурсивного алгоритма, итеративный вариант не создаёт дополнительных кадров стека и использует фиксированное количество переменных, что обеспечивает константную сложность по памяти $O(1)$ и более высокую производительность при больших размерах входных данных.

3.3 Функциональное тестирование

Таблица 3.1 — Функциональные тесты

№	Описание теста	Входные данные	Ожидаемый результат
1	Пустая последовательность	$\{\}$	Пустой вывод
2	Последовательность из одного элемента	$\{5\}$	5
3	Последовательность из двух элементов	$\{1, 2\}$	1
4	Последовательность из трёх элементов	$\{10, 20, 30\}$	10 30
5	Последовательность из четырёх элементов	$\{1, 2, 3, 4\}$	1 3
6	Последовательность из пяти элементов	$\{5, 10, 15, 20, 25\}$	5 15 25
7	Последовательность с отрицательными числами	$\{-1, -2, -3, -4, -5\}$	-1 -3 -5
8	Последовательность с нулевыми значениями	$\{0, 0, 0, 0, 0, 0\}$	0 0 0
9	Смешанная последовательность (положительные и отрицательные)	$\{-10, 5, -3, 8, 0, -1\}$	-10 -3 0
10	Последовательность с повторяющимися значениями	$\{7, 7, 7, 7, 7, 7\}$	7 7 7
11	Последовательность с большими числами	$\{1000, 2000, 3000, 4000\}$	1000 3000
12	Длинная последовательность (15 элементов)	$\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15\}$	1 3 5 7 9 11 13 15

Все приведённые тесты успешно пройдены, что подтверждает правильность работы алгоритмов в различных ситуациях. Оба алгоритма (рекурсивный и итеративный) демонстрируют идентичные результаты для всех тестовых случаев.

Вывод

В технологической части были рассмотрены средства реализации и проведено функциональное тестирование алгоритмов вывода элементов последовательности с нечётными номерами; реализованы и проверены две версии алгоритмов — рекурсивная и итеративная. Для подтверждения корректности работы использовался расширенный набор функциональных тестов, включающий пустые последовательности, последовательности из одного элемента, с отрицательными числами, с нулевыми значениями и последовательности различной длины, результаты

которых подтвердили правильность работы обеих реализаций.

Оба алгоритма показали идентичные результаты для всех тестовых случаев, что подтверждает их корректность и согласованность работы. Рекурсивный алгоритм демонстрирует более простую и понятную структуру кода, однако требует больше ресурсов памяти. Итеративный алгоритм более эффективен по использованию ресурсов, но может быть менее интуитивно понятен для некоторых разработчиков.

4 Исследовательская часть

В данной главе проведён анализ работы итеративного и рекурсивного алгоритмов вывода элементов последовательности с нечётными номерами с помощью графовых моделей.

4.1 Графовые модели итеративного алгоритма

Для анализа введены основные операторы итеративного алгоритма:

1) проверка конца последовательности: `i < sequence.size();`
2) проверка нечётности индекса: `if ((i + 1) % 2 != 0);`
3) вывод элемента на экран: `std::cout « sequence[i];;`
4) инкремент индекса: `i++`.

4.1.1 Графы итеративного алгоритма

На рисунке 4.1 показан граф управления итеративного алгоритма, где вершины соответствуют операторам, а дуги — последовательности их выполнения.

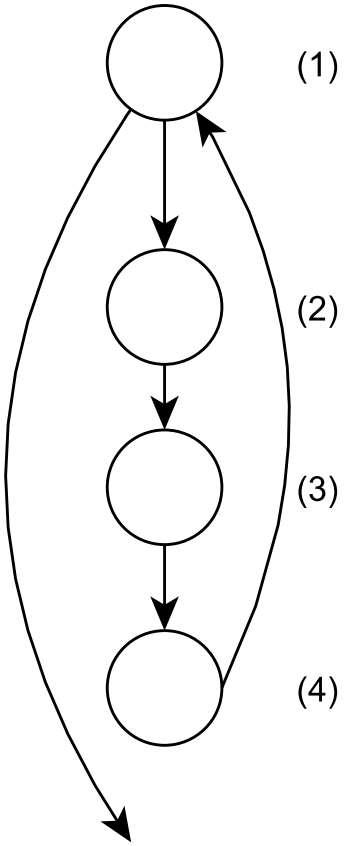

Рисунок 4.1 — Граф управления для итеративного алгоритма

На рисунке 4.2 представлен информационный граф, отображающий потоки передачи

данных между операторами.

