

**Частное учреждение профессионального образования**

**«Высшая школа предпринимательства»**

**(ЧУПО «ВШП»)**

ОТЧЕТ ПО ПРАКТИКЕ

по основной образовательной программе

среднего профессионального образования по специальности

09.02.07 «Информационные системы и программирование»

Вид практики (учебная, производственная, преддипломная): \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Установленный по КУГ срок прохождения практики: с \_\_.\_\_.20\_\_г. по \_\_.\_\_.20\_\_г.

Место прохождения практики (наименование организации):

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |  |  |
| --- | --- | --- |
| Выполнил студент  \_\_-го курса | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  *(подпись)* | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  *(фамилия, имя, отчество)*  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
| Руководитель от образовательной организации | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  *(подпись)* | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  *(ученая степень, фамилия, имя, отчество)*  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  *(должность)* |
| Руководитель от предприятия | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  *(подпись)* | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  *(ученая степень, фамилия, имя, отчество)*  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  *(должность)* |
| Оценка | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |  |
|  | *(прописью)* | Дата сдачи отчета: \_\_.\_\_.20\_\_г. |
|  |  |  |

Тверь, 2024 г.

**Оглавление**

[Введение 3](#_Toc186144335)

[1. Алгоритмы для вычисления ряда Фибоначчи 5](#_Toc186144336)

[1.1. Вычисление n-го числа Фибоначчи с использованием рекурсивного алгоритма 5](#_Toc186144337)

[1.2. Вычисление n-го числа Фибоначчи с использованием цикла 6](#_Toc186144338)

[1.3. Вычисление n-го числа Фибоначчи с записью числового ряда в массив 8](#_Toc186144339)

[1.4. Вычисление n-го числа Фибоначчи при помощи формулы Бине 10](#_Toc186144340)

[1.5. Определение четности n-го большого числа Фибоначчи 11](#_Toc186144341)

[2. Алгоритмы Хаффмана 12](#_Toc186144342)

[2.1. Кодирование строки по алгоритму Хаффмана 13](#_Toc186144343)

[2.2. Декодирование строки по алгоритму Хаффмана 14](#_Toc186144344)

[Заключение 16](#_Toc186144345)

[Список источников 19](#_Toc186144346)

[Приложение 1. QR код. 20](#_Toc186144347)

[Приложение 2. Вычисление n-го числа Фибоначчи с использованием рекурсивного алгоритма 21](#_Toc186144348)

[Приложение 3. Вычисление n-го числа Фибоначчи с использованием цикла 22](#_Toc186144349)

[Приложение 4. Вычисление n-го числа Фибоначчи с записью числового ряда в массив 23](#_Toc186144350)

[Приложение 5. Вычисление n-го числа Фибоначчи при помощи формулы Бине 24](#_Toc186144351)

[Приложение 6. Определение четности n-го большого числа Фибоначчи 25](#_Toc186144352)

[Приложение 7. Кодирование строки по алгоритму Хаффмана 26](#_Toc186144353)

[Приложение 8. Декодирование строки по алгоритму Хаффмана 28](#_Toc186144354)

[Приложение 9. Антиплагиат 29](#_Toc186144355)

# Введение

Алгоритмы – это набор четко определенных инструкций или шагов, которые выполняются в определенном порядке для решения конкретной задачи или достижения определенной цели. Ключевыми свойствами алгоритмов являются дискретность, определённость, результативность, массовость и конечность. [1]

Целью данной практики является изучение и применение основных алгоритмов для решения задач различной сложности. В рамках практики особое внимание уделялось разработке и анализу алгоритмов для вычисления чисел Фибоначчи, а также алгоритмов Хаффмана, используемых для кодирования и декодирования данных.

Были поставлены следующие задачи:

1. Изучить принципы работы линейных, разветвляющихся, циклических и рекурсивных алгоритмов.
2. Реализовать и проанализировать алгоритмы вычисления чисел Фибоначчи.
3. Разработать алгоритмы построения и использования дерева Хаффмана для кодирования данных.
4. Проверить работоспособность разработанных алгоритмов на практике и оценить их эффективность.

