Министерство образования и молодежной политики

Свердловской области

Государственное автономное профессиональное образовательное

учреждение Свердловской области «Ирбитский политехникум»

КУРСОВАЯ РАБОТА

по ПМ.01 Разработка программных модулей программного обеспечения для

компьютерных систем

МДК 01.01. Системное программирование

Тема:Рекурсия, сложная рекурсия. Применение рекурсии в алгоритмах.

Выполнил:

студент группы ПКС-306

специальность 09.02.03

Программирование

в компьютерных системах

Ипполитов Юрий Иванович

**Проверил:** преподаватель

Вишнякова Н.В.

Ирбит 2022СОДЕРЖАНИЕ

[Введение 3](#_Toc37435645)

[1.Понятие рекурсии. Примеры рекурсивных алгоритмов. 5](#_Toc37435646)

[2. Решение задач с применениемрекурсивных алгоритмов 10](#_Toc37435647)

[2.1 Запись последовательность чисел в обратном порядке 10](#_Toc37435648)

[2.2 Двоичный поиск в массиве 10](#_Toc37435649)

[2.3 Вычисление чисел Фибоначчи 11](#_Toc37435650)

[2.4 Быстрая сортировка (quicksort) 12](#_Toc37435651)

[2.5 Сортировка слиянием (mergesort) 12](#_Toc37435652)

[2.6 Фракталы: круговой фрактал 14](#_Toc37435653)

[2.7 Нахождение наибольшего общего делителя 14](#_Toc37435654)

[2.8 Вычисление факториала 15](#_Toc37435655)

[3. Анализ рекурсивных алгоритмов 17](#_Toc37435656)

[3.1 Метод подстановки (итераций) 17](#_Toc37435657)

[3.2 Метод математической индукции 17](#_Toc37435658)

[3.3 Анализ алгоритма бинарного поиска 19](#_Toc37435659)

[3.4 Анализ алгоритма поиска 19](#_Toc37435660)

[3.5 Анализ алгоритма сортировки слиянием 19](#_Toc37435661)

[3.6 Анализ трудоемкости быстрой сортировки 19](#_Toc37435662)

[3.7 Хвостовая рекурсия и цикл 20](#_Toc37435663)

[Заключение 23](#_Toc37435664)

[Список источников 24](#_Toc37435665)

Приложения

# ВВЕДЕНИЕ

Благодаря бурному развитию научных, технических и технологических исследований стало возможным хранить огромные объёмы данных и преобразовывать их со скоростями, которые ещё несколько десятилетий назад были немыслимы. При создании сложных информационных систем перед проектировщиками стали нетривиальные задачи, требующие разработки новых концепций программирования.

В настоящее время область практического применения рекурсии весьма широка. Она включает, в частности, сложные задачи численного анализа, алгоритмы трансляции, а также различные операции над списками, являющиеся необходимым аппаратом разработки современных автоматизированных систем управления. Поэтому аппарат рекурсии предусматривается практически во всех актуальных языках программирования.

Полезность, важность и необходимость рекурсии, как одного из концептуальных методов решения практических задач подчеркивалась многими мэтрами информатики. Одни из них, это два лауреата премии Тьюринга: американский специалист по системному программированию Д.Кнут и английский теоретик информатики Ч.Хоар.

Значимость рекурсии определяется тем, что она является одним из наиболее мощных, а также самым общим методом научного познания. Она эффективно применяется во многих прикладных и теоретических естественнонаучных дисциплинах, и стала неотъемлемой их частью.

Цель курсовой работы - приобретение практических навыков при изучение методов и подходов применения рекурсивных алгоритмов.

Задачи:

1) подобрать и систематизировать литературу по теме;

2) изучить применения рекурсии в математике;

3) освоить рекурсивные алгоритмы в программировании;

4) разработать программы с использованием рекурсивных алгоритмов;

5) оформить документацию.

Объект исследования

Предмет исследования

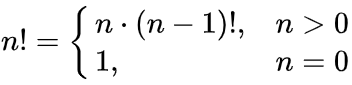
# ПОНЯТИЕ РЕКУРСИИ. ПРИМЕРЫ РЕКУРСИВНЫХ

# АЛГОРИТМОВ.

*Рекурсия*— определение, описание, изображение какого-либо объекта или процесса внутри самого этого объекта или процесса, то есть ситуация, когда объект является частью самого себя. Термин «рекурсия» используется в различных специальных областях знаний — от лингвистики до логики, но наиболее широкое применение находит в математике и информатике.

