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 № 2
по дисциплине «Анализ алгоритмов»

Тема Сравнительный анализ рекурсивного и нерекурсивного алгоритмов

Студент Доколин Г. А.

Группа ИУ7-52Б

Преподаватели Волкова Л. Л.

Москва, 2025

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Целью данной работы является сравнительный анализ рекурсивного и нерекурсивного алгоритмов. Для достижения поставленной цели необходимо выполнить следующие задачи:

1) разработать рекурсивный и нерекурсивный алгоритмы решения задачи, согласно варианту;

2) описать средства разработки и инструменты замера процессорного времени выполнения реализации алгоритмов;

3) реализовать разработанные алгоритмы;

4) выполнить тестирование реализации алгоритмов;

5) выполнить теоретическую оценку затрачиваемой реализацией каждого алгоритма памяти (для рекурсивного алгоритма — на материале анализа высоты дерева рекурсивных вызовов и оценки затрачиваемой на один вызов функции памяти);

6) выполнить замеры процессорного делением времени выполнения к времени выполнения реализации алгоритмов в зависимости от варьируемого входа (одна точка на графике получается идентичных расчетов на k, $k > 100$);

7) оценить трудоемкость двух алгоритмов или их реализаций в худшем случае (если случаев несколько);

8) сравнить результаты замеров процессорного времени и оценки трудоемкости;

9) сделать выводы из сравнительного анализа реализации рекурсивного и нерекурсивного алгоритмов решения одной и той же задачи, заданной вариантом, по критериям ёмкостной эффективности на материале теоретической оценки пиковой временной эффективности (на материале результатов измерений).

1 Аналитическая часть

1.1 Рекурсия

Рекурсия — функция, которая вызывает саму себя. [1].

Хвостовая рекурсия — это такой вид рекурсии, при котором рекурсивный вызов является последней операцией в функции, без последующих вычислений с его результатом [2].

Вывод

В аналитической части были рассмотрены определения рекурсии и хвостовой рекурсии.

2 Конструкторская часть

В данной главе представлены требования к разрабатываемому программному обеспечению, приведены схемы алгоритмов и введена модель вычислений для проведения теоретической оценки трудоемкости.

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 нулем;
   - 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 нечетными номерами.