**Ключевые свойства алгоритмов**

1. **Дискретность**  
   Алгоритм состоит из отдельных, четко определенных шагов, которые выполняются последовательно и поэтапно. Каждый шаг может быть выполнен за конечный отрезок времени, что позволяет разделить процесс решения задачи на понятные и ограниченные этапы.
2. **Определённость**  
   Каждый шаг алгоритма строго определён, исключая любые неопределенности или двусмысленности. Это свойство гарантирует, что для одинаковых входных данных алгоритм всегда будет выдавать один и тот же результат.
3. **Результативность**  
   Алгоритм обязательно приводит к конечному результату за определенное количество шагов, если выполняются все условия. Результат может быть как успешным, так и указывать на отсутствие решения.
4. **Массовость**  
   Алгоритм должен быть применим не только к одной конкретной задаче, но и к широкому классу задач, входные данные для которых могут быть различными.
5. **Конечность**  
   Выполнение алгоритма завершается после конечного числа шагов. Это свойство исключает возможность бесконечного выполнения или зацикливания алгоритма.

**Основные виды алгоритмов**

1. **Линейные**  
   Выполняются последовательно, шаг за шагом.
2. **Разветвляющиеся**  
   Выполняют разные действия в зависимости от выполнения условий.
3. **Циклические**  
   Повторяют одни и те же действия до выполнения заданного условия.
4. **Рекурсивные**  
   Вызывают сами себя для решения более мелких подзадач.
5. **Параллельные**  
   Выполняются одновременно в нескольких потоках или процессах.

Эти виды охватывают основные подходы к построению и решению алгоритмических задач. [2]

# 1. [Алгоритмы для вычисления ряда Фибоначчи](https://it.vshp.online/#/pages/up02/up02_fibonacci?id=%d0%a3%d0%9f02-%d0%90%d0%bb%d0%b3%d0%be%d1%80%d0%b8%d1%82%d0%bc%d1%8b-%d0%b4%d0%bb%d1%8f-%d0%b2%d1%8b%d1%87%d0%b8%d1%81%d0%bb%d0%b5%d0%bd%d0%b8%d1%8f-%d1%80%d1%8f%d0%b4%d0%b0-%d0%a4%d0%b8%d0%b1%d0%be%d0%bd%d0%b0%d1%87%d1%87%d0%b8)

Вычисление ряда Фибоначчи — это классическая алгоритмическая задача, которую нередко дают на собеседованиях, когда хотят проверить что кандидат имеет некоторые представления о «классических» алгоритмах. [3]

## **1.1.** [**Вычисление n-го числа Фибоначчи с использованием рекурсивного алгоритма**](https://it.vshp.online/#/pages/up02/up02_tasks?id=_11-%d0%92%d1%8b%d1%87%d0%b8%d1%81%d0%bb%d0%b5%d0%bd%d0%b8%d0%b5-n-%d0%b3%d0%be-%d1%87%d0%b8%d1%81%d0%bb%d0%b0-%d0%a4%d0%b8%d0%b1%d0%be%d0%bd%d0%b0%d1%87%d1%87%d0%b8-%d1%81-%d0%b8%d1%81%d0%bf%d0%be%d0%bb%d1%8c%d0%b7%d0%be%d0%b2%d0%b0%d0%bd%d0%b8%d0%b5%d0%bc-%d1%80%d0%b5%d0%ba%d1%83%d1%80%d1%81%d0%b8%d0%b2%d0%bd%d0%be%d0%b3%d0%be-%d0%b0%d0%bb%d0%b3%d0%be%d1%80%d0%b8%d1%82%d0%bc%d0%b0)

Задача №1

*Дано целое число  , необходимо написать функцию  для вычисления n-го числа Фибоначчи с использованием рекурсии. Функция  должна вызывать сама себя в теле функции для вычисления соответствующих*

*В результате выполнения, функция должна вывести на экран вычисленное число Фибоначчи, например  должна вывести число 8, а  — соответственно 0.*

*Необходимо замерить время выполнения алгоритма с точностью до миллисекунды любым доступным способом для пяти произвольных n, и на основании произведенных замеров сделать предположение о сложности алгоритма.*

Для решения этой задачи я разработал следующий алгоритм:

def fib(n):

if n == 0:

return 0

elif n == 1:

return 1

else:

return fib(n - 1) + fib(n - 2)

В этом коде выполняется вычисление чисел Фибоначчи с помощью рекурсивной функции.  
В функции реализован следующий алгоритм:

Если функция возвращает 0, поскольку это первое число Фибоначчи.

