**Частное учреждение профессионального образования**

**«Высшая школа предпринимательства»**

**(ЧУПО «ВШП»)**

# ДНЕВНИК ПРАКТИКИ

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)*

# Содержание:

## Введение.

Учебная практика по курсу «Алгоритмы и структуры данных» проводилась с 01 июля 2024 года по 27 декабря 2024 года на кафедре информатики и вычислительной техники. Основной целью было укрепление навыков разработки и анализа алгоритмов, оценивание вычислительной сложности кода, а также развитие основ групповой разработки и документирования программных проектов.

Основные задачи, которые ставились перед студентами, включали:

- Изучение теории анализа алгоритмов, таких как Big O, включая оценку временной и пространственной сложности.
- Изучение, реализация и тестирование алгоритмов для вычисления чисел Фибоначчи, охватывая:
- Рекурсивные и итеративные подходы.
- Использование формулы Бине.
- Мемоизацию и другие методы оптимизации вычислений.
- Освоение алгоритма Хаффмана для эффективного кодирования строк:
- Понимание построения дерева Хаффмана и генерации кодов.
- Создание функций кодирования и декодирования для данных текстов.
- Изучение различных алгоритмов сортировки, включая: пузырьковую сортировку, шейкерную, «расчёской», выбором, вставками, быструю, слиянием, пирамидальную и поразрядную сортировки.
- Сохранение результатов в публичном репозитории (GitHub) и подготовка отчёта по практике с анализом сложности алгоритмов.

В общем, перечень выполненных задач включал:

- Поиск и изучение теоретических и справочных материалов, а также нужных разделов учебников и методических рекомендаций.
- Практическую реализацию алгоритмов на языке Python
- Создание программных модулей для вычисления чисел Фибоначчи и реализацию кодировщика/декодировщика с использованием алгоритма Хаффмана.
- Документирование результатов измерений времени выполнения и оценки сложности алгоритмов.
- Подготовку итогового отчёта, оформленного в соответствии с требованиями практики.

## Основная часть

Вычислительная сложность

В этот день изучение было сосредоточено на вычислительной сложности. Во время практики я разобрался с основными концепциями асимптотической оценки алгоритмов: Big O, Ω, Θ. Особое внимание уделялось учёту временных и пространственных характеристик, классификациям сложности (линейная, квадратичная, экспоненциальная и другие) и важности их в масштабных проектах. Я провёл несколько простых тестов, которые показали, как с увеличением размера входных данных изменяется время выполнения операций, таких как суммирование элементов массива или вложенные циклы. Это прояснило, что точная оценка сложности гарантирует эффективную работу алгоритмов даже при значительном увеличении объёмов данных.

### Алгоритмы сортировки

Последнее занятие посвящалось алгоритмам сортировки. Я изучил множество классических методов: пузырьковую сортировку, шейкерную, «расчёской», выбором, вставками, быструю сортировку, слиянием и пирамидальную. Каждому из них я уделил внимание с точки зрения основной идеи, теоретической оценки сложности и тестирования на различных наборах данных. Пузырьковая сортировка показала низкую эффективность при работе с большими массивами (O(n^2)). Быстрая сортировка и сортировка слиянием продемонстрировали среднюю сложность O(n log n) и оказались быстрее на больших объёмах данных. Методы вставок и выбором полезны для небольших наборов данных или в качестве частей комбинированных алгоритмов. По результатам я осознал, что выбор метода сортировки диктуется не только общей сложностью, но и

спецификой данных (размер, степень упорядоченности, ограничения на память и т.д.).

Так, каждый день в дневнике отражает изученные темы и достигнутые результаты. От асимптотической сложности до конкретных реализаций алгоритмов (Фибоначчи, Хаффман, сортировки), я приобрёл системное понимание важности правильного выбора и оценки алгоритмов, что не только оптимизирует ресурсы, но и углубляет моё понимание логики построения эффективных решений в современной разработке.