В математике

Рекурсия имеет отношение к методу определения функций и числовых рядов: рекурсивно заданная функция определяет своё значение через обращение к себе самой с другими аргументами. При этом возможно два варианта:



Конечная рекурсивная функция. Она задаётся таким образом, чтобы для любого конечного аргумента за конечное число рекурсивных обращений привести к одному из отдельно определённых частных случаев, вычисляемых без рекурсии. Классический пример: рекурсивно-определённый факториал целого неотрицательного числа: Здесь каждое следующее рекурсивное обращение делается с аргументом, меньшим на единицу.

Поскольку n, по определению, целое неотрицательное число, через n рекурсивных обращений вычисление функции гарантированно придёт к частному случаю 0!=1, на котором рекурсия прекратится. Таким образом, несмотря на рекурсивность определения, вычисление функции для любого аргумента из области определения окажется конечным.

Бесконечная рекурсивная функция. Она задаётся в виде обращения к самой себе во всех случаях (по крайней мере, для некоторых из аргументов). Подобным образом могут задаваться бесконечные ряды, бесконечные непрерывные дроби и так далее. Практическое вычисление точного значения здесь, естественно, невозможно, поскольку потребует бесконечного времени. Требуемый результат находится аналитическими методами. Тем не менее, если речь идёт не о получении абсолютно точного значения, а о вычислении заданного приближения искомого значения, то тут в некоторых случаях возможно прямое использование рекурсивного определения: вычисления по нему ведутся до тех пор, пока необходимая точность не будет достигнута. Примером может служить разложение в непрерывную дробь числа e (рис 1):

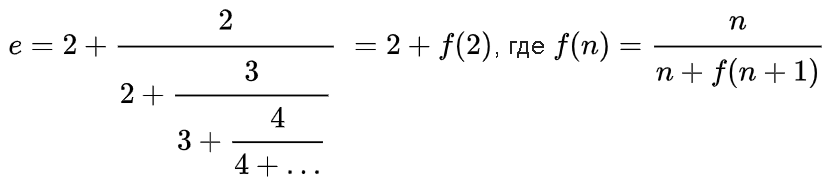


Рис 1 Разложение в непрерывную дробь числа e

Прямой расчёт по приведённой формуле вызовет бесконечную рекурсию, но можно доказать, что значение f(n) при возрастании n стремится к единице (поэтому, несмотря на бесконечность ряда, значение числа Эйлера конечно). Для приближённого вычисления значения e достаточно искусственно ограничить глубину рекурсии некоторым наперёд заданным числом и по достижении его использовать вместо f(n) единицу.

Другим примером рекурсии в математике является числовая последовательность, заданная рекуррентной формулой, когда каждый следующий член последовательности вычисляется как результат функции от n предыдущих членов. Таким образом с помощью конечного выражения (представляющего собой совокупность рекуррентной формулы и набора значений для первых n членов ряда) может даваться определение бесконечной последовательности.

С рекурсией тесно связана математическая индукция: она является естественным способом доказательства свойств функций на натуральных числах, рекурсивно заданных через свои меньшие значения.

В математике отдельно рассматривается класс так называемых «примитивно рекурсивных» функций. Определение примитивно рекурсивной функции также рекурсивно, оно задаёт набор примитивных функций и набор правил; функция является примитивно рекурсивной, если она может быть представлена как комбинация примитивно рекурсивных функций, образованная по этим правилам.

Примеры рекурсии в математике:

Метод Гаусса — Жордана для решения систем линейных алгебраических уравнений является рекурсивным.

Уже упоминавшийся факториал целого неотрицательного числа.

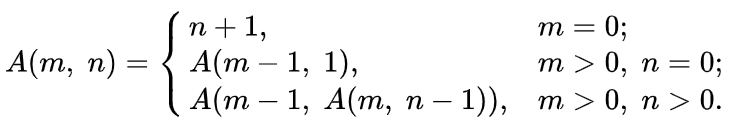
Числа Фибоначчи определяются с помощью рекуррентного соотношения:

Первое и второе числа Фибоначчи равны 1

Для n>2n-e число Фибоначчи равно сумме (n-1)-го и(n-2)-го чисел Фибоначчи.

Практически все геометрические фракталы задаются в форме бесконечной рекурсии (например, треугольник Серпинского).