Если , функция возвращает 1, соответствующее второму числу Фибоначчи.

Для всех остальных значений , функция вызывает сама себя для и , а затем складывает их результаты.

Рекурсивный подход основан на свойстве последовательности Фибоначчи:

Для анализа производительности алгоритма я также добавил замер времени выполнения. Полное описание алгоритма и его анализа представлено в приложении (см. Приложение 2).

Для проверки работы алгоритма я выбрал несколько значений , чтобы вычислить соответствующие числа Фибоначчи и замерить время выполнения.  
Результаты тестирования представлены ниже:

fib(5) = 5, время выполнения: 0.00 мс

fib(10) = 55, время выполнения: 0.00 мс

fib(15) = 610, время выполнения: 0.00 мс

fib(20) = 6765, время выполнения: 1.99 мс

fib(24) = 46368, время выполнения: 11.97 мс

Из результатов тестирования видно, что время выполнения быстро растёт при увеличении значения n. Это связано с экспоненциальной сложностью алгоритма: количество вызовов функции увеличивается многократно, и одни и те же значения вычисляются повторно.

## **1.2. Вычисление n-го числа Фибоначчи с использованием цикла**

**Задача №2**

*Дано целое число , необходимо написать функцию  для вычисления -го числа Фибоначчи с использованием цикла. Функция  должна производить расчет от 11 до , на каждой последующей итерации используя значение числа(чисел), необходимых для расчета, полученных на предыдущей итерации.*

*В результате выполнения, функция должна вывести на экран вычисленное число Фибоначчи, например  должна вывести число 2, а  — соответственно 13.*

*Необходимо замерить время выполнения алгоритма с точностью до миллисекунды любым доступным способом для пяти произвольных , и на основании произведенных замеров сделать предположение о сложности алгоритма.*

Для решения задачи был разработан алгоритм вычисления -го числа Фибоначчи с использованием цикла. Этот подход позволяет значительно оптимизировать вычисления по сравнению с рекурсивным методом, так как исключает повторяющиеся операции и сводит сложность алгоритма к .

def fib(n):

if n == 0:

return 0

elif n == 1:

return 1

a, b = 0, 1

for \_ in range(2, n + 1):

a, b = b, a + b

return b

В этом коде выполняется вычисление чисел Фибоначчи с помощью итеративного алгоритма.

Функция принимает одно целое число и возвращает соответствующее число Фибоначчи.

Если , функция возвращает 0. Если , возвращается 1, так как это базовые случаи последовательности.

Для всех значений :

Переменные *a* и *b* инициализируются как 0 и 1, что соответствует первым двум числам последовательности Фибоначчи.

Далее выполняется цикл, который обновляет значения *a* и *b* по правилу: , . Это позволяет вычислять каждое последующее число Фибоначчи, используя только два последних значения.

По завершении цикла итоговое значение возвращается как результат.

Этот алгоритм эффективен, поскольку сохраняет только два последних значения чисел Фибоначчи, что минимизирует затраты по памяти.

Результаты замера производительности:

fib(5) = 5, время выполнения: 0.00 мс

fib(10) = 55, время выполнения: 0.00 мс

fib(15) = 610, время выполнения: 0.00 мс

fib(20) = 6765, время выполнения: 0.00 мс

fib(32) = 2178309, время выполнения: 0.00 мс

На основании проведенного тестирования можно сделать вывод, что итеративный алгоритм для вычисления чисел Фибоначчи демонстрирует высокую эффективность даже для относительно больших значений . В отличие от рекурсивного подхода, данный алгоритм избегает многократного повторного вычисления одних и тех же значений, что существенно снижает время выполнения.

Результаты тестирования показывают, что время выполнения остаётся практически неизменным даже при увеличении , что подтверждает линейную сложность алгоритма .

Полный код с замером времени выполнения представлен в приложении (см. Приложение 2).

