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

В современном мире, где технологическое развитие не имеет смысла, существует множество подходов к решению одной и той же технической задачи. Каждое из них имеет свои преимущества и недостатки, которые необходимо учитывать при выборе. Тем не менее, независимо от вариантов обилия, предпочтение обычно отдается только одному, принятому решению. Умение выбора такого решения является необходимым навыком, который позволяет не только создавать эффективное программное обеспечение, но и минимизировать временные и ресурсные затраты.

Научиться оценивать производительность алгоритмов и находить наиболее рациональные подходы к решению задач — одна из ключевых целей для любого разработчика программного обеспечения. Это особенно актуально в условиях, когда требования к скорости и настройке программ становятся все более строгими.

В рамках учебной практики, проходившей с **8 ноября по 27 декабря**, я сосредоточился на изучении понятия **«вычислительная сложность»**. Этот параметр играет решающую роль при анализе алгоритмов, так как позволяет количественно оценить, эффективно реализовать свою задачу при увеличении объема данных. Чтобы лучше понять теоретическую основу, я изучил и применил вычислительную сложность на примере таких классических алгоритмов, как:

- **алгоритм Хаффмана** — используется для сжатия данных,
- **последовательность Фибоначчи** — математический объект с широким применением в программировании,
- **алгоритмы сортировки**, которые являются фундаментальными для разработки в любой области.

Цель учебной практики:

Развить понимание вычислительной сложности и научиться применять эти знания на практике, используя классические алгоритмы.

Задачи учебной практики:

Для достижения целей я поставил перед собой следующие задачи:

1. Разберемся с понятием «вычислительная сложность» и изучим его теоретическую основу.
2. Исследуются алгоритмы расчета чисел Фибоначчи, включая их особенности и производительность.
3. Освоить метод Хаффмана, его принципы работы и применимость области применения.
4. Рассмотрите алгоритмы сортировки, изучите их особенности и применимость в зависимости от задачи.

Выполненные работы:

В процессе практики мной были выполнены следующие задания:

- Решение задач по вычислению чисел Фибоначчи с использованием различных подходов, таких как рекурсия, итерация и методы оптимизации.
- Изучение и реализация алгоритма Хаффмана, включающего построение дерева и кодирования строк.

Этим задачам удается закрепить теоретические знания о применении, улучшить навыки написания кода и улучшить понимание сложности алгоритмов.

Для решения поставленных задач я использовал язык программирования **Python**, который выбрал по необходимым причинам:

1. **Высокая скорость разработки.** Python предоставляет множество встроенных библиотек и инструментов, упрощающих выполнение сложных программ.
2. **Уверенное вмешательство.** Данный язык является для меня основным вариантом разработки, в котором особое внимание уделяется алгоритмам, а не изучению синтаксиса.

3. **Логический и читаемый синтаксис.** Python обеспечивает простоту и ясность кода, что делает его удобным для анализа и улучшения программ.[3]

Вычислительная сложность

В рамках первого учебно-практического занятия мы ознакомились с понятием вычислительная сложность. Были рассмотрены такие аспекты, как история возникновения данного понятия, расчет значения и способы выражения.

Вычислительная сложность - понятие в информатике и теории алгоритмов, обозначающее функцию зависимости объёма работы, которая выполняется некоторым алгоритмом, от размера входных данных.

Нам объяснили как выражается вычислительная сложность на нескольких примерах:

- «почистить ковёр пылесосом» требует время, линейно зависящее от его площади. Если площадь ковра увеличить в N раз, то и время, затраченное на его очистку тоже увеличиться в N раз.
- «найти имя в телефонной книге» требует всего лишь времени, логарифмически зависящего от количества записей. Так как телефонная книга отсортирована по алфавиту, достаточно открыть её посередине для уменьшения круга поиска нужного имени в двое. Ещё раз поделив оставшиеся страницы пополам, мы сократим поиск имени в 4 раза, и так далее. [5]

Рассмотрение входных данных большого размера и оценка порядка роста времени работы алгоритма приводят к понятию асимптотической сложности алгоритма. Наиболее популярной нотацией для описания вычислительной сложности алгоритмов является «О большое» Нотация О большое - это математическая нотация, которая описывает ограничивающее поведение функции, когда аргумент стремится к определенному значению или бесконечности. Впервые обозначение «О большое» было введено во втором томе книги математика Пауля Бахмана, вышедшем в 1894 году. [2]