## Алгоритмы для вычисления чисел Фибоначчи.

В рамках учебной практики я изучил алгоритмы для вычисления чисел Фибоначчи и провел их сравнительный анализ с точки зрения эффективности. Вначале я рассмотрел наивный рекурсивный метод, который демонстрирует экспоненциальный рост количества вызовов и становится неэффективным при больших значениях n. Затем я реализовал итеративный подход, который работает за линейное время и требует минимальных ресурсов памяти. Также я протестировал формулу Бине, обладающую теоретической константной сложностью, однако при больших n возникают погрешности из-за ограниченной точности вычислений с плавающей запятой. В завершение я исследовал метод мемоизации, который позволяет сохранять результаты уже выполненных вызовов и устранять избыточную рекурсию. В итоге я пришел к выводу, что для большинства задач, связанных с числами Фибоначчи, наиболее эффективными являются итеративный и мемоизированный методы.

### Задача 1: Рекурсивный (наивный) способ вычисления n-го числа Фибоначчи

**Постановка задачи:**

Требуется реализовать рекурсивный алгоритм для вычисления n-го числа Фибоначчи F(n) по формуле:

$$F(n)=F(n-1)+F(n-2) \quad F(n)=F(n-1)+F(n-2)$$

с базовыми случаями F(0) = 0 и F(1) = 1. Ограничение: $0 \le n \le 24 \quad 0 \le n \le 24$, так как время выполнения алгоритма растёт экспоненциально.

**Формат входных данных:**

На вход подаётся одно целое число *n* в диапазоне [0..24].

**Формат выходных данных:**

На выходе должно быть возвращено одно целое число, равное F(n).

**Описание решения:**

1. **Базовые случаи:**
   - Если n=0, возвращается 0.
   - Если n=1, возвращается 1.
1. **Рекурсивный случай:**
   - Для n>1*n*>1 число Фибоначчи вычисляется как сумма двух предыдущих чисел:

$$F(n)=F(n-1)+F(n-2)F(n)=F(n-1)+F(n-2)$$

1. **Ограничение:**
   - Из-за экспоненциального роста времени выполнения алгоритм применим только для небольших значений n (до 24).

**Пример работы:**

Для входного значения n=6*n*=6 результат будет:

```
8
```

(так как F(6) = 8).

**Реализация на Python:**

```
python

def fib_recursive(n):

# Базовые случаи

if n <= 1:

    return n

 # Рекурсивный случай
```

```
return fib_recursive(n - 1) + fib_recursive(n - 2)
```

**Объяснение работы кода:**

1. **Базовые случаи:**
   - Если n=0*n*=0 или n=1*n*=1, функция возвращает n*n*, так как F(0) = 0 и F(1) = 1.
1. **Рекурсивный случай:**
   - Для n>1*n*>1 функция вызывает саму себя для вычисления F(n-1) и F(n-2), а затем возвращает их сумму.
1. **Экспоненциальная сложность:**
   - Время выполнения алгоритма растёт экспоненциально (примерно O(2n)*O*(2*n*)), так как каждое число Фибоначчи вычисляется заново на каждом шаге.

**Преимущества решения:**

- **Простота реализации:** Код легко читается и понимается.
- **Наглядность:** Прямая рекурсия точно отражает математическое определение чисел Фибоначчи.

**Недостатки решения:**

- **Низкая эффективность:** Экспоненциальная сложность делает алгоритм непригодным для больших значений n*n*.
- **Повторные вычисления:** Многие значения F(k) вычисляются многократно, что приводит к избыточным операциям.

**Пример использования:**

```
python

print(fib_recursive(6))  # Вывод: 8

print(fib_recursive(10)) # Вывод: 55
```

**Ключевые моменты:**

1. **Базовые случаи:**
   - F(0) = 0 и F(1) = 1 — это условия завершения рекурсии.
1. **Рекурсивный вызов:**
   - F(n) вычисляется как сумма двух предыдущих чисел Фибоначчи.
1. **Ограничение на** n*n*:
   - Из-за экспоненциальной сложности алгоритм работает только для небольших значений n*n*.

Это решение идеально подходит для изучения рекурсии, но для практических задач с большими n*n* рекомендуется использовать итеративные методы или методы с мемоизацией.

**Задача 2: Итеративный способ вычисления n-го числа Фибоначчи**

**Постановка задачи:**
Требуется вычислить n-е число Фибоначчи F(n) с использованием итеративного метода. Считаем, что F(1) = 1 и F(2) = 1. Ограничение: 1≤n≤321≤*n*≤32.