## **1.3. Вычисление n-го числа Фибоначчи с записью числового ряда в массив**

**Задача №3**

*Дано целое число  , необходимо написать функцию  для вычисления -го числа Фибоначчи. Функция должна в процессе выполнения записывать вычисленные значения в массив таким образом что индекс записанного числа в массиве должен соответствовать порядковому номеру числа Фибоначчи. При этом уже вычисленные значения должны браться из массива, а вновь вычисляемые должны записываться в массив только в случае если они еще не были вычислены.*

*В результате выполнения, функция должна вывести на экран массив, содержащий все вычисленные числа Фибоначчи вплоть до заданного, включая его например должна вывести массив*

Для решения этой задачи я разработал следующий алгоритм:

def fib(n):

if n == 0:

return [0]

elif n == 1:

return [0, 1]

fib\_sequence = [0, 1]

for i in range(2, n + 1):

fib\_sequence.append(fib\_sequence[i - 1] + fib\_sequence[i - 2])

return fib\_sequence

Здесь вычисляется числовой ряд Фибоначчи до n-го элемента с сохранением всех вычисленных значений в массив. Проверяется, если n = 0, возвращается массив [0]. Если , возвращается массив [0, 1]. Для остальных случаев создаётся массив с начальными значениями [0, 1]. С помощью цикла вычисляются следующие числа Фибоначчи, которые добавляются в массив. Возвращается итоговый массив, содержащий числовой ряд Фибоначчи до n-го элемента. Полный код с вычислением находится в приложении (см. Приложение 4).

В качестве тестирования было задано значение . Тестирование выдало следующий результат:

fib(8) = [0, 1, 1, 2, 3, 5, 8, 13, 21]

Данный подход позволяет не только получить n-е число Фибоначчи, но и одновременно сохранить весь числовой ряд до заданного элемента. Результаты тестирования подтверждают корректность алгоритма и его способность обрабатывать числа в заданном диапазоне. Такой метод удобен для анализа последовательности чисел Фибоначчи и может быть расширен для решения задач, связанных с обработкой данных из числового ряда.

## [**1.4. Вычисление n-го числа Фибоначчи при помощи формулы Бине**](https://it.vshp.online/#/pages/up02/up02_tasks?id=_14-%d0%92%d1%8b%d1%87%d0%b8%d1%81%d0%bb%d0%b5%d0%bd%d0%b8%d0%b5-n-%d0%b3%d0%be-%d1%87%d0%b8%d1%81%d0%bb%d0%b0-%d0%a4%d0%b8%d0%b1%d0%be%d0%bd%d0%b0%d1%87%d1%87%d0%b8-%d0%bf%d1%80%d0%b8-%d0%bf%d0%be%d0%bc%d0%be%d1%89%d0%b8-%d1%84%d0%be%d1%80%d0%bc%d1%83%d0%bb%d1%8b-%d0%91%d0%b8%d0%bd%d0%b5)

**Задание №4**

*Дано целое число 1≤n≤64 , необходимо написать функцию fib(n) для вычисления -го числа Фибоначчи. Функция  должна производить вычисление по формуле Бине.*

*Важно учесть что Формула Бине точна математически, но компьютер оперирует дробями конечной точности, и при действиях над ними может накопиться ошибка, поэтому при проверке результатов необходимо производить округление и выбирать соответствующие типы данных.*

*В результате выполнения, функция должна вывести на экран вычисленное число Фибоначчи, например  должна вывести число 2178309.*

Для решения этой задачи я разработал следующий алгоритм:

def fib(n):

sqrt\_5 = math.sqrt(5)

phi = (1 + sqrt\_5) / 2

psi = (1 - sqrt\_5) / 2

fib\_n = (phi\*\*n - psi\*\*n) / sqrt\_5

return round(fib\_n)

В первую очередь вычисляется квадратный корень из 5, затем определяются два ключевых значения: золотое сечение, обратное золотое сечение. Далее вычисляется само число Фибоначчи по формуле Бине. Результат округляется с помощью функции для устранения возможных погрешностей, связанных с вычислениями с плавающей точкой.

Для тестирования я взял значение . Полный код включающий тестирование находится в приложении (см. Приложение 5).

Результат тестирования:

fib(32) = 2178309

В данном задаче было выполнено вычисление n-го числа Фибоначчи с использованием формулы Бине, что позволило обеспечить высокую эффективность вычислений с константной временной сложностью .