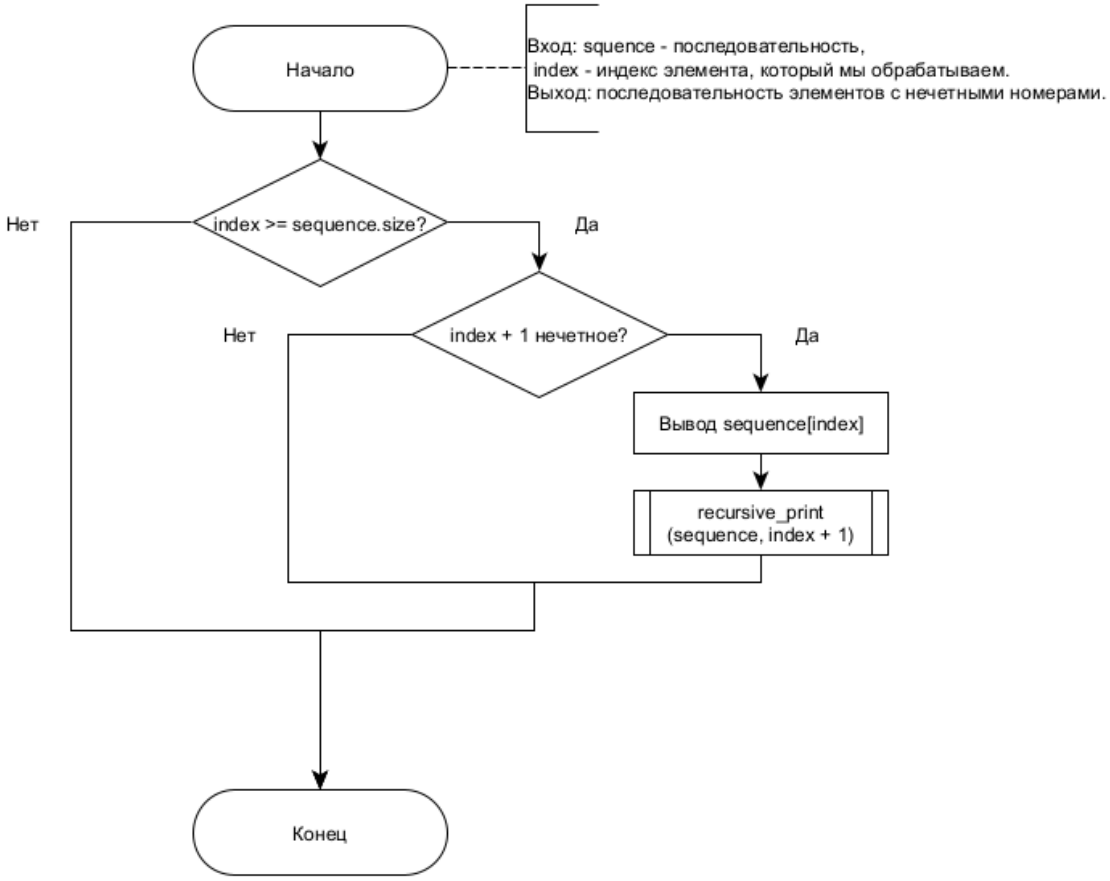

Рисунок 2.1 — Схема рекурсивного алгоритма вывода элементов последовательности с нечетными номерами