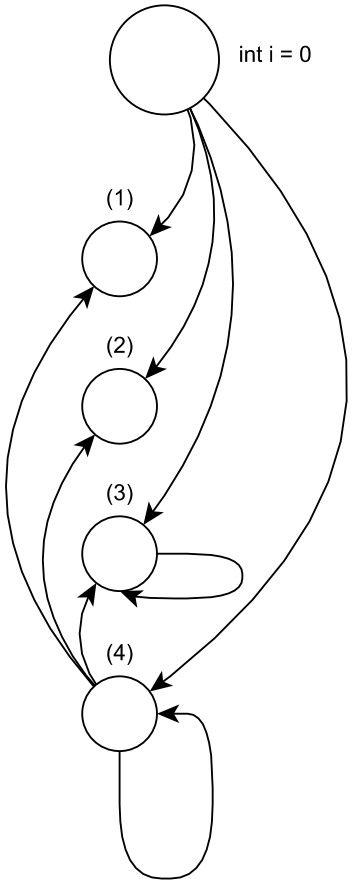

Рисунок 4.2 — Информационный граф для итеративного алгоритма

На рисунке 4.3 показан граф операционной истории, где вершины — срабатывания операторов, а дуги — последовательность их выполнения.

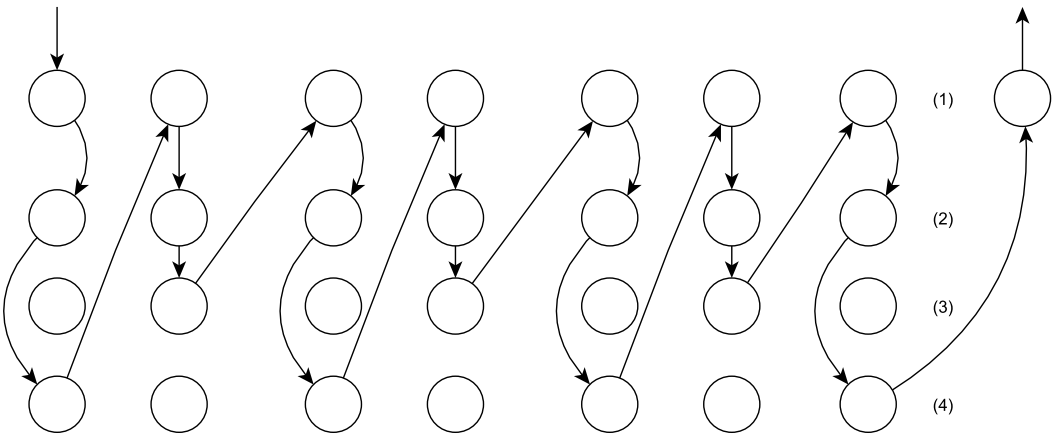

Рисунок 4.3 — Граф операционной истории для итеративного алгоритма

На рисунке 4.4 приведён граф информационной истории, демонстрирующий информа-

ционные зависимости между срабатываниями операторов.

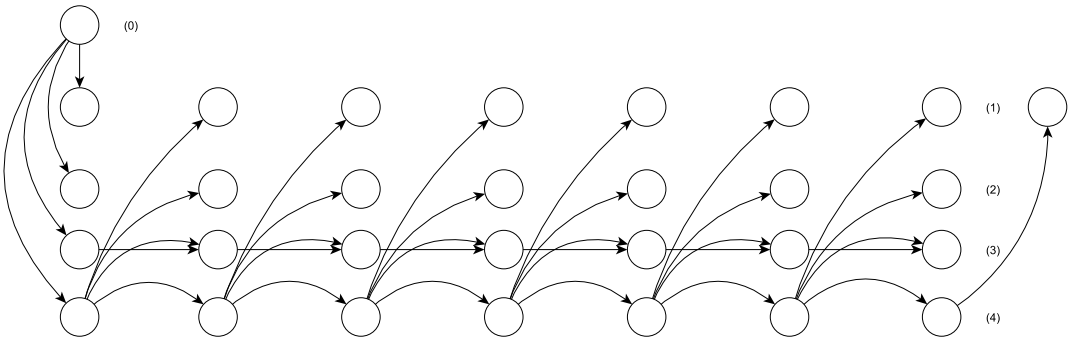

Рисунок 4.4 — Граф информационной истории для итеративного алгоритма

4.2 Графовые модели рекурсивного алгоритма

Основные операторы рекурсивного алгоритма:

1) проверка конца последовательности: `if (index >= sequence.size());`

2) проверка нечётности позиции: `if ((index + 1) % 2 != 0);`

3) вывод текущего элемента: `std::cout « sequence[index] « ;`

4) рекурсивный вызов функции: `printOddPositionsRecursive(sequence, index + 1)`.