Существуют следующие наиболее часто встречающиеся классы сложности алгоритмов:

- O(1) - константное время, например: определение чётности числа (представленного в двоичном виде); целого
- O(log n) - логарифмическое время, например: двоичный поиск; • O(n) - линейное время, например: поиск наименьшего или наибольшего элемента в неотсортированном массиве;
- O(n log n) - линейно-логарифмическое время, например: самый быстрый вариант сортировки сравнением, сортировка слиянием;
- O(n^2) - квадратичное время, например: пузырьком, сортировка вставками; сортировка n×n матриц, вычисление частичной корреляции;
- O(n^3) кубическое время, например: обычное умножение двух
- O(n^k) - полиномиальное время, например: алгоритм Кармаркара для линейного программирования, АКС-тест простоты;
- O(k^n) - экспоненциальное время, например: решение задачи порядке перемножения матриц с помощью полного перебора;
- O(n!) -- факториальное время, например: решение коммивояжёра полным перебором. задачи

Алгоритмы для вычисления ряда Фибоначчи

В рамках данной учебной практики мы изучили числовой ряд Фибоначчи, а также его математическое и историческое значение. Последовательность чисел, впервые описанная итальянским математиком Леонардо Фибоначчи в 1202 году, представляет собой ряд чисел, в которых каждый следующий элемент Фибоначчи равен сумме двух предыдущих.

Числа Фибоначчи находят широкое применение в различных областях, таких как программирование, теория чисел, искусство, архитектура, биология, и их изучение является необходимым этапом для освоения базовых физических алгоритмов.

На занятиях нам были продемонстрированы примеры классических алгоритмов для вычисления чисел Фибоначчи. Мы реализовали их с использованием языка программирования Python и проанализировали разницу в производительности.[3]

Задача №1

Первая задача, связанная с вычислением чисел Фибоначчи, звучит следующим образом:

Дано целое число $1 \leq n \leq 24$, необходимо написать функцию fib(n) для вычисления n n-го числа Фибоначчи с использованием рекурсии. Функция fib(n) должна вызывать сама себя в теле функции для вычисления соответствующих $(n-1)$ и $(n-2)$. В результате выполнения, функция должна вывести на экран вычисленное число Фибоначчи, например fib(6) должна вывести число 8, а fib(0) — соответственно 0.

Для решения этой задачи я написал такой алгоритм:

```
. . .
```

```
def fib(n):
    """Рекурсивная функция для вычисления n-го числа
Фибоначчи"""
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
...
```

1. **Имя функции и описание**: Функция называется fib, и её цель — вычислить n-е число Фибоначчи, где n — входной параметр функции.
2. **Параметр функции**: Функция принимает один параметр — n, который представляет собой индекс числа в последовательности Фибоначчи.
3. **Условия**:
   - Первое условие: if n <= 0:. Если значение n меньше или равно нулю, функция возвращает 0. Это базовый случай, так как F(0)=0F(0).
   - Второе условие: elif n == 1:. Если 1n=1, функция возвращает 1. Это второй базовый случай, так как F(1)=1F(1).
4. **Рекурсивный случай**: Если 1n>1, то функция возвращает результат сложения двух рекурсивных вызовов: fib(n - 1) + fib(n - 2). Это и есть основная идея алгоритма Фибоначчи, где каждое число в последовательности равно сумме двух предыдущих.
5. **Работа функции**:
   - Функция рекурсивно вызывает себя для n−1n и 2n−2 до тех пор, пока не достигает базовых случаев, где n≤1n. Тогда она начинает возвращать значения, которые постепенно складываются в итоговый результат.

Задача №2

Вторая задача, связанная с вычислением чисел Фибоначчи, звучит следующим образом:

Дано целое число 1≤n≤32, необходимо написать функцию fib(n) для вычисления n n-го числа Фибоначчи с использованием цикла. Функция fib(n) должна производить расчет от 1 до n , 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 fib(3) должна вывести число 2, а fib(7) — соответственно 13.