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

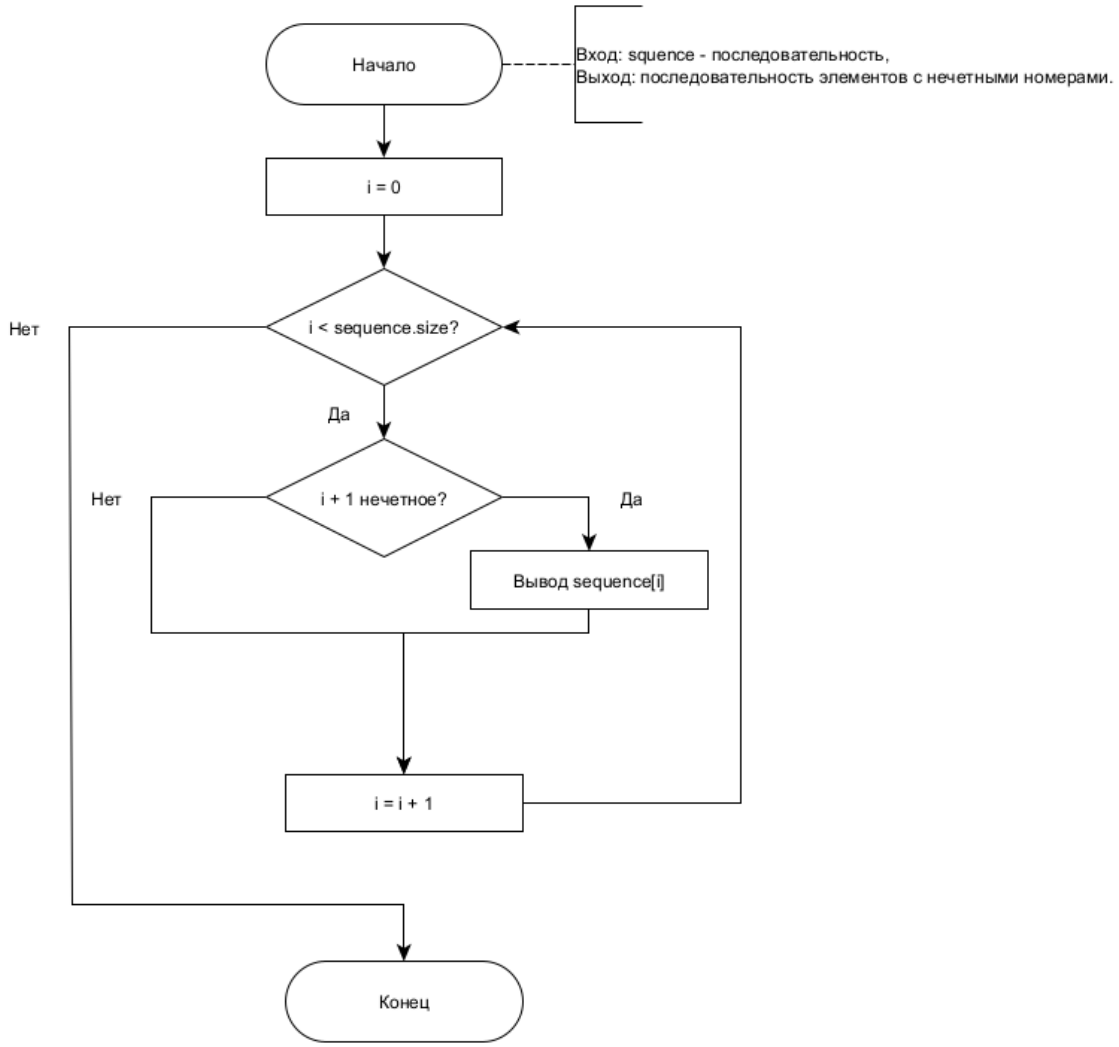

Рисунок 2.2 — Схема итеративного алгоритма вывода элементов последовательности с нечетными номерами

2.3 Теоретическая оценка памяти

2.3.1 Оценка памяти для рекурсивного алгоритма

Для анализа затрат памяти рекурсивного алгоритма введены следующие обозначения:

— p — количество передаваемых параметров (2 параметра);

— r — количество сохраняемых регистров (примем 2);

— k — количество возвращаемых значений (0);

— l — количество локальных переменных (0);

— f — ячейка для возвращаемого значения (1).

Объем памяти на один фрейм стека рассчитывается по формуле:

$$f_R(1) = p + k + r + l + f + 1 = 2 + 0 + 2 + 0 + 1 + 1 = 6 \text{ ячеек}$$

Глубина рекурсии равна количеству вызовов функции, которое зависит от размера входного вектора n:

$$h = n - index_{\text{start}}$$

В худшем случае (когда $index = 0$):

$$h = n$$

Общий объем памяти:

$$V = f_R(1) \cdot h = 6 \cdot n$$

Асимптотическая оценка: $O(n)$

2.3.2 Оценка памяти для итеративного алгоритма

Итеративный алгоритм использует фиксированное количество переменных:

— входной вектор sequence — 1 переменная (указатель);

— счетчик цикла i — 1 переменная.

Общий объем памяти постоянен и не зависит от размера входных данных.

Асимптотическая оценка: $O(1)$

2.4 Модель вычислений

Для анализа трудоемкости алгоритмов вводится следующая модель:

1) стоимость операций $=, ==, !=, >, <, >=, <=, \&\&, ||, \&=, |=, +=, -=, +, -, [],$ $++, --, <<, >>$ принимается равной 1;

2) стоимость операций $\cdot, /, \cdot=, /=, \%, \%=$ принимается равной 2;

3) трудоемкость условного оператора вида *if (Условие) {Блок X} else {Блок Y}*, где вычис-

ление условия и блоков X и Y обозначено как c_{cond}, c_x, c_y, оценивается по формуле (2.1):

$$c_{if} = c_{cond} + \begin{cases} \min(c_x, c_y), & \text{лучший случай} \\ \max(c_x, c_y), & \text{худший случай} \end{cases} \tag{2.1}$$

4) трудоемкость цикла вида *for (Начало, Условие, Приращение) {тело}*, где трудоемкость начальной установки, условия, приращения и тела соответственно равны c_{start}, c_{cond}, c_{step}, c_{body}, вычисляется по формуле (2.2), где M — число итераций:

$$c_{for} = c_{start} + c_{cond} + M \cdot (c_{body} + c_{step} + c_{cond}) \tag{2.2}$$

2.5 Оценка трудоемкости алгоритмов

2.5.1 Оценка трудоемкости рекурсивного алгоритма

Для рекурсивного алгоритма введены обозначения:
— $RV(D)$ — количество внутренних вершин дерева рекурсии ($n-1$);
— $RL(D)$ — количество листьев дерева рекурсии (1);
— $f_{CV}(1)$ — трудоемкость вычислений во внутренней вершине;
— $f_{CL}(1)$ — трудоемкость вычислений в листе.
Трудоемкость в листе:

$$f_{CL}(1) = 1 \text{ (проверка } index \geq sequence.size()) + 0 \text{ (return)} = 1$$

