

**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)* |

**Содержание**

[**Введение** 2](#_Toc994410534)

[Основные свойства алгоритмов 3](#_Toc1302703163)

[Основные виды алгоритмов 4](#_Toc112734698)

[1. Алгоритм для вычисления чисел Фибоначчи 5](#_Toc888134216)

[1.1. Определение и теоретические основы 6](#_Toc51449246)

[1.2. Свойства чисел Фибоначчи 6](#_Toc822854387)

[1.3. Рекурсивный способ вычисления n-го числа Фибоначчи 6](#_Toc1205484788)

[1.5. Вычисление n-го числа Фибоначчи с записью числового ряда в массив 11](#_Toc451838676)

[1.6. Вычисление n-го числа Фибоначчи при помощи формулы Бине 12](#_Toc108380754)

[1.7. Определение четности n - го большого числа Фибоначчи 14](#_Toc1550699521)

[2. Алгоритмы Хаффмана 15](#_Toc1716545557)

[2.1. Кодирование строки по алгоритму Хаффмана 16](#_Toc1330221403)

[2.2. Декодирование строки по алгоритму Хаффмана 18](#_Toc1578047570)

[Заключение 18](#_Toc794909216)

[Список источников 19](#_Toc1976831012)

[Приложение 1 .Декодирование строки по алгоритму Хаффмана 20](#_Toc266891747)

[Приложение 2. Декодирование строки по алгоритму Хаффмана 22](#_Toc782552591)

[Приложение 3. Антиплагиат 23](#_Toc2064193315)

[Приложение 4. GitHub 23](#_Toc652177408)

# **Введение**

Алгоритмы – это набор четко определенных инструкций или шагов, которые выполняются в определенном порядке для решения конкретной задачи или достижения определенной цели.[1]

**Основные свойства алгоритмов**

1. Детерминированность:

Алгоритм должен быть четким и однозначным. Каждый шаг алгоритма должен быть точно определен, не оставляя места для неоднозначности. Это означает, что на каждом этапе выполнения алгоритма результат должен быть однозначно предсказуемым.

1. Конечность:

Алгоритм должен завершаться после конечного числа шагов. Если алгоритм не имеет конца или зацикливается, он считается некорректным.

1. Реализуемость:

Алгоритм должен быть осуществим на практике, то есть его шаги должны быть выполнимы с использованием ограниченных ресурсов (например, времени и памяти).

1. Эффективность:

Эффективность алгоритма оценивается по времени и памяти, которые он использует. Чем меньше времени и памяти требуется алгоритму для выполнения, тем эффективнее он считается. Это свойство важно для задач с большими объемами данных или сложными вычислениями.

1. Гибкость:

Хороший алгоритм должен быть универсальным и применимым в различных контекстах. Это свойство подразумевает возможность его адаптации под новые задачи или входные данные с минимальными изменениями.

1. Массовость:

Алгоритм должен быть пригодным для решения широкого круга задач, а не только для специфических случаев. Это свойство предполагает, что алгоритм работает на всех входных данных, для которых он предназначен.

**Основные виды алгоритмов**

1. Линейные:

Линейные алгоритмы — это алгоритмы, где каждый шаг выполняется последовательно, один за другим, без повторений или ветвлений. Структура таких алгоритмов имеет вид простой последовательности действий

1. Циклические:

Циклические алгоритмы содержат цикл, который повторяет определенные шаги несколько раз до тех пор, пока не будет выполнено определенное условие. Это позволяет выполнять задачи, которые требуют повторяющихся действий.

1. Разветвляющиеся:

Разветвляющиеся алгоритмы включают конструкции, где выполнение зависит от условий. Это означает, что на основе проверки некоторого условия алгоритм может выбрать один из нескольких путей выполнения.

1. Параллельные:

Параллельные алгоритмы предназначены для выполнения нескольких операций одновременно, используя несколько процессоров или ядер. Эти алгоритмы могут значительно ускорить выполнение задач, требующих большого времени на обработку данных, за счет распределения вычислений между несколькими потоками.

1. Рекурсивные:

Рекурсивные алгоритмы используют принцип самостоятельного вызова, когда алгоритм вызывает сам себя для решения более мелких подзадач. Рекурсия используется, когда задача может быть разделена на одинаковые подзадачи, которые решаются аналогичным образом.

# **1. Алгоритм для вычисления чисел Фибоначчи**

## **1.1. Определение и теоретические основы**

Числа Фибоначчи представляют собой последовательность чисел, где каждое следующее число является суммой двух предыдущих. Формальное определение[2].