Стандартный пример вычислимой рекурсивной функции, не являющейся примитивно рекурсивной — функция Аккермана: для неотрицательных целых чисел m и n следующим образом:



В программировании

В программировании рекурсия — вызов функции (процедуры) из неё же самой, непосредственно (простая рекурсия) или через другие функции (сложная или косвенная рекурсия), например, функция A вызывает функцию B, а функция B — функцию A. Количество вложенных вызовов функции или процедуры называется глубиной рекурсии. Рекурсивная программа позволяет описать повторяющееся или даже потенциально бесконечное вычисление, причём без явных повторений частей программы и использования циклов.

Структурно рекурсивная функция на верхнем уровне всегда представляет собой команду ветвления (выбор одной из двух или более альтернатив в зависимости от условия (условий), которое в данном случае уместно назвать «условием прекращения рекурсии»), имеющую две или более альтернативные ветви, из которых хотя бы одна является рекурсивной и хотя бы одна — терминальной. Рекурсивная ветвь выполняется, когда условие прекращения рекурсии ложно, и содержит хотя бы один рекурсивный вызов — прямой или опосредованный вызов функцией самой себя. Терминальная ветвь выполняется, когда условие прекращения рекурсии истинно; она возвращает некоторое значение, не выполняя рекурсивного вызова. Правильно написанная рекурсивная функция должна гарантировать, что через конечное число рекурсивных вызовов будет достигнуто выполнение условия прекращения рекурсии, в результате чего цепочка последовательных рекурсивных вызовов прервётся и выполнится возврат.

Помимо функций, выполняющих один рекурсивный вызов в каждой рекурсивной ветви, бывают случаи «параллельной рекурсии», когда на одной рекурсивной ветви делается два или более рекурсивных вызова. Параллельная рекурсия типична при обработке сложных структур данных, таких как деревья. Простейший пример параллельно-рекурсивной функции — вычисление ряда Фибоначчи, где для получения значения n-го члена необходимо вычислить (n-1)-й и (n-2)-й.

Реализация рекурсивных вызовов функций в практически применяемых языках и средах программирования, как правило, опирается на механизм стека вызовов — адрес возврата и локальные переменные функции записываются в стек, благодаря чему каждый следующий рекурсивный вызов этой функции пользуется своим набором локальных переменных и за счёт этого работает корректно. Оборотной стороной этого довольно простого по структуре механизма является то, что на каждый рекурсивный вызов требуется некоторое количество оперативной памяти компьютера, и при чрезмерно большой глубине рекурсии может наступить переполнение стека вызовов.

Вопрос о желательности использования рекурсивных функций в программировании неоднозначен: с одной стороны, рекурсивная форма может быть структурно проще и нагляднее, в особенности, когда сам реализуемый алгоритм по сути рекурсивен. Кроме того, в некоторых декларативных или чисто функциональных языках (таких как Пролог или Haskell) просто нет синтаксических средств для организации циклов, и рекурсия в них — единственный доступный механизм организации повторяющихся вычислений. С другой стороны, обычно рекомендуется избегать рекурсивных программ, которые приводят (или в некоторых условиях могут приводить) к слишком большой глубине рекурсии. Так, широко распространённый в учебной литературе пример рекурсивного вычисления факториала является, скорее, примером того, как не надо применять рекурсию, так как приводит к достаточно большой глубине рекурсии и имеет очевидную реализацию в виде обычного циклического алгоритма.

Имеется специальный тип рекурсии, называемый «хвостовой рекурсией» (структура рекурсивного алгоритма такова, что рекурсивный вызов является последней выполняемой операцией в функции, а его результат непосредственно возвращается в качестве результата функции). Интерпретаторы и компиляторы функциональных языков программирования, поддерживающие оптимизацию кода (исходного или исполняемого), автоматически преобразуют хвостовую рекурсию к итерации, благодаря чему обеспечивается выполнение алгоритмов с хвостовой рекурсией в ограниченном объёме памяти. Такие рекурсивные вычисления, даже если они формально бесконечны (например, когда с помощью рекурсии организуется работа командного интерпретатора, принимающего команды пользователя), никогда не приводят к исчерпанию памяти. Однако далеко не всегда стандарты языков программирования чётко определяют, каким именно условиям должна удовлетворять рекурсивная функция, чтобы транслятор гарантированно преобразовал её в итерацию. Одно из редких исключений — язык Scheme (диалект языка Lisp), описание которого содержит все необходимые сведения.

Теоретически, любую рекурсивную функцию можно заменить циклом и стеком. Однако такая модификация, как правило, бессмысленна, так как приводит лишь к замене автоматического сохранения контекста в стеке вызовов на ручное выполнение тех же операций с тем же или большим расходом памяти. Исключением может быть ситуация, когда рекурсивный алгоритм приходится моделировать на языке, в котором рекурсия запрещена.