**Формат входных данных:**
На вход подаётся одно целое число n.

**Формат выходных данных:**
На выходе должно быть возвращено одно целое число, равное F(n).

**Описание решения:**

1. **Базовые случаи:**
   - Если n=1*n*=1 или n=2*n*=2, возвращается 1.
1. **Итеративный процесс:**
   - Инициализируются две переменные: a=1*a*=1 (F(1)) и b=1*b*=1 (F(2)).
   - В цикле от 3 до n вычисляется следующее число Фибоначчи:

**result=a+bresult=*a*+*b***

- Переменные обновляются: a*a* принимает значение b*b*, a b*b* — значение resultresult.

1. **Результат:**
   - После завершения цикла возвращается значение b*b*, которое равно F(n).

**Реализация на Python:**

```
python

def fib_loop(n):

  # Базовые случаи

  if n == 1 or n == 2:

return 1

# Инициализация переменных

   a, b = 1, 1

 # Итеративный процесс

  for _ in range(3, n + 1):

 a, b = b, a + b

  # Возвращаем результат     return b
```

**Объяснение работы кода:**

1.

2. **Базовые случаи:**
   - Если n=1*n*=1 или n=2*n*=2, функция возвращает 1, так как F(1) = 1 и F(2) = 1.

1. **Инициализация переменных:**

- Переменные a и b хранят два последних числа Фибоначчи.

1. **Цикл:**

- На каждой итерации вычисляется следующее число Фибоначчи как сумма a и b.
- Переменные a и b обновляются для следующего шага.

1. **Результат:**

- После завершения цикла возвращается значение b, которое равно F(n).

**Преимущества решения:**

- **Эффективность:** Время выполнения алгоритма линейное O(n)$O(n)$, а используемая память константная O(1)$O(1)$.
- **Простота:** Код легко читается и понимается.
- **Универсальность:** Подходит для вычисления чисел Фибоначчи в пределах 1≤n≤321≤$n$≤32.

**Задача 3. Вычисление n-го числа Фибоначчи с записью в массив**

*Дано целое число n (1 <= n <= 40). Требуется вычислить все числа Фибоначчи вплоть до n-го включительно и вернуть результат в виде массива. Считаем, что F(1) = 1, F(2) = 1.*

Формат входных данных:
На вход подаётся одно целое число n в диапазоне [1..40].

Формат выходных данных:
На выходе должен быть возвращён массив (список), где каждый элемент с индексом i соответствует числу Фибоначчи F(i). Например, для n = 8 результат будет: [0, 1, 1, 2, 3, 5, 8, 13, 21]. Нулевой элемент (0) добавлен для удобства индексирования.
Описание решения:

- Создаётся массив fibs длиной n + 1, инициализированный нулями. Это позволяет использовать индексы от 0 до n.
- Устанавливаются начальные значения: fibs[0] = 0 и fibs[1] = 1.
- В цикле от 2 до n массив заполняется по формуле Фибоначчи: fibs[i] = fibs[i - 1] + fibs[i - 2].
- Возвращается заполненный массив.

**Пример работы:**

Для входного значения n = 8 результат будет:

```
[0, 1, 1, 2, 3, 5, 8, 13, 21]
```

**Реализация на Python:**

```
def fib_array(n):
 # Создаём массив длиной n+1, заполненный нулями
 fibs = [0] * (n + 1)
   # Устанавливаем начальные значения
 fibs[0], fibs[1] = 0, 1
  # Заполняем массив числами Фибоначчи
 for i in range(2, n + 1): fibs[i] = fibs[i - 1] + fibs[i - 2]
 # Возвращаем результат
return fibs
```

**Объяснение работы кода:**

1. Массив fibs создаётся длиной n + 1, чтобы индексы совпадали с номерами чисел Фибоначчи.
2. Первые два элемента массива инициализируются значениями 0 и 1, так как F(0) = 0 и F(1) = 1.

3. В цикле от 2 до n каждый элемент массива вычисляется как сумма двух предыдущих элементов.
4. Функция возвращает массив, содержащий последовательность чисел Фибоначчи до n-го включительно.

