[Год]

[Название документа]

[ПОДЗАГОЛОВОК ДОКУМЕНТА]

КИРИЛЛ МАТВЕЕВ

Содержание

Введение

Цель данной работы состоит в изучении и анализе алгоритмов, связанных с вычислением чисел Фибоначчи, алгоритмов Хаффмана и методов сортировки. Эти алгоритмы находят широкое применение в информатике, охватывая такие области, как задачи оптимизации, кодирование данных и выполнение базовых операций обработки информации. Основное внимание уделено исследованию вычислительной сложности указанных алгоритмов и проведению сравнительного анализа их эффективности в различных условиях.

Для достижения поставленной цели в процессе работы были выполнены следующие задачи:

Изучены основные подходы к вычислению чисел Фибоначчи, включая рекурсивный метод, применение мемоизации, итеративные алгоритмы и метод быстрого возведения матриц в степень.

Проведен анализ алгоритма Хаффмана, применяемого для построения оптимальных префиксных кодов, а также изучены его ключевые этапы: построение дерева Хаффмана и генерация кодов для символов.

Исследованы классические алгоритмы сортировки, такие как сортировка пузырьком, сортировка вставками, сортировка выбором и сортировка слиянием, с акцентом на анализ их временной и пространственной сложности.

Проведен сравнительный анализ всех изученных алгоритмов с оценкой их вычислительной сложности и эффективности в различных сценариях применения.

В процессе практики была проведена разработка и тестирование программной реализации каждого алгоритма, что позволило не только изучить их теоретические аспекты, но и оценить их поведение на практике.

Вычислительная сложность

Приступая к изучению практики, мы начали с введения в понятие вычислительной сложности. На первом занятии были рассмотрены различные типы сложности, их особенности, а также основные нотации, применяемые для описания вычислительной сложности.

Вычислительная сложность — понятие в информатике и теории алгоритмов, обозначающее функцию зависимости объёма работы, которая выполняется некоторым алгоритмом, от размера входных данных. [1]

Примеры, как вычислительная сложность выражается:

Линейная сложность O(n):

Представим, что требуется почистить ковер пылесосом. Время уборки будет зависеть от площади ковра. Если площадь увеличится вдвое, время уборки также возрастет в два раза. Этот процесс имеет линейную сложность.

Логарифмическая сложность O(log n):

Поиск имени в упорядоченной по алфавиту телефонной книге может быть выполнен с помощью бинарного поиска. Каждый шаг уменьшает количество оставшихся страниц вдвое, что значительно ускоряет процесс. В результате имя можно найти за несколько шагов даже в очень большой книге.

Рассмотрение алгоритмов с большими размерами входных данных приводит к понятию асимптотической сложности, которая помогает оценить порядок роста времени работы алгоритма. Алгоритмы с меньшей асимптотической сложностью, как правило, более эффективны, особенно при увеличении размера входных данных.

Наиболее популярной нотацией для описания вычислительной сложности алгоритмов является «О большое» (Big O Notation или просто Big O).

Нотация О большое — это математическая нотация, которая описывает ограничивающее поведение функции, когда аргумент стремится к определенному значению или бесконечности. О большое является членом семейства нотаций, изобретенных Паулем Бахманом, Эдмундом Ландау и рядом других ученых, которые в совокупности называются нотациями Бахмана-Ландау или асимптотическими нотациями. [2]

В рамках теории вычислительной сложности кроме нотации «О большое» существуют и другие: «о малое», «Омега» и «Тета»

- О-большое (O(n)) — описывает верхнюю границу сложности, то есть наихудший случай выполнения алгоритма.
- о-малое (o(n)) — уточняет верхнюю границу, исключая точную оценку, и характеризует порядок роста функции.
- Омега (Ω(n)) — описывает нижнюю границу сложности, то есть наилучший случай.

Тета (Θ(n)) — дает точную оценку сложности, указывая, какова сложность с учетом особенностей входных данных.

Исходя из всего перечисленного можно сказать, что нотация О-большое используется для описания алгоритмической сложности, выражая, как меняется предполагаемое время выполнения алгоритма или объем потребляемой им памяти в зависимости от размера входных данных. Она позволяет абстрагироваться от незначительных деталей реализации, сосредотачиваясь на главном — скорости роста ресурсоемкости алгоритма при увеличении объема задачи. Эта нотация особенно полезна для сравнения эффективности разных подходов, помогая выбрать оптимальный из них.