*Рекурсия* — это свойство объекта подражать самому себе. Объект является рекурсивным если его части выглядят также как весь объект. Рекурсия очень широко применяется в математике и программировании.

# 2. РЕШЕНИЕ ЗАДАЧ С ПРИМЕНЕНИЕМ РЕКУРСИВНЫХ АЛГОРИТМОВ

## 2.1 Запись последовательность чисел в обратном порядке

Рекурсивный алгоритм всегда разбивает задачу на части, которые по своей структуре являются такими же как исходная задача, но более простыми. Для решения подзадач функция вызывается рекурсивно, а их результаты каким-либо образом объединяются. Разделение задачи происходит лишь тогда, когда ее не удается решить сразу (она является слишком сложной).

Одним из самых простых примеров рекурсии является написание чисел в обратном порядке. Условием завершения рекурсии будет сам аргумент функции, который следует уменьшать на единицу, пока он >= 1.

1. начало; rown(number);
2. number = Ввод
3. начало циклапока number>= 1
4. вывод number
5. number:= number-1
6. конец цикла
7. конец;

## 2.2 Двоичный поиск в массиве

Двоичный поиск выполняется над отсортированным массивом. На каждом шаге искомый элемент сравнивается со значением, находящимся посередине массива. В зависимости от результатов сравнения либо левая, либо правая части могут быть «отброшены».

1. начало; binary\_search(array, begin, end, element) ;
2. выполняет поиск элемента со значением element;
3. в массиве упорядоченном по возрастанию массиве array;
4. между индексами begin и end
5. если begin>end
6. конец; вернуть false - элемент не найден
7. mid := (end + begin) div 2; вычисление индекса элемента посередине рассматриваемой части массива
8. если array[mid] = element
9. конец; вернуть true (элемент найден)
10. если array[mid] <element
11. результат := binary\_search(array, mid+1, end, element)
12. иначе
13. результат := binary\_search(array, begin, mid, element)
14. конец; вернуть результат

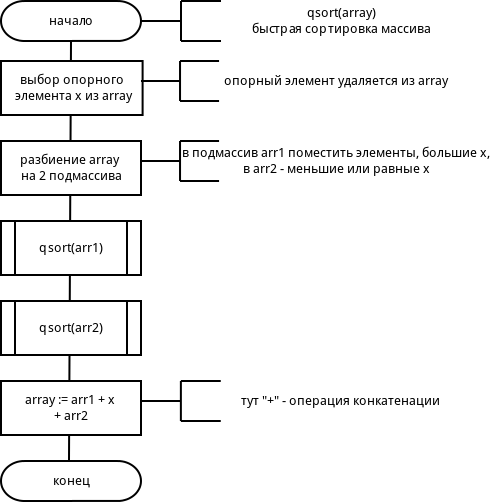
## 2.3 Вычисление чисел Фибоначчи

Числа Фибоначчи определяются рекуррентным выражением, т.е. таким, что вычисление элемента которого выражается из предыдущих элементов: F0=0,F1=1,Fn=Fn−1+Fn−2,n>2.

1. начало; fibonacci(number)
2. если number = 0
3. конец; вернуть 0
4. если number = 1
5. конец; вернуть 1
6. fib\_1 := fibonacci(number-1)
7. fib\_2 := fibonacci(number-2)
8. результат := fib\_1 + fib\_2
9. конец; вернуть результат

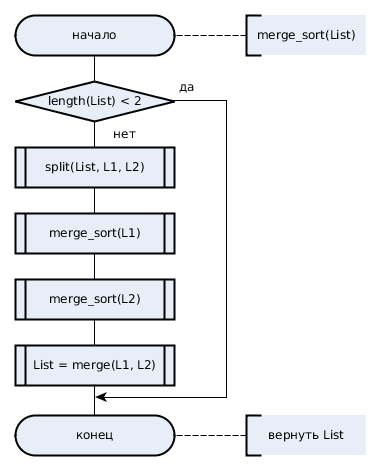
## 2.4 Быстрая сортировка (quicksort)

Алгоритм быстрой сортировки на каждом шаге выбирает один из элементов (опорный) и относительно него разделяет массив на две части, которые обрабатываются рекурсивно. В одну часть помещаются элементы меньше опорного, а в другую — остальные.