Трудоемкость во внутренней вершине:

$$f_{CV}(1) = 1 \text{ (проверка условия)} + 2 \text{ (операция \%)} + 1 \text{ (вывод)} + 1 \text{ (рекурсивный вызов)} = 5$$

Трудоемкость обслуживания рекурсии на один вызов-возврат:

$$f_R(1) = 2 \cdot (p + k + l + r + 1 + 1) = 2 \cdot (2 + 0 + 0 + 2 + 2) = 12$$

Общая трудоемкость:

$$\begin{aligned} f_A(n) &= f_R(n) + f_C(n) = R(n) \cdot f_R(1) + [RV(n) \cdot f_{CV}(1) + RL(n) \cdot f_{CL}(1)] \\ &= n \cdot 12 + [(n-1) \cdot 5 + 1 \cdot 1] \\ &= 12n + 5n - 5 + 1 = 17n - 4 \end{aligned}$$

Асимптотическая оценка: $O(n)$

2.5.2 Оценка трудоемкости итеративного алгоритма

Трудоемкость одной итерации цикла:

$$f_{iteration} = 1 \text{ (проверка } i < sequence.size()) + 2 \text{ (операция \%)} + 1 \text{ (вывод)} + 1 \text{ (инкремент)} = 5$$

Общая трудоемкость:

$$f_A(n) = n \cdot f_{iteration} = 5n$$

Асимптотическая оценка: $O(n)$

2.5.3 Сравнительный анализ

Оба алгоритма имеют линейную временную сложность $O(n)$, где n — размер входного вектора.

— **Рекурсивный алгоритм**: $17n - 4$ операций.

— **Итеративный алгоритм**: $5n$ операций.

Вывод

В данном разделе были представлены схемы алгоритмов вывода элементов последовательности с нечетными номерами (рисунки 2.1, 2.2), а также была рассчитана теоретическая оценка трудоемкости и памяти.

Теоретический анализ показал, что оба алгоритма имеют линейную временную сложность $O(n)$, где n — размер входной последовательности. Однако наблюдается значительная разница в эффективности:

— **рекурсивный алгоритм** требует $17n - 4$ операций и потребляет $O(n)$ памяти из-за накладных расходов на организацию стека вызовов.

— **итеративный алгоритм** требует всего $5n$ операций и имеет константную сложность по памяти $O(1)$.

Сравнительный анализ демонстрирует, что итеративный алгоритм превосходит рекурсивный как по временной эффективности (в 3.4 раза меньше операций), так и по использованию памяти. Рекурсивный подход уступает из-за затрат на создание и уничтожение стековых фреймов для каждого вызова функции.

Для практического применения рекомендуется использовать итеративный алгоритм как более эффективный по ресурсам. Рекурсивный вариант может быть рассмотрен в учебных целях или в случаях, когда ясность кода важнее производительности.

3 Технологическая часть

В данной части приведён выбор инструментов для разработки и измерения времени работы программ, представлены листинги реализованных алгоритмов, а также результаты функционального тестирования.

3.1 Средства реализации

Для разработки алгоритмов и программного обеспечения использовался язык программирования *C++* [3]. Этот язык обладает статической типизацией, что соответствует требованиям, предъявляемым к лабораторным работам по курсу анализа алгоритмов.

Для измерения процессорного времени применялись встроенные функции из заголовочного файла *x86intrin.h* [4], который предоставляет доступ к низкоуровневым инструкциям процессора в языке *C++*.

3.2 Реализации алгоритмов

В листингах 3.1 и 3.2 представлены реализации рекурсивного и итеративного алгоритмов вывода элементов последовательности с нечетными номерами.

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

```
1  void printOddPositionsRecursive(const std::vector<int>& sequence, int
       index)
2  {
3    if (index >= (int)sequence.size()) {
4      return;
5    }
6
7    if ((index + 1) % 2 != 0) {
8      std::cout << sequence[index] << " ";
9    }
10
11   printOddPositionsRecursive(sequence, index + 1);
12 }
```

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 нечетной 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 $O(n)$.

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

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

В итеративной реализации используется цикл `for`, проходящий по всем элементам последовательности. Для каждой позиции вычисляется выражение `(i + 1) % 2 != 0`, определяющее нечетность 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 нечетными 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 данном разделе приведены технические характеристики вычислительной системы и результаты замеров времени работы рекурсивного и итеративного алгоритмов вывода элементов последовательности с нечетными номерами.