**Преимущества решения:**

- Простота и наглядность реализации.
- Линейная сложность O(n) по времени и памяти.
- Удобство использования, так как индексы массива соответствуют номерам чисел Фибоначчи.

Это решение эффективно решает задачу вычисления чисел Фибоначчи и может быть использовано в различных приложениях, где требуется работа с этой последовательностью.

**Задача 4. Вычисление n-го числа Фибоначчи по формуле Бине**

Вычислить F(n) по формуле Бине:

F(n) = (phi^n - (1 - phi)^n) / sqrt(5),

где phi = (1 + sqrt(5)) / 2.
Округлять результат. Ограничение: 1 <= n <= 64.

**Постановка задачи:**
Требуется вычислить n-е число Фибоначчи F(n) с использованием формулы Бине. Формула Бине позволяет получить значение F(n) за константное время O(1), но при больших значениях n (например, n > 64) могут возникать погрешности из-за ограниченной точности вычислений с плавающей запятой. Ограничение: 1 ≤ n ≤ 64.

**Формат входных данных:**
На вход подаётся одно целое число n.

**Формат выходных данных:**

На выходе должно быть возвращено одно целое число, равное F(n).

**Описание решения:**

1. Вычисляется значение «золотого сечения»: $\phi=\frac{1+\sqrt{5}}{2}$
2. Вычисляется вспомогательное значение: $\psi=\frac{1-\sqrt{5}}{2}$
3. Подставляются значения в формулу Бине: $F(n)=\frac{\phi^n-\psi^n}{\sqrt{5}}$
4. Результат округляется до ближайшего целого числа с помощью функции round.

**Пример работы:**

Для входного значения n = 10 результат будет:

```
python:

55
```

(так как F(10) = 55).

**Реализация на Python:**

```
import math  def fib_binet(n):

  # Вычисляем "золотое сечение" и вспомогательное значение

phi = (1 + math.sqrt(5)) / 2     psi = (1 - math.sqrt(5)) / 2

# Применяем формулу Бине

fib_n = (phi ** n - psi ** n) / math.sqrt(5)

# Округляем результат до ближайшего целого

 return round(fib_n)
```

**Объяснение работы кода:**

1. **Вычисление констант:**
   - phi — это «золотое сечение», равное 1+5221+5.
   - psi — вспомогательное значение, равное 1−5221−5.
1. **Применение формулы Бине:**

Формула Бине использует возведение в степень и деление для вычисления числа Фибоначчи.

1. **Округление результата:**
   - Поскольку формула Бине возвращает вещественное число, результат округляется до ближайшего целого с помощью функции round.

**Преимущества решения:**

- **Высокая скорость:** Формула Бине позволяет вычислить число Фибоначчи за константное время O(1).
- **Простота реализации:** Код компактен и легко читается.

**Ограничения:**

- **Погрешности:** На больших значениях n (например, n > 64) могут возникать погрешности из-за ограниченной точности вычислений с плавающей запятой.

**Пример использования:**

```
python

print(fib_binet(10))  # Вывод: 55

print(fib_binet(20))  # Вывод: 6765
```

**Ключевые моменты:**

1. **Формула Бине:**
   - Позволяет вычислить число Фибоначчи напрямую, без использования рекурсии или итераций.
1. **Округление:**
   - Результат округляется до ближайшего целого, чтобы получить точное значение F(n).

Это решение идеально подходит для задач, где требуется быстро вычислить число Фибоначчи для небольших значений n (в пределах 1 ≤ n ≤ 64). Для больших значений n рекомендуется использовать итеративные методы или методы с мемоизацией.

**Задача 5: Определение чётности n-го большого числа Фибоначчи**

**Постановка задачи:**
Необходимо определить, является ли n-е число Фибоначчи чётным или нечётным. Значение n может достигать 10^6, поэтому для избежания переполнения используется только последняя цифра числа Фибоначчи на каждом шаге. Считаем, что F(1) = 1 и F(2) = 1.

**Формат входных данных:**
На вход подаётся одно целое число n в диапазоне [1..1000000].

**Формат выходных данных:**
На выходе должна быть возвращена строка "even", если число Фибоначчи чётное, или "odd", если нечётное.