4.2.1 Графы рекурсивного алгоритма

На рисунке 4.5 представлен граф управления рекурсивного алгоритма, демонстрирующий порядок выполнения операторов.

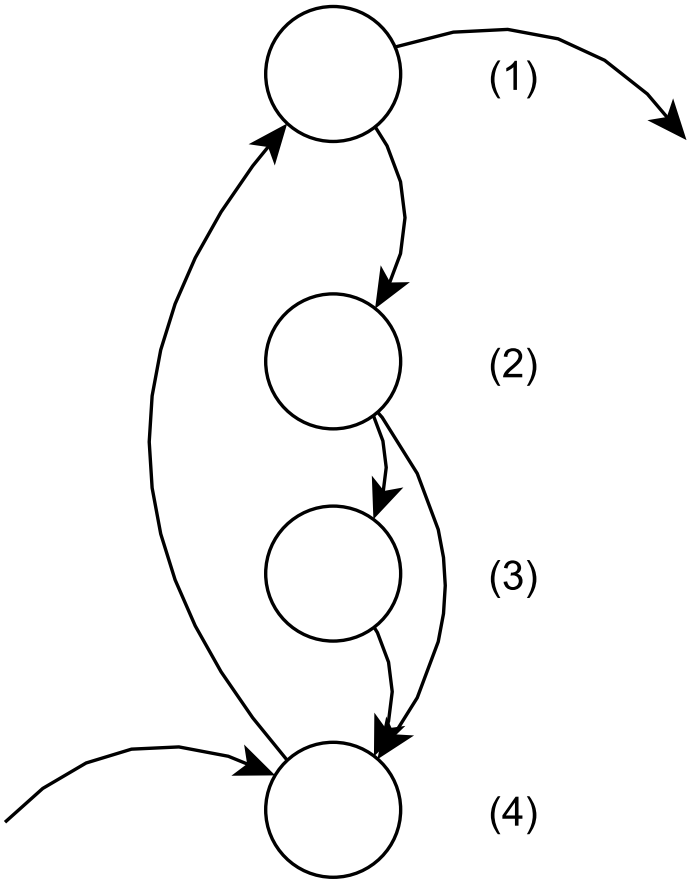

Рисунок 4.5 — Граф управления для рекурсивного алгоритма

На рисунке 4.6 приведён информационный граф, отображающий потоки данных между операторами.

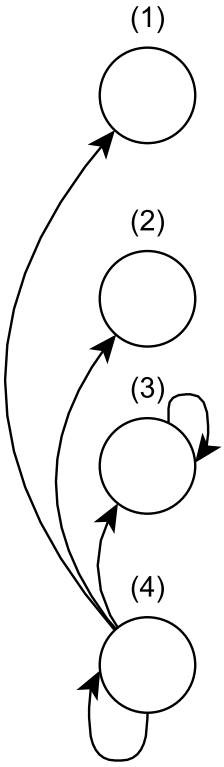

Рисунок 4.6 — Информационный граф для рекурсивного алгоритма

На рисунке 4.7 показан граф операционной истории, где вершины — срабатывания операторов, а дуги — последовательность их выполнения.

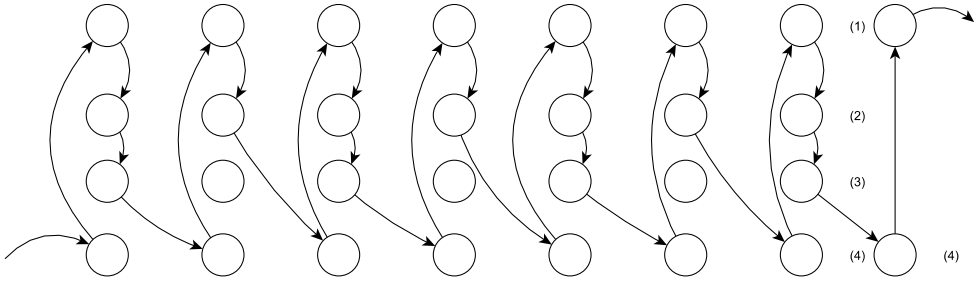

Рисунок 4.7 — Граф операционной истории для рекурсивного алгоритма

На рисунке 4.8 изображён граф информационной истории, демонстрирующий информационные зависимости между срабатываниями операторов.