## 2.5 Сортировка слиянием (mergesort)

В основе алгоритма сортировки слиянием лежит возможность быстрого объединения упорядоченных массивов (или списков) так, чтобы результат оказался упорядоченным. Алгоритм разделяет исходный массив на две части произвольным образом (обычно пополам), рекурсивно сортирует их и объединяет результат. Разделение происходит до тех пор, пока размер массива больше единицы, т.к. пустой массив и массив из одного элемента всегда отсортированы.



На каждом шаге слияния из обоих списков выбирается первый необработанный элемент. Элементы сравниваются, наименьший из них добавляется к результату и помечается как обработанный. Слияние происходит до тех пор, пока один из списков не окажется пуст.

1. начало; merge(Array1, Size1, Array2, Size2) ;
2. исходные массивы упорядочены;
3. в результат формируется упорядоченный массив длины Size1+Size2
4. i := 0, j := 0
5. вечный\_цикл
6. если i >= Size1
7. дописать элементы от j до Size2 массива Array2 в конец результата
8. выход из цикла
9. если j >= Size2
10. дописать элементы от i до Size1 массива Array1 в конец результата
11. выход из цикла
12. если Array1[i] < Array2[j]
13. результат[i+j] := Array1[i]
14. i := i + 1
15. иначе (если Array1[i] >= Array2[j])
16. результат[i+j] := Array2[j]
17. j := j + 1
18. конец; вернуть результат

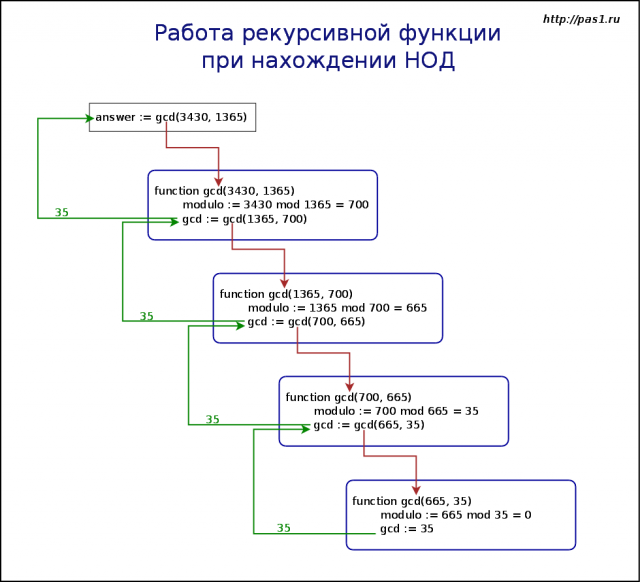
## 2.6 Фракталы: круговой фрактал

В основе создания фрактала лежит свойством самоподобия, то есть составленная из нескольких частей, каждая из которых подобна всей фигуре целиком.

В окружность радиуса R вписывают семь окружностей радиуса R/3 таким образом, чтобы они все касались, но не пересекали друг друга. В каждую из этих семи окружностей вписываются по семь окружностей R/9 и т. д.

## 2.7 Нахождение наибольшего общего делителя

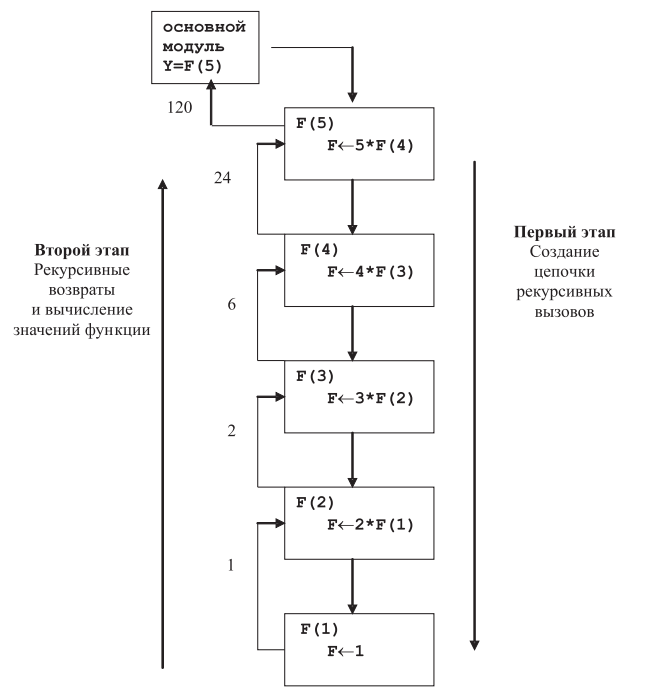
В основе программы подсчета НОД лежит метод Евклида и рекурсия. Алгоритм Евклида является универсальным способом, позволяющим вычислять наибольший общий делитель двух положительных целых чисел.