**Описание решения:**

1. Инициализируются две переменные: a = 0 (F(0)) и b = 1 (F(1)).

2. Если n = 0, возвращается "even", так как F(0) = 0.
3. Если n = 1, возвращается "odd", так как F(1) = 1.
4. В цикле от 2 до n вычисляется последняя цифра следующего числа Фибоначчи по формуле: c = (a + b) % 10.
5. Переменные a и b обновляются: a принимает значение b, a b — значение c.
6. После завершения цикла проверяется чётность значения b (последняя цифра F(n)). Если b % 2 == 0, возвращается "even", иначе — "odd".

**Пример работы:**

Для входного значения **n = 3** результат будет:

"even"

(так как F(3) = 2, что является чётным числом).

**Реализация на Python:**

```
def fib_eo(n):
 # Базовые случаи
        if n == 0:
        return "even"
        elif n == 1:
        return "odd"
         # Инициализация переменных
          a, b = 0, 1
```

```
    # Вычисление последней цифры F(n)
    for _ in range(n - 1):
        a, b = b, (a + b) % 10  # Сохраняем только последнюю цифру
    # Определение чётности
    return "even" if b % 2 == 0 else "odd"
```

**Объяснение работы кода:**

1. **Базовые случаи:**
   - Если n = 0, возвращается "even", так как F(0) = 0.
   - Если n = 1, возвращается "odd", так как F(1) = 1.
1. **Инициализация:**
   - Переменные a и b хранят последние цифры двух предыдущих чисел Фибоначчи.
1. **Цикл:**
   - На каждой итерации вычисляется последняя цифра следующего числа Фибоначчи: (a + b) % 10.
   - Переменные a и b обновляются для следующего шага.
1. **Определение чётности:**
- После завершения цикла проверяется чётность последней цифры F(n) и возвращается соответствующий результат.

**Преимущества решения:**

- Эффективность: используется только последняя цифра числа Фибоначчи, что позволяет работать с большими значениями n (до 10^6).
- Простота: код легко читается и понимается.
- Минимальное использование памяти: хранятся только две переменные (a и b).

Это решение идеально подходит для задач, где требуется определить чётность числа Фибоначчи без вычисления самого числа целиком, что особенно важно при работе с большими значениями n.

## Алгоритмы Хаффмана для кодирования и декодирования данных

Алгоритмы Хаффмана стали ключевой темой изучения на данном этапе. Я подробно разобрал принцип построения дерева Хаффмана: сначала выполняется подсчёт частот встречаемости символов в тексте, затем создаётся приоритетная очередь, в которой символы с наименьшими частотами объединяются в новые узлы. В результате формируется бинарное дерево, где левый переход обозначается как «0», а правый — «1». На практике я реализовал функции для преобразования исходного текста в двоичную строку (кодирование) и обратного преобразования (декодирование). Это позволило понять, как обеспечивается беспрефиксность кодов и почему закодированные данные могут быть однозначно восстановлены без использования дополнительных разделителей.

### Практический пример построения дерева Хаффмана

В рамках практического занятия я рассмотрел пример построения дерева Хаффмана для конкретного текста, например, «skibidi». Процесс включал несколько этапов:

1. **Подсчёт частот символов:** Определил, сколько раз каждый символ встречается в тексте.
2. **Формирование списка узлов:** Создал узлы для каждого символа с учётом их частот.
3. **Построение дерева:** Попарно объединял узлы с наименьшими частотами, формируя новые узлы, до тех пор, пока не останется один корень.
4. **Генерация кодов:** Прошёл по дереву, присваивая символам двоичные коды: «0» для левых переходов и «1» — для правых.

В результате часто встречающиеся символы получили короткие коды, а редкие — более длинные. Это наглядно продемонстрировало эффективность алгоритма Хаффмана для сжатия данных.

**Проверка корректности реализации**

Для проверки корректности работы алгоритма я реализовал функции кодирования и декодирования. После преобразования исходного текста в двоичную строку и обратного декодирования исходный текст был полностью восстановлен. Это подтвердило правильность реализации алгоритма и его способность сохранять информацию без потерь.

1.