Последовательность чисел Фибоначчи имеет несколько важных свойств и используется в различных областях науки и техники, включая биологию, криптографию и вычислительные задачи.

**1.2. Свойства чисел Фибоначчи**

Числа Фибоначчи обладают рядом интересных математических свойств:

1. Золотое сечение: отношение двух последовательных чисел Фибоначчи стремится к числу, известному как золотое сечение φ ≈ 1.618, которое широко используется в искусстве и архитектуре.

2. Печать чисел через квадрат матрицы: числа Фибоначчи можно выразить через возведение матрицы в степень, что позволяет существенно ускорить вычисления.

3. Сумма чисел Фибоначчи: Сумма первых чисел Фибоначчи выражается через формулу[2].

## **1.3. Рекурсивный способ вычисления n-го числа Фибоначчи**

Приведённый код реализует классический рекурсивный алгоритм для вычисления чисел Фибоначчи. Основная идея алгоритма заключается в том, что каждое число Фибоначчи — это сумма двух предыдущих чисел в последовательности.

Алгоритм:

1. Если n = 0, возвращается 0.
2. Если n = 1, возвращается 1.
3. Для всех других значений n функция вызывает саму себя для n−1 и n−2, после чего возвращает их сумму.

Пример кода:

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()

Объяснение кода:

1. Импорт библиотеки time:

Первая строка импортирует модуль time, который используется для измерения времени. Модуль предоставляет функцию time.time(), которая возвращает текущее время в секундах с момента эпохи (1 января 1970 года). Мы будем использовать его для замера времени выполнения алгоритма.

1. Функция fib(n):

Это рекурсивная реализация вычисления n-го числа Фибоначчи.

Числа Фибоначчи — это последовательность чисел, где каждое последующее число равно сумме двух предыдущих:

F(0)=0

F(1)=1

F(n)=F(n−1)+F(n−2) для n>11

В данном случае:

Если n=0, возвращается 0, так как первое число в последовательности Фибоначчи — это 0.

Если n=1, возвращается 1, так как второе число — это 1.

Для других значений n, функция вызывает себя рекурсивно для вычисления F(n−1) и F(n−2) и возвращает их сумму. Это обеспечивает нахождение числа Фибоначчи для любого n.

1. Функция fib\_execution\_timer():

Эта функция используется для замера времени работы функции fib(n) для нескольких значений n (5, 10, 15, 20, 24) и выводит результат вычисления и время, затраченное на вычисление.

Что происходит в функции:

test\_values = [5, 10, 15, 20, 24]: создается список значений nnn, для которых будем вычислять числа Фибоначчи.

for n in test\_values:: цикл перебирает каждое значение n из списка test\_values.

start\_time = time.time(): фиксируется текущее время в секундах перед вызовом функции fib(n).

result = fib(n): вызывается функция fib(n) для текущего значения n, и результат сохраняется в переменную result.

end\_time = time.time(): фиксируется текущее время в секундах после выполнения функции fib(n).

execution\_time = (end\_time - start\_time) \* 1000: вычисляется время выполнения в миллисекундах (разница между временем начала и завершения работы функции).

print(f"fib({n}) = {result}, время выполнения: {execution\_time:.2f} мс"): выводится результат выполнения функции, а также время, затраченное на вычисление числа Фибоначчи для текущего n.

1. Основной блок:

Это конструкция Python, которая проверяет, является ли данный файл основным (то есть, был ли он запущен напрямую, а не импортирован как модуль). Если это основной файл, выполняется функция fib\_execution\_timer(), которая запускает измерение времени выполнения для различных значений n.

Итог:

1. Когда скрипт запускается, он создает список значений n для которых нужно вычислить число Фибоначчи.
2. Для каждого значения n выполняется рекурсивное вычисление числа Фибоначчи.
3. Для каждого вычисления фиксируется время начала и конца, и затем выводится:

* Результат вычисления числа Фибоначчи.
* Время выполнения этого вычисления в миллисекундах.

1.4. Итерационный способ вычисления n-го числа Фибоначчи

Приведённый код реализует итерационный алгоритм для вычисления чисел Фибоначчи. В отличие от рекурсивного подхода, итерационный способ использует цикл, что делает его более эффективным в плане производительности, особенно для больших значений n. В этом методе числа Фибоначчи вычисляются последовательно, начиная с базовых значений (0 и 1), и на каждом шаге результат предыдущих двух чисел используется для вычисления следующего.