Данный алгоритм демонстрирует значительные преимущества в скорости, так как вычисление числа Фибоначчи осуществляется за одно математическое выражение, без необходимости итераций или рекурсии. Однако, важно учитывать возможные погрешности для больших значений , возникающие из-за операций с плавающей точкой. Для снижения погрешностей применялась функция

Алгоритм является удобным для быстрого получения числа Фибоначчи, но требует внимательного подхода при работе с большими значениями.

## **1.5. Определение четности n-го большого числа Фибоначчи**

**Задача №5**

*Дано целое число , необходимо написать функцию  для определения четности -го числа Фибоначчи.*

*Как мы помним, числа Фибоначчи растут очень быстро, поэтому при их вычислении нужно быть аккуратным с переполнением. В данной задаче, впрочем, этой проблемы можно избежать, поскольку нас интересует только последняя цифра числа Фибоначчи: если  — последние цифры чисел  и  соответственно, то — последняя цифра числа .*

*В результате выполнения, функция должна вывести на экран четное ли число или нет ( или  соответственно), например  должна вывести , т.к. последняя цифра данного числа — 5.*

Для решения этой задачи я написал алгоритм:

def fib\_eo(n):

if n == 0:

return "even"

elif n == 1:

return "odd"

a, b = 0, 1

for \_ in range(2, n + 1):

a, b = b, (a + b) % 10

return "even" if b % 2 == 0 else "odd"

В данном алгоритме реализовано определение четности n-го числа Фибоначчи с использованием вычислений только последней цифры чисел Фибоначчи, чтобы избежать переполнения. Сначала обрабатываются базовые случаи (результат "") и (результат ""). Далее, начиная с , значения чисел Фибоначчи обновляются в цикле с помощью переменных и , где хранит текущую последнюю цифру числа. На каждой итерации вычисляется новая последняя цифра числа Фибоначчи как . В результате возвращается "", если последняя цифра четная , или "", если нечетная. Полный код находится в приложении (см. Приложение 6).

Для определения чётности -го большого числа я задал значение . В результате вывод показал (нечётное), так как последняя цифра этого числа — 3, что подтверждает корректность работы. Алгоритм обладает временной сложностью , что делает его чрезвычайно быстрым и экономичным с точки зрения производительности.

# [2. Алгоритмы Хаффмана](https://it.vshp.online/#/pages/up02/up02_tasks?id=_2-%d0%97%d0%b0%d0%b4%d0%b0%d1%87%d0%b8-%d0%bf%d0%be-%d0%b0%d0%bb%d0%b3%d0%be%d1%80%d0%b8%d1%82%d0%bc%d0%b0%d0%bc-%d0%a5%d0%b0%d1%84%d1%84%d0%bc%d0%b0%d0%bd%d0%b0)

**Алгоритм Хаффмана** (англ. *Huffman's algorithm*) — алгоритм оптимального префиксного кодирования некоторого алфавита с минимальной избыточностью. Был разработан в 1952 году аспирантом Массачусетского технологического института Дэвидом Альбертом Хаффманом при написании им курсовой работы.

Основная идея алгоритма состоит в следующем: зная вероятности появления символов в сообщении, можно описать процедуру построения кодов переменной длины, состоящих из целого количества битов. Символам с большей вероятностью ставятся в соответствие более короткие коды. Коды Хаффмана обладают свойством префиксности (то есть ни одно кодовое слово не является префиксом другого), что позволяет однозначно их декодировать. Часто благодаря особому свойству префиксности данные коды называют **беспрефиксными.** [4]

## [**2.1. Кодирование строки по алгоритму Хаффмана**](https://it.vshp.online/#/pages/up02/up02_tasks?id=_21-%d0%9a%d0%be%d0%b4%d0%b8%d1%80%d0%be%d0%b2%d0%b0%d0%bd%d0%b8%d0%b5-%d1%81%d1%82%d1%80%d0%be%d0%ba%d0%b8-%d0%bf%d0%be-%d0%b0%d0%bb%d0%b3%d0%be%d1%80%d0%b8%d1%82%d0%bc%d1%83-%d0%a5%d0%b0%d1%84%d1%84%d0%bc%d0%b0%d0%bd%d0%b0)

**Задача №1**

По данной строке, состоящей из строчных букв латинского алфавита:

Errare humanum est.

постройте оптимальный беспрефиксный код на основании классического алгоритма кодирования Хаффмана.