Алгоритмы для чисел Фибоначчи

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

Для вычисления чисел Фибоначчи я рассмотрел несколько подходов на языке JavaScript, реализованных в виде кода, с использованием различных методов. В каждом случае были выбраны оптимальные инструменты для достижения целей и оценки производительности.

Задача №1

Дано целое число $1 \leq n \leq 24$, необходимо написать функцию fib(n) для вычисления 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.

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 написал алгоритм:

```
// Рекурсив функции для вычсиления числа фибоначи
function fib(n){
    if (n <= 1){
        return n;
    } else{
        return fib(n - 1) + (n - 2);
    }
}
```

В данном коде реализовано вычисление чисел Фибоначчи с использованием рекурсивной функции на языке JavaScript, а также замер времени выполнения для каждого вычисления. Рекурсивная функция fib(n) определяет число Фибоначчи следующим образом: если n = 0, функция возвращает 0, если n = 1, функция возвращает 1. Для всех других значений n функция вызывает сама себя для (n−1) и (n−2), возвращая их сумму. Для ознакомления полного разбора алгоритма (см. Приложение 2)

Для проверки и тестирования алгоритма я выбрал несколько значений n, для которых вычислил числа Фибоначчи в миллисекундах и замерил время выполнения. Вот результаты:

```
fib(5) = 7, выполнено за 0.015 милисекунды
fib(10) = 37, выполнено за 0.002 милисекунды
fib(15) = 92, выполнено за 0.002 милисекунды
fib(20) = 172, выполнено за 0.001 милисекунды
fib(24) = 254, выполнено за 0.002 милисекунды
```

Исходя из тестирования, рекурсивный подход неэффективен для больших значений n из-за увеличения количества вызовов функций и повторного вычисления одних и тех же значений. Для улучшения производительности можно использовать оптимизированные методы, такие как мемоизация или итеративный подход.

Задача №2

Дано целое число $1 \leq n \leq 32$, необходимо написать функцию fib(n) для вычисления *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.

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

Для решения этой задачи я написал алгоритм:

```
function fib(n) {
    if (n === 1 || n === 2) {
        return 1;
    }
    let prev = 1, curr = 1;
    for (let i = 3; i <= n; i++) {
        let temp = curr;
        curr = prev + curr;
        prev = temp;
    }
    return curr;
}
```

В данном коде вычисление чисел Фибоначчи реализовано с использованием итеративного подхода. Функция fib(n) возвращает n-е число Фибоначчи, обеспечивая более эффективное вычесление по сравнению с рекурсивным методом. Если n = 1, или n = 2, функция сразу возвращает 1. Для значений $n \geq 3$ используется цикл, в котором две переменные prev и curr хранят предыдущие числа Фибоначчи. На каждой итерации цикла: обновляется на значение текущего числа, а curr — становится суммой текущего и

предыдущего числа. По завершении цикла переменная curr содержит n-е число Фибоначчи. Такой подход исключает избыточные вычисления, характерные для рекурсивного метода, и имеет линейную временную сложность. Для ознакомления полного разбора алгоритма (см. Приложение 3)

Для проверки и тестирования алгоритма я выбрал несколько значений n, для которых вычислил числа Фибоначчи в миллисекундах и замерил время выполнения. Вот результаты:

```
fib(5), Фибоначчи = 5, Время: 0.0253 мс

fib(10), Фибоначчи = 55, Время: 0.0014 мс

fib(15), Фибоначчи = 610, Время: 0.0021 мс

fib(20), Фибоначчи = 6765, Время: 0.0011 мс

fib(30), Фибоначчи = 832040, Время: 0.0020 мс
```

Исходя из тестирования вычисление n-го числа Фибоначчи с использованием цикла для вычисления чисел Фибоначчи является оптимальным и быстрым решением для любых значений n. Он решает задачу эффективно, с линейной сложностью O(n), и работает стабильно даже для больших значений. При этом время выполнения настолько быстрое, что для малых n, как в тестах, оно может быть округлено до 0 мс, что подчеркивает быстродействие алгоритма.