Пример кода:

import time  
  
def fib(n):  
 if n == 1 or n == 2:  
 return 1  
  
  
 a, b = 1, 1  
  
  
 for \_ in range(3, n + 1):  
 a, b = b, a + b  
  
 return b  
  
  
def measure\_time():  
 test\_values = [5, 10, 15, 20, 30]  
  
 for n in test\_values:  
 start\_time = time.time()  
 result = fib(n)  
 end\_time = time.time()  
  
 elapsed\_time = (end\_time - start\_time) \* 1000  
 print(f"fib({n}) = {result}, время выполнения: {elapsed\_time:.3f} миллисекунд")  
  
measure\_time()

Объяснение кода:

Функция fib(n):

Если n=1 или n=2, то возвращаем 1, так как это базовые значения в последовательности Фибоначчи.

Для значений n > 2 алгоритм использует цикл, чтобы последовательно вычислить числа Фибоначчи. Начальные значения a=1 и b=1 соответствуют первым двум числам Фибоначчи.

В цикле происходит обновление значений a и b: новое значение a получает значение b, а новое значение b — сумму a и b.

Цикл продолжается до тех пор, пока не будет вычислено нужное n-е число Фибоначчи.

В конце функция возвращает значение b, которое и является искомым числом Фибоначчи для заданного n.

Функция measure\_time():

Эта функция выполняет несколько тестов на вычисление чисел Фибоначчи для разных значений n. Для каждого значения n она замеряет время выполнения функции fib(n).

Время выполнения замеряется с использованием функции time.time(), которая возвращает время в секундах с момента начала эпохи (обычно 1 января 1970 года).

Результат времени выполнения выводится в миллисекундах с точностью до трёх знаков после запятой.

Преимущества итерационного подхода:

Итерационный алгоритм гораздо более эффективен по сравнению с рекурсивным, так как избегает повторных вычислений и накладных расходов на вызовы функции.

Время выполнения алгоритма линейно зависит от n, что делает его подходящим для вычислений с большими значениями n.

Заключение: Итерационный метод позволяет существенно повысить производительность при вычислении чисел Фибоначчи, особенно когда необходимо вычислить значения для больших n.

1.5. Вычисление n-го числа Фибоначчи с записью числового ряда в массив

В данной части работы используется метод, который позволяет вычислять числа Фибоначчи и сохранять их в массив для дальнейшего использования.

Пример кода:

def fib(n):  
 fib\_array = [0, 1]  
 for i in range(2, n + 1):  
 fib\_array.append(fib\_array[i - 1] + fib\_array[i - 2])  
  
 print(fib\_array)  
  
fib(8)

Пояснение к коду:

1. Инициализация списка fib\_array:  
В начале создаём список fib\_array, в котором будут храниться все числа Фибоначчи от F(0) до F(n). Это означает, что *F(0)=F(1)=1*.

2. Цикл for:  
Далее начинается цикл, который проходит с индекса 2 до значения n.

На каждом шаге цикла вычисляется новое число Фибоначчи как сумма двух предыдущих чисел, и это число добавляется в массив.

Например, на первом шаге цикла (для i=2) вычисляется: F(2) = F(1) + F(0) = 1 + 0 = 1

Число 1 добавляется в список, и так продолжается до достижения нужного значения n.

3.Вывод результатов:

После завершения цикла список fib\_array будет содержать все числа Фибоначчи от F(0) до F(n). Мы выводим этот список с помощью команды print.

4. Пример вызова функции:  
**В примере код вызывает функцию для вычисления чисел Фибоначчи до 8-го числа включительно.**

1.6. Вычисление n-го числа Фибоначчи при помощи формулы Бине  
Формула Бине (или формула явного выражения для чисел Фибоначчи) позволяет вычислить n-е число Фибоначчи без использования рекурсии или итераций. Она была выведена с использованием математических свойств чисел Фибоначчи и связи с золотым сечением.

Пример кода:

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()

Объяснение кода:

Подключаем библиотеку math, которая предоставляет функции для математических операций, таких как вычисление квадратного корня.

1. sqrt\_5 = math.sqrt(5) — вычисляем квадратный корень из 5. Это значение используется как часть формулы Бине для нормировки.
2. phi = (1 + sqrt\_5) / 2 — вычисляем золотое сечение ϕ. Это значение примерно равно 1.6180339887.
3. psi = (1 - sqrt\_5) / 2 — вычисляем сопряжённое значение золотого сечения ψ. Это значение примерно равно -0.6180339887.
4. fib\_n = (phi\*\*n - psi\*\*n) / sqrt\_5 — здесь происходит вычисление числа Фибоначчи по формуле Бине. Мы возводим ϕ и ψ в степень n, вычитаем их и делим на √ 5.
5. Return round(fib\_n) — округляем результат до ближайшего целого. Это необходимо, так как формула может давать число с плавающей запятой, а числа Фибоначчи всегда целые.