В результате выполнения, функция huffman\_encode() должна вывести на экран в первой строке — количество уникальных букв, встречающихся в строке и размер получившейся закодированной строки в битах. В следующих строках запишите коды символов в формате "'symbol': code". В последней строке выведите саму закодированную строку.

Пример вывода для данного текста:

12 67

' ': 000

'.': 1011

'E': 0110

'a': 1110

'e': 1111

'h': 0111

'm': 010

'n': 1000

'r': 110

's': 1001

't': 1010

'u': 001

0110110110111011011110000111001010111010000010100001111100110101011

В решении задачи кодирования строки по алгоритму Хаффмана был реализован алгоритм (см. Приложение 7) в которому используется функция , которая предназначена для декодирования строки, закодированной с использованием алгоритма Хаффмана. Эта функция принимает 2 параметра:

— словарь, где ключ — символ, а значение — его Хаффман-код.

— строка, закодированная в виде последовательности битов функция преобразует последовательность битов в раскодированную строку, сопоставляя каждый Хаффман-код из закодированной строки с соответствующим символом из словаря , и возвращает её.

## **2.2. Декодирование строки по алгоритму Хаффмана**

**Задача №2**

Восстановите строку по её коду и беспрефиксному коду символов.

12 60

' ': 1011

'.': 1110

'D': 1000

'c': 000

'd': 001

'e': 1001

'i': 010

'm': 1100

'n': 1010

'o': 1111

's': 011

'u': 1101

100011110001001101000111111011001010011000010110011010111110

В первой строке входного файла заданы два целых числа через пробел: первое число — количество различных букв встречающихся в строке, второе число — размер получившейся закодированной строки, соответственно. В следующих строках записаны коды символов в формате "'symbol': code". Символы могут быть перечислены в любом порядке. Каждый из этих символов встречается в строке хотя бы один раз. В последней строке записана закодированная строка. Заданный код таков, что закодированная строка имеет минимальный возможный размер.

Пример вывода для данного текста:

Docendo discimus.

В решении задачи декодирования строки по алгоритму Хаффмана был реализован алгоритм (см. Приложение 8) , где используется функция , которая принимает два параметра: словарь , где каждому символу соответствует его Хаффман-код, и строку , состоящую из битов, закодированных этим кодом. В процессе декодирования функция создает обратное отображение, преобразуя коды в символы, после чего последовательно читает биты из закодированной строки, накапливая их в буфере. Когда буфер совпадает с одним из кодов, соответствующий символ добавляется в результирующую строку, а буфер обнуляется. Таким образом, строка восстанавливается из её битового представления. Код выводит на экран результат декодирования — в данном случае строку "Docendo discimus.".

# Заключение

В рамках практики были рассмотрены ключевые аспекты разработки алгоритмов, их классификация, а также методы оценки эффективности. Основное внимание было уделено таким свойствам алгоритмов, как:

1. Дискретность — алгоритм представляет собой последовательность отдельных шагов;
2. Определённость — каждый шаг алгоритма чётко определён и не допускает двусмысленности;
3. Результативность — алгоритм гарантированно приводит к результату за конечное число шагов;
4. Массовость — алгоритм применяется к множеству различных данных одного типа;
5. Конечность — выполнение алгоритма завершается за конечное время.

Рассмотрены основные виды алгоритмов: линейные, разветвляющиеся, циклические, рекурсивные и параллельные. Особое внимание было уделено алгоритмам, которые имеют большую практическую значимость в программировании и используются для решения задач.

В рамках выполнения практических заданий были реализованы следующие алгоритмы:

1. **Числа Фибоначчи**  
Разработаны и проанализированы четыре подхода к вычислению чисел Фибоначчи:

1. Рекурсивный алгоритм. Основное преимущество этого подхода — простота реализации и читаемость кода. Однако его недостаток заключается в низкой производительности из-за экспоненциальной сложности.
2. Итеративный алгоритм. Это наиболее оптимальный алгоритм, обладающий линейной сложностью. Он обеспечивает минимальное использование памяти и быстрый расчёт.
3. Использование массива. Данный алгоритм удобен, если требуется сохранить промежуточные результаты вычислений, но он требует дополнительной памяти.
4. Алгоритм с использованием формулы Бине. Позволяет мгновенно вычислить число Фибоначчи на любой позиции, но имеет ограничения из-за возможной потери точности при работе с большими числами.