Для решения этой задачи я написал такой алгоритм:

```
...
def fib(n):
    """Функция для вычисления n-го числа Фибоначчи с
использованием цикла"""
    if n <= 0:
        return 0
    elif n == 1:
        return 1

    prev , curr = 0 , 1
    for _ in range ( 2 , n + 1 ):
        prev , curr = curr , prev + curr
    return curr
...
```

Функция fib(n) вычисляет n-е число Фибоначчи с использованием циклического подхода. Вместо рекурсии, здесь применяется итерация для вычисления значений последовательности Фибоначчи.

Сначала функция проверяет условия для базовых случаев: если n меньше или равно 0, возвращается 0, а если n равно 1, возвращается 1.

Если n больше 111, то функция использует цикл для вычисления следующих чисел. Начальные значения prev и curr устанавливаются на 000 и 111 соответственно. Затем цикл начинается с 222-го элемента и продолжается до n-го. В каждом шаге цикла переменные обновляются: prev становится старым значением curr, а curr становится суммой prev и curr.

После завершения цикла функция возвращает значение curr, которое представляет n-е число Фибоначчи. Этот итеративный подход более эффективен, чем рекурсия, так как имеет линейную сложность O(n) и требует меньше памяти.

Задача №3

Третья задача, связанная с вычислением чисел Фибоначчи, звучит следующим образом:

Дано целое число $1 \leq n \leq 40$, необходимо написать функцию fib(n) для вычисления n n-го числа Фибоначчи. Функция fib(n) 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 fib(8) должна вывести массив: [0, 1, 1, 2, 3, 5, 8, 13, 21].

Для решения этой задачи я написал такой алгоритм:

```
...
def fib(n):
```

```
    """Функция для вычисления чисел Фибоначчи с записью в
массив"""
    if n < 0:
        return []

    # Инициализация массива
    fib_sequence = [0] * (n + 1)

    # Задание начальных условий
    if n >= 1:
        fib_sequence[1] = 1

    # Вычисление чисел Фибоначчи
    for i in range(2, n + 1):
        fib_sequence[i] = fib_sequence[i - 1] +
fib_sequence[i - 2]

    return fib_sequence
...
```

Функция fib(n) вычисляет последовательность чисел Фибоначчи от F(0) до F(n) и возвращает их в виде массива. Если n<0n < 0n<0, возвращается пустой массив, так как последовательность для отрицательных индексов не определена. Для вычислений создаётся массив длиной n+1n + 1n+1, все элементы которого изначально равны нулю. Начальные условия задаются: F(0)=0 (уже задано в массиве) и F(1)=1, если n≥1n. Далее используется цикл, который проходит от 2 до n, на каждом шаге вычисляя текущее число Фибоначчи как сумму двух предыдущих чисел, сохранённых в массиве. Результаты записываются в массив, а по завершении работы возвращается массив, содержащий всю последовательность. Например, вызов fib(5) вернёт массив [0, 1, 1, 2, 3, 5]. Такой подход позволяет быстро вычислить последовательность с линейной сложностью O(n) и сохранить все числа для дальнейшего использования.[6]

Задача №4

Четвёртая задача, связанная с вычислением чисел Фибоначчи, звучит следующим образом:

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

$$F(n) = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}$$

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 fib(32) должна вывести число 2178309.

Для решения этой задачи я написал такой алгоритм:

```
...
def fib(n):
    """Функция для вычисления n-го числа Фибоначчи с
использованием формулы Бине"""
    if n < 0:
        raise ValueError ( "n должно быть больше или равно 0"
)

    # Формула Бине
    sqrt_5 = math.sqrt ( 5 )
    phi = (1 + sqrt_5) / 2  # Золотое сечение
    psi = (1 - sqrt_5) / 2

    # Вычисление числа Фибоначчи с округлением
    fib_n = (phi ** n - psi ** n) / sqrt_5
    return round ( fib_n )
...
```

Функция fib(n) вычисляет n-е число Фибоначчи с использованием формулы Бине. Если n < 0, функция вызывает исключение ValueError , так как последовательность Фибоначчи определена только для неотрицательных чисел. В основе расчётов лежит формула Бине, которая использует золотое сечение (ф) и его сопряжённое значение (ф). Сначала вычисляются ф = (1 + √5)/2, а затем по формуле (ф^n – ф^n)/√5 определяется n-е число Фибоначчи. Полученное значение округляется с помощью функции round() для устранения возможных ошибок, связанных с вычислениями с плавающей точкой, и возвращается результат. Этот метод эффективен и позволяет вычислять числа Фибоначчи за O(1) времени, но может быть менее точным для очень больших n, из-за ограничений точности чисел с плавающей точкой.