1. Из основной ветки программы вызывается функция gcd. Результат работы данной функции в дальнейшем будет присвоен переменной answer.
2. В функции gcd вычисляется остаток от деления чисел 3430 и 1365. Поскольку он не равен нулю, то осуществляется повторный вызов функции, но уже с числами 1365 и 700 (700 – это остаток от первого деления).
3. При третьем вызове функции передаются числа 700 и 665.
4. При четвертом – 665 и 35. Здесь остаток равен 0. Следовательно, результатом работы функции является число 35 (gcd = 35).
5. Результат четвертого вызова возвращается в третий.
6. Третий – во второй.
7. Второй – в первый.
8. Первый – в основную ветку программы и присваивается переменной answer.

## 2.8 Вычисление факториала

Факториалом числа называют произведение всех натуральных чисел до него включительно. Формула для нахождения факториала n! = 1 \* … \* (n-2) \* (n-1) \* n.



# 3. Анализ рекурсивных алгоритмов

При анализе сложности циклических алгоритмов рассчитывается трудоемкость итераций и их количество в наихудшем, наилучшем и среднем случаях,однако не получится применить такой подход к рекурсивной функции, т.к. в результате будет получено рекуррентное соотношение. Например, для функции поиска элемента в массиве:

Tsearchn={O(1)O(1)+O(Tsearchn−1)n=0n>0

Рекуррентные отношения не позволяют нам оценить сложность — мы не можем их просто так сравнивать, а значит, и сравнивать эффективность соответствующих алгоритмов. Необходимо получить формулу, которая опишет рекуррентное отношение — универсальным способом сделать это является подбор формулы при помощи метода подстановки, а затем доказательство соответствия формулы отношению методом математической индукции.

## 3.1 Метод подстановки (итераций)

Заключается в последовательной замене рекуррентной части в выражении для получения новых выражений. Замена производится до тех пор, пока не получится уловить общий принцип и выразить его в виде не рекуррентной формулы. Например, для поиска элемента в массиве:

Tsearchn=O(1)+O(Tsearchn−1)=2×O(1)+O(Tsearchn−2)=3×O(1)+O(Tsearchn−3)

Можно предположить, что Tsearchn = Tsearchn−k+k×O(1), но тогда Tsearchn =Tsearch0+n×O(1)=O(n).

Мы вывели формулу, однако первый шаг содержит предположение, т.е. не имеется доказательства соответствия формулы рекуррентному выражению - получить доказательство позволяет метод математической индукции.

## 3.2 Метод математической индукции

Позволяет доказать истинность некоторого утверждения (Pn), состоит из двух шагов:

доказательство утверждения для одного или нескольких частных случаев P0,P1,…;

из истинности Pn (индуктивная гипотеза) и частных случаев выводится доказательство Pn+1.

Докажем корректность предположения, сделанного при оценки трудоемкости функции поиска (Tsearchn=(n+1)×O(1)):

Tsearch1=2×O(1) верно из условия (можно подставить в исходную рекуррентную формулу);

допустим истинность Tsearchn=(n+1)×O(1);

требуется доказать, что Tsearchn+1=((n+1)+1)×O(1)=(n+2)×O(1);

подставим n+1 в рекуррентное соотношение: Tsearchn+1=O(1)+Tsearchn;

в правой части выражения возможно произвести замену на основании индуктивной гипотезы: Tsearchn+1=O(1)+(n+1)×O(1)=(n+2)×O(1);

утверждение доказано.

Часто, такое доказательство — достаточно трудоемкий процесс, но еще сложнее выявить закономерность используя метод подстановки. В связи с этим применяется, так называемый, общий метод [5].

Общий (основной) метод решения рекуррентных соотношений

Общий метод не является универсальным, например с его помощью невозможно провести оценку сложности приведенного выше алгоритма вычисления чисел Фибоначчи. Однако, он применим для всех случаев использования подхода «разделяй и властвуй» [3]:

Tn=a⋅T(nb)+fn;a,b=const,a≥1,b>1,fn>0,∀n.

Уравнения такого вида получаются если исходная задача разделяется на a подзадач, каждая из которых обрабатывает nb элементов. fn — трудоемкость операций разбиения задачи на части и комбинирование решений. Помимо вида соотношения, общий метод накладывает ограничения на функцию fn, выделяя три случая:

∃ε>0:fn=O(nlogba—ε)⇒Tn=Θ(nlogba);

fn=Θ(nlogba)⇒Tn=Θ(nlogba⋅logn);

∃ε>0,c<1:fn=Ω(nlogba+ε),fnb≤c⋅fn⇒Tn=Θ(fn).