**Задача 6: Кодирование строки по алгоритму Хаффмана**

**Постановка задачи:**
Закодировать строку с использованием алгоритма Хаффмана.

**Формат входных данных:**
Одна строка с произвольными символами (например, "Errare humanum est.").

**Формат выходных данных:**

1. Количество уникальных символов.
2. Длина закодированной строки в битах.
3. Список кодов символов в формате 'символ': код.
4. Закодированная строка (двоичный поток).

**Описание решения:**

1. **Подсчёт частот символов:**
   Используется словарь для подсчёта частот.

2. **Построение дерева Хаффмана:**

- Создаётся куча из узлов (символ, частота).
- Узлы объединяются в дерево, начиная с наименьших частот.

1. **Генерация кодов символов:**

- Рекурсивно обходится дерево, присваивая коды: `0` для левого перехода, `1` для правого.

1. **Кодирование строки:**
   Каждый символ заменяется на его код, формируя закодированную строку.
2. **Вывод результатов:**

- Количество уникальных символов.
- Длина закодированной строки.
- Список кодов.
- Закодированная строка.

**Пример:**

**Вход:**

```
"Errare humanum est."
```

**Выход:**

```
12 67

' ': 000

'.': 1011

'E': 0110

'a': 1110

'e': 1111

'h': 0111
```

```
'm': 010

'n': 1000

'r': 110

's': 1001

't': 1010

'u': 001

01101101101110110111100001110010101110100000101000011111001101010
11
```

Основные шаги:

1. Подсчитать частоты символов.
2. Построить дерево Хаффмана.
3. Сгенерировать коды символов.
4. Закодировать строку.
5. Вывести результаты.

Основной код (Python):

Приложение 2.

**Задача 7. Декодирование строки по алгоритму Хаффмана**

**Постановка задачи:**

Даны:

1. Количество уникальных символов (uniqueCount).
2. Список строк вида 'символ': код.
3. Двоичная закодированная строка. Требуется восстановить исходную строку.

**Формат входных данных:**

1. Число uniqueCount.
2. uniqueCount строк в формате 'символ': код.
3. Двоичная строка (из 0 и 1).

**Формат выходных данных:**

Раскодированная строка.

**Описание решения:**

1. **Создание обратного словаря:**

- Инвертировать словарь кодов, чтобы ключами были коды, а значениями — символы.

1. **Декодирование строки:**

- Пройти по двоичной строке, накапливая биты в буфере.
- Когда буфер совпадает с одним из ключей в обратном словаре, добавить соответствующий символ в результат и сбросить буфер.

Основной код (Python):

Приложение 3.

Объяснение кода:

- **Входные данные:**

  uniqueCount: Количество уникальных символов.
  code_map: Словарь, где ключи — символы, а значения — их коды.
  encoded_string: Закодированная двоичная строка.

- **Обратный словарь:**

  Создаётся словарь reverse_map, где ключи — коды, а значения — символы.

- **Декодирование:**

  Проходим по двоичной строке, накапливая биты в buffer.
  Когда buffer совпадает с ключом в reverse_map, добавляем соответствующий символ в decoded_string и сбрасываем buffer.

- **Вывод:**

  Раскодированная строка выводится на экран.

## Заключение:

В ходе учебной практики были успешно выполнены все поставленные задачи, что позволило достичь основной цели — углублённого изучения и практического применения ключевых алгоритмов, таких как вычисление чисел Фибоначчи, сортировка данных, а также кодирование и декодирование информации с использованием алгоритма Хаффмана. Результаты подтвердили важность выбора оптимального алгоритма, так как его эффективность напрямую влияет на производительность программного решения.

При реализации различных методов вычисления чисел Фибоначчи (рекурсивный, итеративный, с мемоизацией и по формуле Бине) было показано, что время выполнения может варьироваться в десятки раз. Наивная рекурсия, несмотря на простоту, оказалась неэффективной для больших значений из-за экспоненциальной сложности. Итеративные методы и мемоизация продемонстрировали линейную сложность, что делает их более практичными. Формула Бине, хотя и имеет константную сложность, ограничена точностью вычислений с плавающей запятой.

Алгоритм Хаффмана доказал свою эффективность для сжатия данных. Его ключевое преимущество — беспрефиксность кодов, что позволяет однозначно декодировать информацию без дополнительных разделителей. Практическая реализация кодирования и декодирования текста подтвердила, что теоретические знания успешно применяются на практике.