Задача №3

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 алгоритм:

```
function fib(n) {
    if (n < 0 || n > 40) {
        throw new Error('n должно быть в диапазоне от 1 до 40');
    }

    // Массив для хранения чисел Фибоначчи
    const fibArray = [0, 1];

    // Заполнение массива до n-го числа
    for (let i = 2; i <= n; i++) {
        fibArray[i] = fibArray[i - 1] + fibArray[i - 2];
    }
```

В этом коде я ты вычислял последовательность Фибоначчи до n-го элемента включительно. Сначала создается массив fibArray длиной n+1,

где первые два элемента являются это 0 и 1, а остальные инициализируются нулями. Потом просто с помощью цикла можно проходить по индексам от 2 до n и таким образом вычисляется каждый элемент последовательности как сумму двух предыдущих. Для полного ознакомления кода можно обратиться (см. Приложение 4)

Для проверки и тестирования алгоритма я задавал значение n=8 и вызывалась функцию для вычисления последовательности и вывелся определенный результат:

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

Исходя из вычисления n-го числа Фибоначчи с сохранением числового ряда в массиве корректно работает и демонстрирует линейную временную сложность O(n). Алгоритм гибко возвращает всю последовательность Фибоначчи до n-го элемента включительно, но при этом требует O(n) памяти.

Задача №4

Дано целое число $1 \leq n \leq 64$, необходимо написать функцию fib(n) для вычисления *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 алгоритм:

```
function fib(n) {
    if (n < 1 || n > 64) {
        throw new Error('n должно быть в диапазоне от 1 до 64');
    }

    const sqrt5 = Math.sqrt(5);
    const phi = (1 + sqrt5) / 2;
    const psi = (1 - sqrt5) / 2;

    // Вычисление числа Фибоначчи по формуле Бине
    const fibNumber = Math.round((Math.pow(phi, n) - Math.pow(psi, n))
/ sqrt5);

    console.log(fibNumber);
    return fibNumber;
}
```

В этом коде я вычисляю n-е число Фибоначчи, используя формулу Бине. Сначала определяются два ключевых значения: r (золотое сечение) и i (обратное золотое сечение), которые рассчитываются через корень из 5. Затем само число Фибоначчи вычисляется по формуле, где берется разность r^n и i^n, деленная на корень из 5. Результат округляется с помощью функции Math.round для устранения погрешностей, связанных с вычислениями с плавающей точкой. Также добавлена проверка, чтобы значение n находилось в пределах от 1 до 64, и в случае выхода за эти пределы выводило ошибку. Для полного ознакомления кода можно обратиться (см. Приложение 5)

Для проверки и тестирования алгоритма вычисления n-го числа Фибоначчи при помощи формулы Бине, я задал, например значение n=32 и вызывалась функцию для вычисления числа через Бине и вывелся определенный результат:

```
2178309
```

Исходя из вычисление n-го числа Фибоначчи с использованием формулы Бине работает корректно и демонстрирует константную временную сложность O (1). Алгоритм эффективно вычисляет конкретное число Фибоначчи без

необходимости сохранять весь числовой ряд, что делает его память независимой от n. Однако, из-за вычислений с плавающей точкой возможны погрешности для больших значений n, что стоит учитывать при использовании.

Задача №5

Дано целое число $1 \leq n \leq 10^6$, необходимо написать функцию fib_eo(n) для определения четности 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 $0 \leq a, b \leq 9$ — последние цифры чисел Fn и Fn+1 соответственно, то (a+b) mod10 — последняя цифра числа Fn+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 алгоритм:

```
function fib_eo(n) {
    if (n < 1 || n > 1_000_000) {
        throw new Error('n должно быть в диапазоне от 1 до 10^6');
    }

    // Начальные значения для последней цифры чисел Фибоначчи
    let a = 0; // F(0) % 10
    let b = 1; // F(1) % 10

    // Вычисление последней цифры числа Фибоначчи по модулю 10
    for (let i = 2; i <= n; i++) {
        const next = (a + b) % 10;
        a = b;
        b = next;
    }

    // Последняя цифра числа Фибоначчи
    const lastDigit = n === 1 ? a : b;
```

```
    // Определение четности
    const result = lastDigit % 2 === 0 ? 'even' : 'odd';

    console.log(result);
    return result;
}
```

В этом коде я определяю чётность n-го числа Фибоначчи, используя свойства последовательности по модулю 10. Сначала вычисляется остаток от деления n на 60, так как последовательность чисел Фибоначчи по модулю 10 повторяется с периодом 60. Затем, начиная с первых двух чисел Фибоначчи, выполняется итерация для нахождения последней цифры n-го числа. В конце определяется чётность этой последней цифры: если она чётная, возвращается "even", если нечётная — "odd". Для ознакомления кода (см. Приложение 6)

Для определения четности n-го большого числа Фибоначчи я задал, например значение n = 841645 и должно вывести odd (нечетное), так как на конце 5, итак, понятно.