Правильность утверждений для каждого случая доказана формально [6]. Задача анализа рекурсивного алгоритма теперь сводится к определению случая основной теоремы, которому соответствует рекуррентное соотношение.

## 3.3 Анализ алгоритма бинарного поиска

Алгоритм разбивает исходные данные на 2 части (b = 2), но обрабатывает лишь одну из них (a = 1), fn=1. nlogba=nlog21=n0=1. Функция разделения задачи и компоновки результата растет с той же скоростью, что и nlogba, значит необходимо использовать второй случай теоремы:

TbinarySearchn=Θ(nlogba⋅logn)=Θ(1⋅logn)=Θ(logn).

## 3.4 Анализ алгоритма поиска

Рекурсивная функция разбивает исходную задачу на одну подзадачу (a = 1), данные делятся на одну часть (b = 1). Мы не можем использовать основную теорему для анализа этого алгоритма, т.к. не выполняется условие b>1.

Для проведения анализа может использоваться метод подстановки или следующие рассуждения: каждый рекурсивный вызов уменьшает размерность входных данных на единицу, значит всего их будет n штук, каждый из которых имеет сложность O(1). Тогда Tsearchn=n⋅O(1)=O(n).

## 3.5 Анализ алгоритма сортировки слиянием

Исходные данные разделяются на две части, обе из которых обрабатываются: a=2,b=2,nlogba=n.

При обработке списка, разделение может потребовать выполнения Θ(n) операций, а для массива — выполняется за постоянное время (Θ(1)). Однако, на соединение результатов в любом случае будет затрачено Θ(n), поэтому fn=n.

Используется второй случай теоремы: TmergeSortn=Θ(nlogba⋅logn)=Θ(n⋅logn).

## 3.6 Анализ трудоемкости быстрой сортировки

В лучшем случае исходный массив разделяется на две части, каждая из которых содержит половину исходных данных. Разделение потребует выполнения n операций. Трудоемкость компоновки результата зависит от используемых структур данных — для массива O(n), для связного списка O(1). a=2,b=2,fn=b, значит сложность алгоритма будет такой же как у сортировки слиянием: TquickSortn=O(n⋅logn).

Однако, в худшем случае в качестве опорного будет постоянно выбираться минимальный или максимальный элемент массива. Тогда b=1, а значит, мы опять не можем использовать основную теорему. Однако, мы знаем, что в этом случае будет выполнено n рекурсивных вызовов, каждый из которых выполняет разделение массива на части (O(n)) — значит сложность алгоритма TquickSortn=O(n2).

При анализе быстрой сортировки методом подстановки, пришлось бы также рассматривать отдельно наилучший и наихудший случаи.

## 3.7 Хвостовая рекурсия и цикл

Анализ трудоемкости рекурсивных функций значительно сложнее аналогичной оценки циклов, но основной причиной, по которой циклы предпочтительнее являются высокие затраты на вызов функции.

После вызова управление передается другой функции. Для передачи управления достаточно изменить значение регистра программного счетчика, в котором процессор хранит номер текущей выполняемой команды — аналогичным образом передается управление ветвям алгоритма, например, при использовании условного оператора. Однако, вызов — это не только передача управления, ведь после того, как вызванная функция завершит вычисления, она должна вернуть управление в точку, и которой осуществлялся вызов, а также восстановить значения локальных переменных, которые существовали там до вызова.

Для реализации такого поведения используется стек (стек вызовов, callstack) — в него помещаются номер команды для возврата и информация о локальных переменных. Стек не является бесконечным, поэтому рекурсивные алгоритмы могут приводить к его переполнению, в любом случае на работу с ним может уходить значительная часть времени.

В ряде случаев рекурсивную функцию достаточно легко заменить циклом, например, рассмотренные выше алгоритмы поиска и бинарного поиска [4]. В некоторых случаях требуется более творческий подход, но чаще всего такая замена оказывается возможной. Кроме того, существует особый вид рекурсии, когда рекурсивный вызов является последней операцией, выполняемой функцией. Очевидно, что в таком случае вызывающая функция не будет каким-либо образом изменять результат, а значит ей нет смысла возвращать управление. Такая рекурсия называется хвостовой — компиляторы автоматически заменяют ее циклом.