Полученные результаты подчеркивают важность алгоритмического проектирования в современных технологиях, таких как обработка больших данных, сжатие информации и оптимизация вычислений. Освоение этих алгоритмов развивает не только навыки

программирования, но и умение анализировать и оптимизировать решения, что является ключевым для успешной профессиональной деятельности.

Таким образом, учебная практика достигла своих целей, закрепив теоретические знания и продемонстрировав их практическую значимость. Это подтверждает, что эффективные алгоритмы и структуры данных остаются фундаментом разработки программного обеспечения.

**Список источников:**


1. УП.02 - Алгоритмы Хаффмана для кодирования и декодирования данных [Электронный ресурс] / – Режим доступа: https://it.vshp.online/#/pages/up02/up02_huffman
2. УП.02 - Алгоритмы для вычисления ряда Фибоначчи [Электронный ресурс] / – Режим доступа: https://it.vshp.online/#/pages/up02/up02_fibonacci
3. УП.02 - Алгоритмы сортировки [Электронный ресурс] / – Режим доступа: https://it.vshp.online/#/pages/up02/up02_sort
4. УП.02 - Вычислительная сложность [Электронный ресурс] / – Режим доступа: https://it.vshp.online/#/pages/up02/up02_complexity
5. Обозначение «Большое О» [Электронный ресурс] / – Режим доступа: https://en.wikipedia.org/wiki/Big_O_notation

**Приложение 1.** – QR-код на репозиторий

**Приложение 2. -** Кодирование строки по алгоритму Хаффмана

```
# 2.1. Кодирование строки по алгоритму Хаффмана
import heapq
from collections import Counter


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 huffman_encode(text):
    # 1. Подсчет частоты символов
    frequency = Counter(text)

    # 2. Создание приоритетной очереди (кучи)
    priority_queue = [Node(char, freq) for char, freq in frequency.items()]
    heapq.heapify(priority_queue)

    # 3. Построение дерева Хаффмана
    while len(priority_queue) > 1:
        left = heapq.heappop(priority_queue)
        right = heapq.heappop(priority_queue)
        merged = Node(None, left.freq + right.freq)
```

```
        merged.left = left
        merged.right = right
        heapq.heappush(priority_queue, merged)

    # 4. Генерация кодов Хаффмана
    huffman_codes = {}

    def generate_codes(node, current_code):
        if node is not None:
            if node.char is not None:
                huffman_codes[node.char] = current_code
            generate_codes(node.left, current_code + '0')
            generate_codes(node.right, current_code + '1')

    root = priority_queue[0]
    generate_codes(root, '')

    # 5. Кодирование строки
    encoded_text = ''.join(huffman_codes[char] for char in text)

    # 6. Вывод результатов в требуемом формате
    unique_chars_count = len(frequency)
    encoded_size = len(encoded_text)

    # Первая строка: количество уникальных символов и размер закодированной строки
    print(f"{unique_chars_count} {encoded_size}")

    # Следующие строки: коды символов в формате "'symbol': code"
    for char, code in sorted(huffman_codes.items()):
```

```
        print(f"'{char}': {code}")

    # Последняя строка: закодированная строка
    print(encoded_text)



# Ввод текста
text = "Errare humanum est."
huffman_encode(text)
```

**Приложение 3. -** Декодирование строки по алгоритму Хаффмана

```
# 2.2. Декодирование строки по алгоритму Хаффмана
def huffman_decode():
    header = "12 67"
    code_map = {
        'E': '0000',
        't': '0001',
        'e': '001',
        'u': '010',
        ' ': '011',
        'a': '100',
        's': '1010',
        'n': '1011',
        'r': '110',
        'm': '1110',
        '.': '11110',
        'h': '11111'
    }
    encoded_string =
"000011011010011000101111111010111010010110101110011001101000
00111110"
    reverse_map = {code: char for char, code in code_map.items()}

    decoded_string = ""
    buffer = ""
    for bit in encoded_string:
        buffer += bit
        if buffer in reverse_map:
            decoded_string += reverse_map[buffer]
```

```
            buffer = ""
    print(decoded_string)

huffman_decode()
```