В функции def binet\_results():

Мы создаём список test\_values, содержащий только одно значение n=32, для которого будет вычислено число Фибоначчи.

В цикле for вычисляем значение F(n) с помощью функции fib(n) и выводим результат.

1.7. Определение четности n - го большого числа Фибоначчи

В данном коде определяется четность n-го числа Фибоначчи по его последней цифре. Числа Фибоначчи образуют последовательность, где каждое следующее число является суммой двух предыдущих. Например, первые несколько чисел Фибоначчи: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 и так далее.

Пример кода:

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)}")

Объяснение действий в коде:

1. Обработка базовых случаев (n = 0 и n = 1):

* Если n=0, то выводится, что число четное, так как F0=0 — четное число.
* Если n=1, то выводится, что число нечетное, так как F1=1 — нечетное число.

2. Инициализация переменных:

Переменные aaa и bbb инициализируются значениями 0 и 1 соответственно, что соответствует первым двум числам последовательности Фибоначчи ( F0 и F1 ).

3. Цикл для вычисления n-го числа Фибоначчи:

В цикле происходит вычисление чисел Фибоначчи до n-го. Однако, чтобы оптимизировать память, мы не храним все числа, а только последнюю цифру каждого числа, используя остаток от деления на 10: (a+b)%10.

4. Возвращение результата:

Когда цикл завершен, переменная b будет хранить последнюю цифру n-го числа Фибоначчи. Мы проверяем, является ли эта цифра четной или нечетной, и возвращаем соответствующий результат.

# 2. Алгоритмы Хаффмана

Алгоритм Хаффмана **— это** эффективный **метод** сжижения данных, который используется для представления символов в виде переменной длины кодов.Основной идеей алгоритма является использование более коротких кодов для более часто встречающихся символов и более длинных кодов для реже встречающихся символов.[3]

2.1. Кодирование строки по алгоритму Хаффмана

Кодирование Хаффмана — это процесс преобразования строки (или другого набора данных) в последовательность битов с помощью кодов переменной длины. Символам, которые появляются часто, присваиваются короткие коды, а символам, которые встречаются реже, присваиваются более длинные коды.[4]

Объяснение кода [См.Приложение 1.]:

1. Ввод строки:

Программа начинается с ввода строки, которую необходимо закодировать с помощью алгоритма Хаффмана. В примере это строка "Errare humanum est."

1. Подсчет частоты символов:

Для начала программа подсчитывает, как часто каждый символ встречается в строке. Это делается с помощью Counter из модуля collections. Этот объект создает словарь, где ключами являются символы строки, а значениями — их частоты.

1. Построение дерева Хаффмана:

* Далее строится дерево Хаффмана. Каждый символ со своей частотой преобразуется в объект класса Node, который представляет собой вершину дерева. Эти вершины добавляются в приоритетную очередь (используя heapq для создания минимальной кучи).
* Пока в куче больше одного элемента, программа извлекает два узла с наименьшими частотами (это будут наименее часто встречающиеся символы) и объединяет их в новый узел. Новый узел получает суммарную частоту этих двух символов, а его дети — это объединенные узлы. Этот процесс продолжается до тех пор, пока в куче не останется только один узел, который будет корнем дерева Хаффмана.

1. Построение кодов для символов:

* После того как дерево построено, необходимо сгенерировать коды для каждого символа. Каждый путь от корня дерева к листу дерева (символу) представляет собой уникальный код для этого символа.
* В процессе обхода дерева по пути от корня к листьям, если мы двигаемся влево — добавляем к коду "0", если вправо — добавляем "1". Так получается уникальный код для каждого символа.

1. Кодирование строки:

После построения кодов для всех символов, программа использует эти коды для кодирования исходной строки. Каждый символ в строке заменяется на его соответствующий код из дерева Хаффмана.

1. Вывод кодов и закодированной строки:

* Программа выводит количество уникальных кодов и длину закодированной строки.
* Затем она выводит список всех символов с их кодами в отсортированном виде (по символам).
* В конце выводится сама закодированная строка, представляющая собой последовательность битов.

1. Декодирование строки:

* После того как строка закодирована, программа декодирует закодированную строку обратно в исходный вид.
* Для декодирования создается словарь, который сопоставляет кодам символы. Программа поочередно читает биты закодированной строки, собирает код, и если он совпадает с кодом в словаре, добавляет соответствующий символ в декодированную строку. Этот процесс повторяется до тех пор, пока не будет расшифрована вся строка.