Исходя из тестирования этого алгоритма можно сказать, что он позволяет эффективно определять чётность n-го числа Фибоначчи, избегая необходимости вычисления самого числа. Благодаря числам Фибоначчи по модулю 10, алгоритм значительно сокращает количество вычислений, используя свойства числовой последовательности. Это приводит к временной сложности O (1), что делает решение очень быстрым и экономным по времени.

Рассмотрев все задачи Фибоначчи, я могу выделить несколько ключевых выводов, которые наиболее важны такие как: Первое это эффективность алгоритмов на практике я понял, что самые эффективные методы - использование формулы Бине и определение чётности по модулю 10, так как они имеют время выполнения, не зависящее от размера входных данных O (1), и не требуют лишних вычислений. Эти методы позволяют быстро и точно получать результат, что особенно важно при больших значениях n. Второе, что я могу подметить это рекурсивный подход, он хотя и понятен, но слишком

неэффективен для больших значений n, так как его временная сложность быстро растет. Этот метод подходит лишь для небольших значений n, и его стоит избегать в задачах с большими числами. Третье могу добавить – это оптимизация памяти методы с использованием массива или хранения всей последовательности чисел Фибоначчи могут быть полезны, если нужно не только получить одно число, но и использовать всю последовательность. Однако для вычисления конкретного числа это не самый эффективный способ, так как он требует лишней памяти.

Задачи по алгоритмам Хаффмана

Алгоритм Хаффмана - алгоритм оптимального префиксного кодирования некоторого алфавита с минимальной избыточностью. [4]