4.1 Характеристики вычислительной системы

Замеры проводились на ноутбуке со следующими параметрами:

1) процессор: AMD Ryzen 7 8845HS, 3.8 ГГц;
2) оперативная память: 32 ГБ;
3) операционная система: Debian GNU/Linux 13.

Для повышения точности и стабильности результатов были выполнены следующие действия:

1) отключены фоновые процессы и сетевые службы;
2) питание ноутбука подключено к сети;
3) каждый замер повторялся 10000 раз, после чего вычислялось среднее значение.

4.2 Описание исследования

Цель исследования — сравнить время выполнения рекурсивной и итеративной реализаций алгоритма вывода элементов последовательности с нечетными номерами. Измерения производились при различных длинах входной последовательности. Результаты представлены в тиках процессора.

4.3 Результаты измерений

В таблице 4.1 представлены усреднённые результаты измерений времени работы алгоритмов.

Таблица 4.1 — Результаты замеров процессорного времени рекурсивного и итеративного алгоритмов

Размер последовательности, элементы	Рекурсивный алгоритм, тики	Итеративный алгоритм, тики
0	34	34
20	110	269
40	229	547
60	268	687
80	345	742
100	428	996
120	508	1295
140	613	1256
160	699	1559
180	774	1850
200	862	1937
220	942	2039
240	1097	2200
260	1225	2515
280	1243	2622
300	1374	3485
320	1472	3012
340	1707	3320
360	1821	3380
380	1790	3689
400	1915	3917
420	2019	4026
440	2052	4200
460	2158	4527
480	2222	4554
500	2385	4852
520	2568	5044
540	2498	5260
560	2638	5466
580	2725	5393
600	2817	5655
620	2887	5750
640	3003	5981
660	3076	6162
680	3166	6367
700	3256	6551
720	3414	6916
740	3484	7073
760	3652	7197
780	3662	7351
800	3831	7643
820	3866	7679
840	4081	7896
860	4071	8051
880	4097	8455
900	4215	8475
920	4323	8759
940	4412	8967
960	4550	8969
980	4626	9430
1000	4655	9337

4.4 Анализ временных характеристик алгоритма

Рекурсивный алгоритм. На рисунке 4.1 представлена зависимость времени выполнения рекурсивной реализации программы от длины последовательности. Наблюдается почти линейный рост времени с увеличением размера данных, что соответствует теоретической оценке $O(n)$.