На основе анализа этих алгоритмов сделан вывод, что итеративный подход является наиболее универсальным и эффективным для большинства практических задач.

**2. Алгоритм Хаффмана**

Изучение алгоритма Хаффмана для кодирования и декодирования строк продемонстрировало важность подходов к сжатию данных. В ходе практики был реализован алгоритм, который позволил получить оптимальные беспрефиксные коды для символов строки. Это позволило добиться значительного уменьшения объёма данных, что особенно актуально в задачах, связанных с передачей и хранением информации.

Особое внимание было уделено процессу построения бинарного дерева, на основе которого формируется кодирование символов. В результате выполнения задачи стало ясно, что алгоритм Хаффмана обеспечивает минимизацию избыточности данных и является одним из самых эффективных методов сжатия без потерь.

Практическая часть позволила: закрепить теоретические знания о принципах работы алгоритмов, освоить методы анализа алгоритмов, их недостатки и достоинства и области их применения. Таким образом, прохождение практики дало ценный опыт, который послужит основой для дальнейшего профессионального развития.

# Список источников

1. Что такое алгоритмы программирования: основы и применение [Электронный ресурс] / – Режим доступа: https://itproger.com/news/chto-takoe-algoritmi-programmirovaniya-osnovi-i-primenenie

2. АЛГОРИТМЫ, СТРУКТУРЫ ДАННЫХ И ПРОГРАММИРОВАНИЕ [Электронный ресурс] / – Режим доступа: https://elib.kstu.kz/fulltext/Uchebnye%20posobiya/ITB/2018/Solodovnikova.pdf

3. УП.02 - Алгоритмы для вычисления ряда Фибоначчи [Электронный ре-сурс] / – Режим доступа: https://it.vshp.online/#/pages/up02/up02\_fibonacci

4. УП.02 - Алгоритмы Хаффмана для кодирования и декодирования дан-ных [Электронный ресурс] / – Режим доступа: <https://it.vshp.online/#/pages/up02/up02_huffman>

# Приложение 1. QR код.



[**https://github.com/qweasads/algorithms\_practicum**](https://github.com/qweasads/algorithms_practicum)

# Приложение 2. Вычисление n-го числа Фибоначчи с использованием рекурсивного алгоритма

import time  
  
def fib(n):  
 if n == 0:  
 return 0  
 elif n == 1:  
 return 1  
 return fib(n - 1) + fib(n - 2)  
  
def fib\_execution\_timer():  
 test\_values = [5, 10, 15, 20, 24]  
 for n in test\_values:  
 start\_time = time.time()  
 result = fib(n)  
 end\_time = time.time()  
 execution\_time = (end\_time - start\_time) \* 1000  
 print(f"fib({n}) = {result}, время выполнения: {execution\_time:.2f} мс")  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 fib\_execution\_timer()

# Приложение 3. Вычисление n-го числа Фибоначчи с использованием цикла

import time  
  
def fib(n):  
 if n == 0:  
 return 0  
 elif n == 1:  
 return 1  
  
 a, b = 0, 1  
 for \_ in range(2, n + 1):  
 a, b = b, a + b  
 return b  
  
def fib\_execution\_timer():  
 test\_values = [5, 10, 15, 20, 32]  
 for n in test\_values:  
 start\_time = time.time()  
 result = fib(n)  
 end\_time = time.time()  
 execution\_time = (end\_time - start\_time) \* 1000  
 print(f"fib({n}) = {result}, время выполнения: {execution\_time:.2f} мс")  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 fib\_execution\_timer()

# Приложение 4. Вычисление n-го числа Фибоначчи с записью числового ряда в массив

def fib(n):  
 if n == 0:  
 return [0]  
 elif n == 1:  
 return [0, 1]  
  
 fib\_sequence = [0, 1]  
 for i in range(2, n + 1):  
 fib\_sequence.append(fib\_sequence[i - 1] + fib\_sequence[i - 2])  
 return fib\_sequence  
  
def fib\_result():  
 test\_values = [8]  
 for n in test\_values:  
 result = fib(n)  
 print(f"fib({n}) = {result}")  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 fib\_result()