1. Вывод декодированной строки:

После успешного декодирования программа выводит восстановленную строку, которая должна совпасть с исходной строкой, если алгоритм был выполнен правильно. [См.Приложение 2]

## 2.2. Декодирование строки по алгоритму Хаффмана

Декодирование строки по алгоритму Хаффмана — это процесс восстановления исходного текста (или данных), который был сжато с использованием кодирования Хаффмана.

Объяснение кода [См.Приложение 2.]:

1. Описание функции huffman\_decode:

Функция huffman\_decode выполняет декодирование закодированной строки с использованием заданных кодов Хаффмана.

Что делает функция:

Создание словаря соответствия кодов символам:  
Сначала создаётся обратный словарь code\_to\_symbol, где ключами будут двоичные коды, а значениями — соответствующие символы. Это позволит быстро найти символ, когда мы столкнёмся с кодом, соответствующим этому символу.

1. **Декодирование строки**: Декодирование строки:  
   Далее мы начинаем читать закодированную строку (encoded\_string) по одному биту за раз и строим из них код. Когда собранный код из буфера совпадает с одним из кодов в code\_to\_symbol, это означает, что мы нашли символ. Этот символ добавляется в результат, а буфер очищается для начала сбора следующего кода.
2. Возвращаем результат:  
   Когда весь код пройден, результат сохраняется в переменную decoded\_string, которая и возвращается из функции.
3. После декодирования строки функция выводит результат в консоль.[См.Приложение 2]

Заключение

В ходе учебной практики по алгоритмам Хаффмана и числам Фибоначчи была проведена работа, направленная на изучение теоретических основ и практическое применение этих важных математических понятий в информатике.

Алгоритм Хаффмана, используемый для сжатия данных, показал свою эффективность в оптимизации представления информации, сокращая объем данных без потери качества. Я изучил основные принципы работы алгоритма, рассмотрел его реализацию и применил на практике. Результаты работы показали, что алгоритм Хаффмана позволяет значительно уменьшить размер данных.

Числа Фибоначчи, в свою очередь, продемонстрировали свою универсальность и важность как в теоретической математике, так и в практических задачах, таких как динамическое программирование и оптимизация алгоритмов. Я изучил различные способы генерации чисел Фибоначчи, их свойства, а также рассмотрел применения в области вычислений, где числа Фибоначчи могут быть использованы для эффективного решения задач.

# Список источников

1. Что такое алгоритмы программирования [Электронный ресурс] / – Режим доступа: <https://habr.com/ru/articles/504008/>
2. Числа Фибоначчи [Электронный ресурс] / – Режим доступа: <https://ru.wikipedia.org/wiki/Числа_Фибоначчи>
3. Алгоритм Хаффмана [Электронный ресурс] / – Режим доступа: <https://habr.com/ru/companies/otus/articles/497566/>
4. Кодирование строки по алгоритму Хаффмана [Электронный ресурс] / – Режим доступа:

<https://www.techiedelight.com/ru/huffman-coding/>

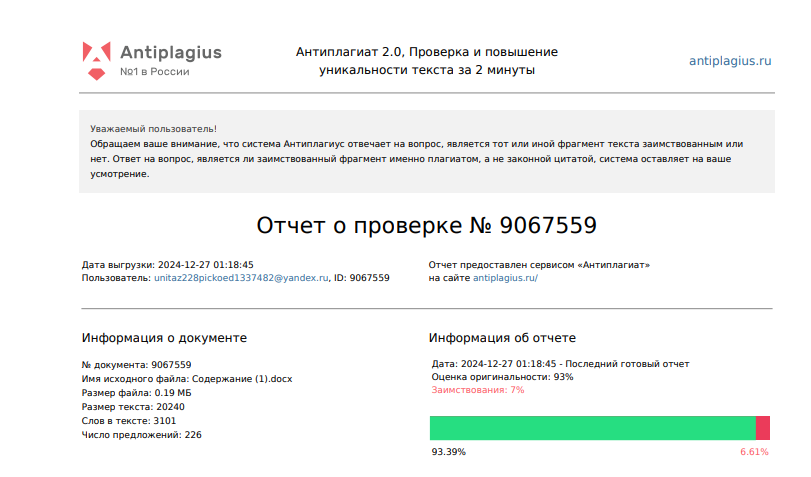
## Приложение 1 .Декодирование строки по алгоритму Хаффмана

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 2. Декодирование строки по алгоритму Хаффмана

def huffman\_decode(symbol\_count, encoded\_size, 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  
  
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(symbol\_count, encoded\_size, codes, encoded\_string)  
print(decoded\_string)

Приложение 3. Антиплагиат



## Приложение 4. GitHub



Ссылка на репозиторий github: https://github.com/obdrystish/algorithms\_practicum.git