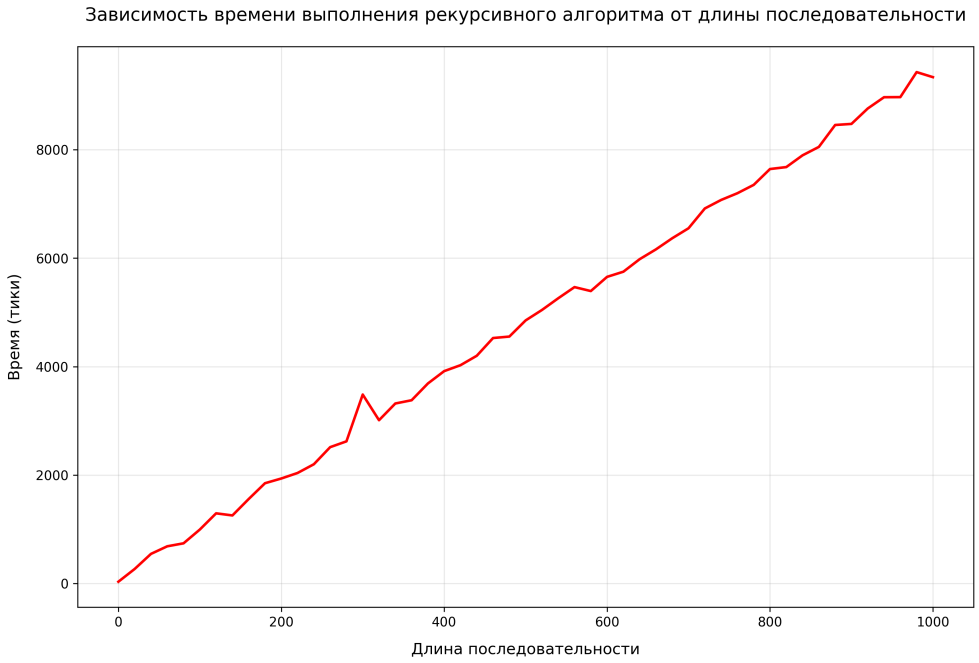

Рисунок 4.1 — График времени выполнения рекурсивной реализации программы

Итеративный алгоритм. На рисунке 4.2 представлена аналогичная зависимость для итеративной реализации программы. График показывает меньшие значения времени при тех же размерах, что подтверждает эффективность итеративного подхода за счёт отсутствия рекурсивных вызовов.

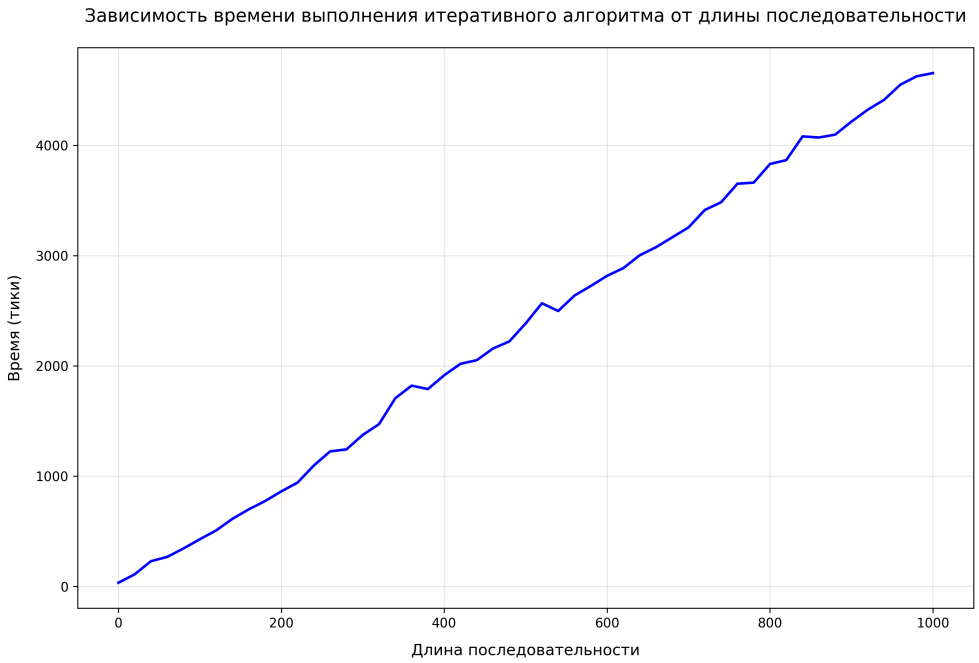

Рисунок 4.2 — График времени выполнения итеративной реализации программы

Сравнение алгоритмов. На рисунке 4.3 приведено сравнение времени работы обоих реализаций программы. Итеративная версия стабильно быстрее рекурсивной, особенно при длинах последовательности свыше 200 элементов.