Задача №5

Пятая задача, связанная с вычислением чисел Фибоначчи, звучит следующим образом:

Дано целое число 1≤n≤10 6 , необходимо написать функцию fib_eo(n) для определения четности n n-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 $0 \le a$, $b \le 9$ 0≤a,b≤9 — последние цифры чисел $F\ n$ F n и $F\ n + 1$ F n+1 соответственно, то ($a + b$) mod 10 (a+b)mod10 — последняя цифра числа $F\ n + 2$ F n+2 .

В результате выполнения, функция должна вывести на экран четное ли число или нет (even или odd соответственно), например fib_eo(841645) должна вывести odd, т.к. последняя цифра данного числа — 5.

Для решения этой задачи я написал такой алгоритм:

```
...
def fib_eo(n):
    """Функция для определения четности n-го числа
Фибоначчи"""
    if n < 0:
        raise ValueError ( "n должно быть больше или равно 0"
)

    # Последние цифры двух первых чисел Фибоначчи
    a , b = 0 , 1

    # Циклическое вычисление последней цифры
    for _ in range ( n ):  # итерация от 0 до n-1
        a , b = b , (a + b) % 10

    # Последняя цифра n-го числа
    last_digit = a

    # Определение четности
    return "even" if last_digit % 2 == 0 else "odd"
...
```

Функция fib_eo(n) определяет, является ли n-е число Фибоначчи чётным или нечётным. Если n<0, вызывается ошибка. Для вычисления последней цифры числа Фибоначчи используется итеративный подход: переменные a и b инициализируются как первые два числа Фибоначчи (0 и 1), а затем обновляются по циклу с учётом остатка от деления на 10. После завершения цикла последняя цифра числа Фибоначчи сохраняется в переменной last_digit, на основе которой определяется его чётность. Функция возвращает строку "even" для чётных чисел и "odd" для нечётных.

Этот алгоритм обладает такой же высокой скоростью работы, как и другие алгоритмы с использованием циклов, но при этом отличается экономным

использованием памяти, так как в переменных сохраняются лишь значения младших разрядов обрабатываемых чисел.[8]

Задачи по алгоритмам Хаффмана

На этой лекции нам рассказали о Дэвиде Альберте Хоффмане, его достижениях в области науки и технологиях, а также подробно объяснили алгоритм, который он разработал.

Задача №1

Первая задача, связанная с кодирование строки по алгоритму Хаффмана, звучит следующим образом:

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
01101101101110110111100001110010101110100000101000011111001101
01011
```

Для решения данной задачи мной был разработан следующий алгоритм (см. Приложение №2).

Этот код реализует алгоритм Хаффмана для сжатия текста. Сначала класс Node описывает узлы дерева Хаффмана, где каждый узел хранит символ, его частоту, а также ссылки на левого и правого потомков. Функция build_huffman_tree строит дерево Хаффмана, используя частоты символов из текста и минимальную кучу для упорядочивания узлов. Функция build_codes обходит дерево, формируя префиксные коды для каждого символа, используя "0" для левого пути и "1" для правого. Основная функция huffman_encode принимает строку, строит дерево Хаффмана, генерирует коды символов, а затем кодирует строку с помощью этих кодов. На выходе отображаются количество символов, длина закодированной строки, таблица кодов и закодированный текст. Этот код демонстрирует основы сжатия данных и используется для учебных целей.

Алгоритм Хаффмана

Этапы работы алгоритма:

1. **Подсчет частот символов**:

- Определяется частота появления каждого символа в строке. Это служит основой для построения дерева Хаффмана.

2. **Создание минимальной кучи**:

- Каждый символ представляется узлом, частота символа используется как приоритет. Узлы добавляются в минимальную кучу, где элемент с наименьшим приоритетом извлекается первым.

3. **Построение дерева**:

- Из кучи извлекаются два узла с наименьшей частотой.
- Создается новый узел, частота которого равна сумме частот двух извлеченных узлов. Этот узел становится их родителем.
- Новый узел добавляется обратно в кучу.
- Процесс повторяется, пока в куче не останется один узел, который станет корнем дерева.

4. **Назначение кодов символов**:

- Присваивается двоичный код каждому символу, основываясь на пути от корня дерева до листа:
- Левый переход добавляет 0.
- Правый переход добавляет 1.

5. **Кодирование строки**:

- Каждый символ строки заменяется соответствующим двоичным кодом, полученным из дерева Хаффмана.

Пример работы алгоритма:

- **Входная строка**: "aaabbc"
- **Частоты символов**: { 'a': 3, 'b': 2, 'c': 1 }
- **Построение дерева**:
    1. Создаются узлы: ['a'(3), 'b'(2), 'c'(1)].
    2. Извлекаются два узла с наименьшей частотой: 'c'(1) и 'b'(2).
    3. Создается их родитель: '(b+c)'(3).
    4. Извлекаются 'a'(3) и '(b+c)'(3) для создания нового узла: '(a+b+c)'(6).

- **Полученное дерево**:

```
    (6)
   /   \
 'a'   (b+c)
       /   \
     'b'   'c'
```

- **Коды символов**:
  - 'a': "0", 'b': "10", 'c': "11".
- **Закодированная строка**: "000101011".

Задача №2

Вторая задача, связанная с кодирование строки по алгоритму Хаффмана, звучит следующим образом:

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.

В результате выполнения, функция huffman_decode() должна вывести на экран строку, которая была закодирована изначально.

Пример вывода для данного текста:
Docendo discimus.

Для решения данной задачи мной был разработан следующий алгоритм (см. Приложение №3).[7]

Функция huffman_decode()декодирует текст, закодированный с использованием алгоритма Хаффмана. На вход подаются данные, содержащие количество символов, длину закодированной строки, таблицу кодов символов и саму закодированную букву. Код считывает входные данные, разделяет их на заголовок, коды таблицы и закодированный текст. Таблица кодов преобразуется в словарь, где ключами являются двоичные коды, а значениями — символы. Затем функция плавно обрабатывает закодированный символ, собирая биты до тех пор, пока их комбинация не совпадет с ключом в таблице кодов. В результате добавляется первый символ, а временная последовательность обнуляется. В итоге

декодированная строка собирается и выводится. Этот код позволяет эффективно восстановить текст, закодированный алгоритмом Хаффмана.

Этапы алгоритма:

1. **Чтение входных данных**:
    - Входная строка разбивается на строки:
        - Первая строка (заголовок) содержит количество уникальных символов и длину закодированной строки (используется только для информации).
        - Последующие строки (кроме последней) содержат символы и их коды, которые используются для построения карты декодирования.
        - Последняя строка — это закодированная строка (последовательность битов), которую нужно декодировать.
2. **Создание карты декодирования**:
    - Каждая строка с описанием символа и его кода разбирается:
        - Символ извлекается из строки (с удалением кавычек).
        - Код извлекается и добавляется в словарь code_map как ключ, а символ — как значение.
    - Итог: Словарь, отображающий двоичные коды в символы.
3. **Декодирование закодированной строки**:
    - Проход по каждому биту закодированной строки:
        - Биты последовательно добавляются в временную переменную temp_code.
        - Если содержимое temp_code совпадает с одним из ключей в code_map, соответствующий символ добавляется в результирующий список decoded_string, а temp_code очищается.
4. **Формирование результата**:

- Все символы из списка decoded_string объединяются в одну строку decoded_result.
- Результат выводится на экран и возвращается из функции.

Разбор входных данных:

- Заголовок: 12 60 (не используется в декодировании).
- Карта кодов:

```
{
    "1011": " ", "1110": ".", "1000": "D",
    "000": "c", "001": "d", "1001": "e",
    "010": "i", "1100": "m", "1010": "n",
    "1111": "o", "011": "s", "1101": "u"
}
```

- Закодированная строка: "10001111000100110100011111101100101001100001011001101 0111110".
- **Процесс декодирования**:

- Считывается закодированная строка побитово:
  - 1000 → D
  - 1110 → .
  - 000 → c
  - 001 → d
  - ...
- Постепенно формируется результирующий список символов.

Алгоритмы сортировки

Алгоритмы сортировки — это методы упорядочивания элементов массива или списка в определенном порядке (обычно по возрасту или убыванию). Они берут на себя свою роль в программировании и компьютерных науках, создавая основу для многих других алгоритмов.[1]

Основные типы методов сортировки:

Сортировка пузырьком (Bubble Sort) — это один из простейших алгоритмов сортировки. Его основной принцип заключается в том, чтобы последовательно увеличивать элементы массива и менять их местами, если они расположены в неправильном порядке. Этот процесс повторяется до тех пор, пока массив не будет полностью отсортирован.[1]

Сортировка расчёской (Comb Sort) — это улучшенная версия сортировки пузырьком, которая позволяет использовать его главный недостаток — медленную скорость из-за большого количества мелких перестановок. В этом алгоритме используется понятие «шага» (промежуток) для сравнения элементов, находящихся на определенном расстоянии друг от друга. Следующий шаг перехода, пока не станет условием 1, что приводит сортировку к классической пузырьковой сортировке.

Сортировка выбора (Selection Sort) — это простой алгоритм сортировки, который на каждом шаге находит элемент из неотсортированной части массива и перемещает его в начало. Этот процесс повторяется до тех пор, пока весь массив не будет отсортирован.[4]

Пирамидальная сортировка (Heap Sort) — это метод сортировки, основанный на поэтапных данных «куча» (куча). Этот алгоритм относится к категории «выбор сортировки», так как он повторно выбирает максимальный (или создает) элемент и перемещает его в конец массива. Основной принцип работы заключается в построении и модификации бинарной кучи.

Поразрядная сортировка (Radix Sort) — это метод некомпаративной сортировки, который сортирует числа по их разрядкам (цифрам), начиная с младшего разряда (LSD — Lest Significant Digit) или старшего разряда (MSD — Most Significant Digit). Этот метод эффективен для чисел или строк с фиксированной длиной.

Заключение

После завершения учебной практики и выполнения работ, выполнил следующие задачи:

- Изучил понятие «вычислительная сложность» - узнали понятие вычислительной сложности, познакомились с несколькими примерами наглядно выражающие понятие вычислительная сложность, познакомились с понятием «О большое» и его классами сложностей.
- Изучил алгоритмы для чисел Фибоначчи - узнали об числовом ряде Фибоначчи, посмотрели на примеры различных алгоритмов разного времени выполнения и эффективности.
- Изучил алгоритм Хаффмана - познакомились с Дэвидом Альбертом Хаффманом и узнали какой алгоритм он изобрёл.
- Изучил алгоритмы сортировки - узнали об множестве различных алгоритмов, позволяющих по-разному выполнить сортировку ряда данных.

  Из предъявленных работ, я выполнил:

- Все задачи по числам Фибоначчи.
- Все задачи по алгоритму Хаффмана.

Электронные ресурсы

1. Алгоритм сортировки — Википедия [Электронный ресурс]. — Режим доступа : https://ru.wikipedia.org/wiki/Алгоритм_сортировки

2. O (обозначение Big O) — Википедия [Электронный ресурс]. — Режим доступа : https://ru.wikipedia.org/wiki/O_большое

3. Язык программирования Python — причины популярности [Электронный ресурс]. — Режим доступа : https://ru.wikipedia.org/wiki/«O»_большое_и_« o » _малое # История

4. ИТ-курсы ВШП — ВШП Онлайн [Электронный ресурс]. — Режим доступа : https://it.vshp.online/#/pages/up02/up02

5. NumPy — Официальная документация [Электронный ресурс]. — Режим доступа : https://numpy.org/

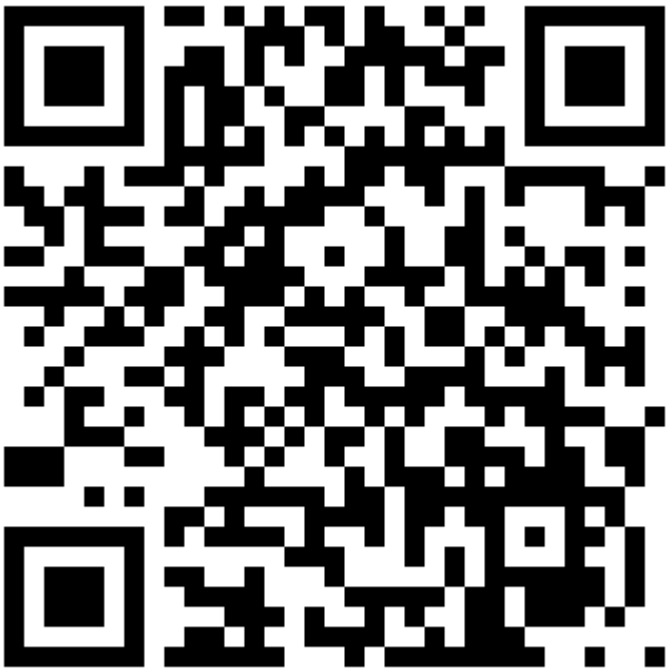
6. Документация Python 3.11.0 — timeit — Измерение времени выполнения небольших фрагментов кода [Электронный ресурс]. — Режим доступа : https://docs.python.org/3/library/timeit.html

7. Руководство по использованию Python [Электронный ресурс]. — Режим доступа : https://docs.python.org/3/tutorial/

8. Scikit-learn — Машинное обучение на Python [Электронный ресурс]. — Режим доступа : https://scikit-learn.org/stable/

Приложение №1

QR-код, содержании ссылку на репозиторий проекта:



Ссылка для ручного ввода:

https://github.com/Rom1z/algorithms_practicum

Инструкция:

Для того чтобы открыть ссылку, зашифрованную в QR-коде, воспользуйтесь приложением для получения QR-кодов через камеру вашего текущего устройства. Также вы можете вручную ввести указанный адрес.

Приложение №2