Зачастую сделать рекурсию хвостовой помогает метод накапливающего параметра [7], который заключается в добавлении функции дополнительного аргумента-аккумулятора, в котором накапливается результат. Функция выполняет вычисления с аккумулятором до рекурсивного вызова. Хорошим примером использования такой техники служит функция вычисления факториала:

factn=n⋅fact(n−1)fact3=3⋅fact2=3⋅(2⋅fact1)=3⋅(2⋅(1⋅fact0))=6factn=factTailn,1factTailn,accumulator=factTail(n−1,accumulator⋅n)factTail3,1=factTail2,3=factTail1,6=factTail0,6=6

В качестве более сложного примера рассмотрим функцию вычисления чисел Фибоначчи. Основная функция вызывает вспомогательную,использующую метод накапливающего параметра, при этом передает в качестве аргументов начальное значение итератора и два аккумулятора (два предыдущих числа Фибоначчи).

1. начало; fibonacci(number)
2. вернуть fibonacci(number, 1, 1, 0)
3. конец
4. начало; fibonacci(number, iterator, fib1, fib2)
5. если iterator == number вернуть fib1
6. вернутьfibonacci(number, iterator + 1, fib1 + fib2, fib1)
7. конец

Функция с накапливающим параметром возвращает накопленный результат, если рассчитано заданное количество чисел, в противном случае — увеличивает счетчик, рассчитывает новое число Фибоначчи и производит рекурсивный вызов. Оптимизирующие компиляторы могут обнаружить, что результат вызова функции без изменений передается на выход функции и заменить его циклом. Такой прием особенно актуален в функциональных и логических языках программирования, т.к. в них программист не может явно использовать циклические конструкции.

# ЗАКЛЮЧЕНИЕ

На основании проведенного исследования можно сделать несколько выводов:

во – первых, рекурсивные алгоритмы есть универсальное средство решения разнообразных алгоритмических проблем. Любая разрешимая задача такого рода имеет рекурсивное решение, которое при этом отличается изяществом и простотой для восприятия человеком;

во – вторых, рекурсивные алгоритмы часто имеют более низкую асимптотическую сложность, чем эквивалентные им итерационные. То есть теоретически они быстрее;

в – третьих, развитие современных программных средств сделало практическое использование рекурсии достаточно несложным делом, а новые концепции и технологии программирования преодолели проблему низкой эффективности рекурсивных программ, созданную необходимостью вызова большого количества подпроцедур.

Рекурсия отражает черту абстрактного мышления, проявляющуюся в самых различных приложениях (в математике, синтаксическом анализе и трансляции, древовидной сортировке и обработке данных с динамической структурой, шахматных задачах и т.д.). Именно в задачах такого рода лучше применять рекурсивные алгоритмы, так как они, избавляют от необходимости последовательного описания процессов, к тому же, практически все действующие языки программирования поддерживают рекурсию.

# СПИСОК ЛИТЕРАТУРЫ

1. Многопоточный сервер Qt. Пул потоков. Паттерн Decorator[Электронный ресурс] – режим доступа: https://pro-prof.com/archives/1390. Дата обращения: 21.02.2015.
2. Джейсон Мак-Колм Смит [Элементарные шаблоны проектирования](https://pro-prof.com/books-software-design) : Пер. с англ. — М. : ООО “И.Д. Вильямс”, 2013. — 304 с.
3. Скиена С. Алгоритмы. Руководство по разработке.-2-е изд.: пер. с англ.-СПб.:БХВ-Петербург, 2011.-720с.: ил.
4. Васильев В. С. Анализ сложности алгоритмов. Примеры [Электронный ресурс] – режим доступа: https://pro-prof.com/archives/1660. Дата обращения: 21.02.2015.
5. А.Ахо, Дж.Хопкрофт, Дж.Ульман, Структуры данных и алгоритмы, М., Вильямс, 2007.
6. Миллер, Р. Последовательные и параллельные алгоритмы: Общий подход / Р. Миллер, Л. Боксер ; пер. с англ. — М. : БИНОМ. Лаборатория знаний, 2006. — 406 с.
7. Сергиевский Г.М. Функциональное и логическое программирование : учеб. пособие для студентов высш. учеб. заведений / Г.М. Сергиевский, Н.Г. Волченков. — М.: Издательский центр «Академия», 2010.- 320с.
8. <https://pro-prof.com/archives/813>
9. Головешкин В.А., Ульянов М.В. Теория рекурсии для программистов. – М.Издательство: «[Физматлит](https://www.labirint.ru/pubhouse/3853/)», 2006.-296