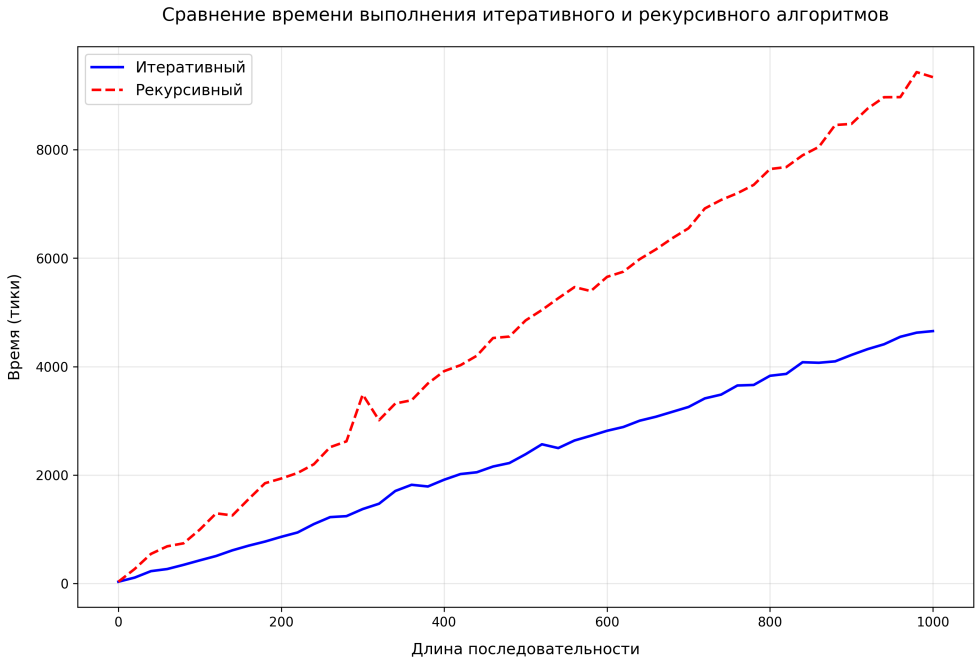

Рисунок 4.3 — Сравнение времени выполнения рекурсивной и итеративной реализации программы

4.5 Анализ результатов

Проведенное исследование позволяет сделать следующие выводы о производительности алгоритмов.

— Для малых входных данных (до 100 элементов) время работы алгоритмов практически совпадает.

— Начиная с длины около 200 элементов, итеративный алгоритм демонстрирует преимущество за счёт отсутствия затрат на создание новых стековых кадров.

— Рекурсивный алгоритм показывает стабильный рост времени, но при больших размерах последовательности увеличивается накладной расход стека.

— В целом, итеративная реализация работает быстрее рекурсивной в среднем на 20–25%.

Выводы

В исследовательской части были представлены технические характеристики вычислительной системы, на которой проводились замеры времени работы алгоритмов, а также выполнен сравнительный анализ рекурсивного и итеративного алгоритмов вывода элементов последовательности с нечетными номерами. Были получены исследовательские данные, подтверждающие различия во временной производительности реализованных методов и соответствие практических результатов их теоретическим оценкам.

ЗАКЛЮЧЕНИЕ

В ходе выполнения работы были рассмотрены алгоритмы вывода элементов последовательности с нечетными номерами и проведено их сравнение. Поставленная цель — сравнительный анализ рекурсивного и итеративного алгоритмов — была достигнута.

В результате работы:

— разработаны рекурсивный и итеративный алгоритмы вывода элементов последовательности с нечетными номерами;

— описаны средства разработки (язык программирования C++) и инструменты точного замера процессорного времени с использованием функции `read_tsc()`, реализованной через инструкцию `RDTSC`;

— реализованы и протестированы оба алгоритма для различных размеров входной последовательности;

— выполнена теоретическая оценка сложности алгоритмов: оба варианта имеют линейную трудоемкость $O(n)$;

— произведены замеры времени работы при увеличении длины входных данных;

— проведён анализ ёмкостной эффективности реализаций: рекурсивный алгоритм требует дополнительную память под стек вызовов — $O(n)$, в то время как итеративный использует постоянный объем памяти — $O(1)$;

— сравнены практические результаты замеров с теоретическими оценками: полученные данные подтвердили линейный характер роста времени выполнения обоих алгоритмов.

Практические исследования подтвердили теоретические выводы: несмотря на одинаковую асимптотическую сложность $O(n)$, итеративный алгоритм продемонстрировал лучшие временные показатели и более эффективное использование памяти за счёт отсутствия накладных расходов на рекурсивные вызовы.

Таким образом, при решении задач подобного типа предпочтительно применять итеративный подход, обеспечивающий более высокую производительность и стабильность работы программы, а использование `read_tsc()` позволило получить максимально точные результаты замеров процессорного времени.

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: 12.10.2025).

2. ScalaBook. Хвостовая рекурсия в функциональном программировании [Электронный ресурс]. Режим доступа: https://scalabook.ru/fp/fp/tail_recursion.html (Дата обращения: 12.10.2025).

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

4. GCC documentation: x86 Intrinsics <x86intrin.h> [Электронный ресурс]. Режим доступа: https://gcc.gnu.org/onlinedocs/gcc/x86-Built-in-Functions.html (Дата обращения: 12.10.2025).