```
from collections import Counter
import heapq

class Node:
    """Класс для узла дерева Хаффмана"""
    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(text):
    """Строит дерево Хаффмана"""
    freq = Counter(text)
    heap = [Node(char, freq) for char, freq in freq.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, prefix="", code_map=None):
    """Создает коды для символов на основе дерева"""
    if code_map is None:
        code_map = {}
    if node is not None:
        if node.char is not None:
            code_map[node.char] = prefix
        build_codes(node.left, prefix + "0", code_map)
        build_codes(node.right, prefix + "1", code_map)
    return code_map


def huffman_encode(text):
    """Кодирование строк по алгоритму Хаффмана"""
    if not text:
        return "", {}
```

```
    # Построение дерева Хаффмана
    root = build_huffman_tree(text)

    # Создание кодов символов
    codes = build_codes(root)

    # Кодирование строки
    encoded_text = "".join(codes[char] for char in text)

    # Вывод результатов
    print(len(codes), len(encoded_text))
    for char, code in sorted(codes.items()):
        print(f"'{char}': {code}")
    print(encoded_text)

    return encoded_text, codes


if __name__ == "__main__":
    text = "Errare humanum est."
    huffman_encode(text)
```

Приложение №3

```
def huffman_decode():
    """Функция для декодирования строки по алгоритму
Хаффмана"""
    # Входные данные
    data = """12 60
' ': 1011
'.': 1110
'D': 1000
'c': 000
'd': 001
'e': 1001
'i': 010
'm': 1100
'n': 1010
'o': 1111
's': 011
'u': 1101
10001111000100110100011111101100101001100001011001101011111 0"
""
    lines = data.splitlines ()
    # Разбор входных данных
    header = lines[0]

    encoded_string = lines[-1]
    code_map = {}
    for line in lines[1:-1]:
        symbol , code = line.split ( ": " )
        symbol = symbol.strip ( "'" )
        code_map[code] = symbol

    # Декодирование строки
    decoded_string = []
    temp_code = ""

    for bit in encoded_string:
        temp_code += bit
        if temp_code in code_map:
            decoded_string.append ( code_map[temp_code] )
            temp_code = ""

    # Вывод результата
    decoded_result = "".join ( decoded_string )
    print ( decoded_result )
    return decoded_result


if __name__ == "__main__":
    huffman_decode ()
```