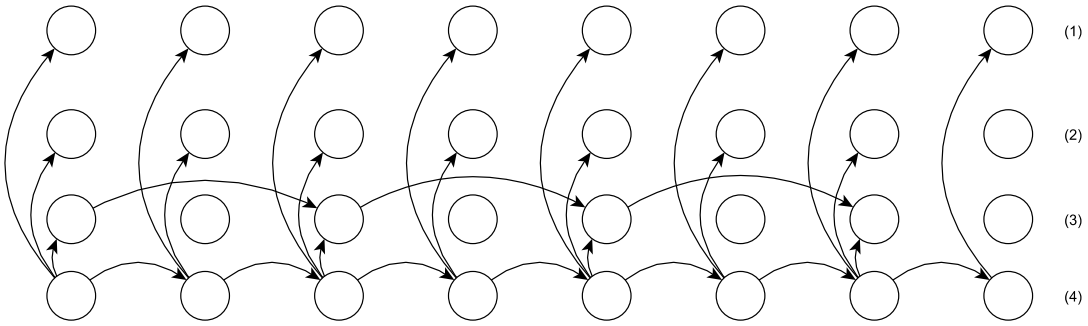

Рисунок 4.8 — Граф информационной истории для рекурсивного алгоритма

Вывод

Анализ графовых моделей показал, что оба алгоритма выполняются последовательно: каждая операция зависит от предыдущей, что исключает возможность распараллеливания. Различие состоит только в организации вычислений: рекурсивный алгоритм использует стек вызовов, итеративный — цикл.

ЗАКЛЮЧЕНИЕ

В ходе выполнения работы были рассмотрены графовые модели рекурсивного и итеративного алгоритмов вывода элементов последовательности с нечётными номерами, а также проведён их детальный анализ. Поставленная цель — на основе построенных графовых моделей выявить участки программного кода, допускающие параллельное исполнение, — была полностью достигнута. В результате анализа установлено, что такие участки отсутствуют.

В результате работы:

— разработаны и реализованы два варианта алгоритма — рекурсивный и итеративный — в соответствии с заданием;

— для каждого алгоритма построены четыре типа графовых моделей: граф управления, информационный граф, граф операционной истории и граф информационной истории;

— проведён структурный анализ всех моделей с целью выявления возможностей параллелизма;

— установлено, что оба алгоритма обладают строго последовательной структурой выполнения: каждая операция зависит от результата предыдущей, а рекурсивный вызов формирует линейную цепочку активаций без ветвлений или независимых веток;

— подтверждено, что ни одна из рассмотренных реализаций не содержит фрагментов, пригодных для распараллеливания.

Таким образом, несмотря на различия в организации кода (использование цикла в итеративном варианте и рекурсивных вызовов в рекурсивном), оба подхода принципиально последовательны и не допускают одновременного выполнения операций. Это накладывает естественное ограничение на применение многопоточности или векторизации при решении подобных задач. Полученные выводы согласуются с теоретическими свойствами линейных алгоритмов обработки последовательностей и подтверждают корректность построенных графовых моделей.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Tproger. Что такое рекурсия и как с ней работать [Электронный ресурс]. Режим доступа: https://tproger.ru/articles/chto-takoe-rekursiya-i-kak-s-nej-rabotat (Дата обращения: 26.10.2025).

2. ScalaBook. Хвостовая рекурсия в функциональном программировании [Электронный ресурс]. Режим доступа: https://scalabook.ru/fp/fp/tail_recursion.html (Дата обращения: 26.10.2025).

3. Старикова О. Л. Ориентированные и неориентированные графы в примерах и задачах: методические указания / Самарский государственный аэрокосмический университет имени академика С. П. Королёва (национальный исследовательский университет). — Самара, 2015. — 4 с.

4. Metanit.com. Руководство по C++ [Электронный ресурс]. Режим доступа: https://metanit.com/cpp/tutorial/ (Дата обращения: 12.10.2025).