# Приложение 5. Вычисление n-го числа Фибоначчи при помощи формулы Бине

import math  
  
def fib(n):  
 sqrt\_5 = math.sqrt(5)  
 phi = (1 + sqrt\_5) / 2  
 psi = (1 - sqrt\_5) / 2  
  
 fib\_n = (phi\*\*n - psi\*\*n) / sqrt\_5  
 return round(fib\_n)  
  
def binet\_results():  
 test\_values = [32]  
 for n in test\_values:  
 result = fib(n)  
 print(f"fib({n}) = {result}")  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 binet\_results()

# Приложение 6. Определение четности n-го большого числа Фибоначчи

def fib\_eo(n):  
  
 if n == 0:  
 return "even"  
 elif n == 1:  
 return "odd"  
  
 a, b = 0, 1  
  
 for \_ in range(2, n + 1):  
 a, b = b, (a + b) % 10  
  
 return "even" if b % 2 == 0 else "odd"  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 n = int(input("Введите число n: "))  
 print(f"Число ({n}) является: {fib\_eo(n)}")

# Приложение 7. Кодирование строки по алгоритму Хаффмана

from collections import Counter  
import heapq  
  
class Node:  
 def \_\_init\_\_(self, char, freq):  
 self.char = char  
 self.freq = freq  
 self.left = None  
 self.right = None  
  
 def \_\_lt\_\_(self, other):  
 return self.freq < other.freq  
  
def build\_huffman\_tree(frequencies):  
 heap = [Node(char, freq) for char, freq in frequencies.items()]  
 heapq.heapify(heap)  
  
 while len(heap) > 1:  
 left = heapq.heappop(heap)  
 right = heapq.heappop(heap)  
 merged = Node(None, left.freq + right.freq)  
 merged.left = left  
 merged.right = right  
 heapq.heappush(heap, merged)  
  
 return heap[0]  
  
def build\_codes(node, current\_code="", codes={}):  
 if node is None:  
 return  
 if node.char is not None:  
 codes[node.char] = current\_code  
 build\_codes(node.left, current\_code + "0", codes)  
 build\_codes(node.right, current\_code + "1", codes)  
 return codes  
  
def huffman\_encode\_string(input\_string):  
 frequencies = Counter(input\_string)  
 root = build\_huffman\_tree(frequencies)  
 codes = build\_codes(root)  
 encoded\_string = "".join(codes[char] for char in input\_string)  
 return codes, encoded\_string  
  
def huffman\_decode(codes, encoded\_string):  
 code\_to\_symbol = {code: symbol for symbol, code in codes.items()}  
 decoded\_string, buffer = "", ""  
 for bit in encoded\_string:  
 buffer += bit  
 if buffer in code\_to\_symbol:  
 decoded\_string += code\_to\_symbol[buffer]  
 buffer = ""  
 return decoded\_string  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 input\_string = "Errare humanum est."  
 codes, encoded\_string = huffman\_encode\_string(input\_string)  
  
 print(f"{len(codes)} {len(encoded\_string)}")  
 for char, code in sorted(codes.items()):  
 print(f"'{char}': {code}")  
 print(encoded\_string)  
  
  
 decoded\_string = huffman\_decode(codes, encoded\_string)  
 print(f"Decoded string: {decoded\_string}")

# Приложение 8. Декодирование строки по алгоритму Хаффмана

def huffman\_decode(codes, encoded\_string):  
 code\_to\_symbol = {code: symbol for symbol, code in codes.items()}  
 decoded\_string, buffer = "", ""  
 for bit in encoded\_string:  
 buffer += bit  
 if buffer in code\_to\_symbol:  
 decoded\_string += code\_to\_symbol[buffer]  
 buffer = ""  
 return decoded\_string  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 symbol\_count = 12  
 encoded\_size = 60  
 codes = {  
 ' ': '1011',  
 '.': '1110',  
 'D': '1000',  
 'c': '000',  
 'd': '001',  
 'e': '1001',  
 'i': '010',  
 'm': '1100',  
 'n': '1010',  
 'o': '1111',  
 's': '011',  
 'u': '1101'  
 }  
 encoded\_string = "100011110001001101000111111011001010011000010110011010111110"  
 decoded\_string = huffman\_decode(codes, encoded\_string)  
 print(f"Decoded string: {decoded\_string}")

# Приложение 9. Антиплагиат