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

01101101101110110111100001110010101110100000101000011111
00110101011
```

Для решения задачи кодирование строки по алгоритму Хаффмана я создал следующий алгоритм (см. Приложение 7)

В решении моей задачи по алгоритму Хаффмана используется функция huffman_decode. Эта функция принимает 4 параметра:

symbol_count количество различных символов, используемых в коде (например, 12). Хотя в данной функции этот параметр не используется, он важен для контекста и может быть полезен для анализа структуры данных. encoded_size длина закодированной строки в битах (например, 60). Этот параметр также не используется напрямую в функции, но может быть полезен для оценки эффективности сжимающей операции.

codes словарь, в котором ключами являются символы, а значениями — их Хаффман-коды (например, ' ': '1011').

encoded_string строка, состоящая из битов, которая была закодирована с использованием алгоритма Хаффмана. Функция huffman_decode преобразует

эту строку обратно в исходный текст, используя предоставленные Хаффман-коды.

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.

Для решения задачи декодирование строки по алгоритму Хаффмана я создал следующий алгоритм (см. Приложение 8)

В решении моей задачи декодирование строки по алгоритму Хаффмана используется функция haffman_decode. Эта функция принимает 4 параметра:
symbolCount: количество различных символов, используемых в кодировке (например, 12). В функции данный параметр не используется напрямую, но он помогает описать контекст и понять, как будет строиться кодировка.
encodedSize: длина закодированной строки в битах (например, 60). Этот параметр также не используется в самой функции, но предоставляет дополнительную информацию о размерах закодированного текста.
codes: словарь, в котором ключами являются символы, а значениями — их Хаффман-коды. Например, для пробела код может быть '1011', для точки — '1110'.
encodedString строка, содержащая последовательность битов, которые необходимо декодировать. В этой строке хранится закодированное сообщение.

Алгоритмы сортировки

На практике, рассматривая тему алгоритмов сортировки, мы выделили основные и наиболее важные из них:

Сортировка пузырьком — один из самых известных, но малоэффективных алгоритмов сортировки. Он работает путем последовательного сравнения соседних элементов и их обмена местами, если предыдущий элемент больше следующего. Несмотря на свою популярность, сортировка пузырьком практически не используется в реальных задачах из-за своей низкой эффективности, особенно когда в конце массива находятся элементы с маленькими значениями (так называемые "черепахи").

Шейкерная сортировка улучшенная версия пузырьковой сортировки, но двунаправленная. Алгоритм сначала перемещается слева направо, а затем справа налево, что позволяет быстрее устранять элементы, стоящие на неправильных позициях.

Сортировка расчёской — это оптимизация пузырьковой сортировки. В отличие от пузырьковой и шейкерной сортировок, где элементы сравниваются только по соседству, в сортировке расчёской для начала используется большое расстояние между сравниваемыми элементами, которое постепенно сужается до минимального. Это позволяет ускорить процесс сортировки за счет более быстрого устранения «мелких» элементов в конце массива.

Сортировка вставками алгоритм, при котором массив постепенно перебирается слева направо, и каждый следующий элемент вставляется на правильную позицию между ближайшими элементами с минимальным и максимальным значением. Это достаточно эффективный алгоритм для сортировки небольших массивов.

Быстрая сортировка состоит из трех шагов: выбора опорного элемента, перераспределения других элементов массива так, чтобы те, что меньше опорного, оказались перед ним, а те, что больше или равны — после. После этого рекурсивно применяется эта же операция к подмассивам слева и справа от опорного элемента. Быстрая сортировка является одним из самых быстрых алгоритмов для сортировки больших массивов.

Сортировка слиянием алгоритм, который эффективен для сортировки структур данных, где доступ к элементам осуществляется последовательно (например, для потоков). Массив разбивается на две примерно равные части, каждая из которых сортируется отдельно, после чего два отсортированных подмассива сливаются в один отсортированный массив.

Пирамидальная сортировка при этой сортировке сначала строится пирамида (или двоичная куча) из элементов исходного массива. Пирамида представляет собой структуру, в которой родительский элемент больше своих дочерних. Затем элементы извлекаются из пирамиды, и массив сортируется.

Поразрядная сортировка (Radix sort) — сортировка по разрядам. Существует две разновидности: LSD (least significant digit) и MSD (most significant digit). В первом случае происходит сортировка элементов по младшим разрядам (все оканчивающиеся на 0, затем на 1 и так до 9). После этого они группируются по следующему с конца разряду, пока они не закончатся. В MSD сортировка происходит по старшему разряду. [5]

Заключение

В ходе практической работы были изучены различные алгоритмы, связанные с вычислением чисел Фибоначчи, алгоритмами Хаффмана и алгоритмами сортировки. Проведенное исследование позволило получить всестороннее представление о данных алгоритмах, их эффективности и области применения в различных условиях.

Основные выводы по итогам работы:

Вычисление чисел Фибоначчи: Анализ разных методов, включая рекурсивный подход и быстрые алгоритмы, такие как возведение матрицы в степень, показал, что выбор алгоритма зависит от требований к скорости выполнения и доступной памяти. Итеративные алгоритмы и метод с матрицами показывают наилучшие результаты при работе с большими числами, обеспечивая хороший баланс между скоростью и расходом ресурсов.

Алгоритм Хаффмана: Этот алгоритм оказался эффективным инструментом для задач сжатия данных. Использование дерева Хаффмана и префиксных кодов позволяет значительно уменьшить размер закодированной информации, что делает его незаменимым в области оптимизации представления данных и их кодирования.

Алгоритмы сортировки: Сравнение классических алгоритмов сортировки, таких как пузырьковая сортировка, сортировка вставками, быстрая сортировка и сортировка слиянием, показало, что простые методы, такие как пузырьковая сортировка, подходят только для небольших объемов данных. В то время как более сложные алгоритмы, например быстрая сортировка и сортировка слиянием, показывают высокую производительность

при работе с большими массивами, что делает их предпочтительным выбором в большинстве практических случаев.

В целом, эффективность алгоритмов во многом зависит от условий их применения, структуры входных данных и ограничений, таких как время выполнения и объем доступной памяти.

Список источников

1. УП.02 - Вычислительная сложность [Электронный ресурс] / – Режим доступа: https://it.vshp.online/#/pages/up02/up02_complexity

2. Обозначение «Большое О» [Электронный ресурс] / – Режим доступа: https://en.wikipedia.org/wiki/Big_O_notation

3. УП.02 - Алгоритмы для вычисления ряда Фибоначчи [Электронный ресурс] / – Режим доступа: https://it.vshp.online/#/pages/up02/up02_fibonacci

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

5. УП.02 - Алгоритмы сортировки [Электронный ресурс] / – Режим доступа: https://it.vshp.online/#/pages/up02/up02_sort

Приложение 1. QR код.

https://github.com/Nasvaychik/algorithms_practicum

Приложение 2. Числа Фибоначчи Задача №1

```
// Рекурсив функции для вычсиления числа фибоначи
function fib(n){
    if (n <= 1){
        return n;
    } else{
        return fib(n - 1) + (n - 2);
    }
}

// Функция для замера времени
function measuringtheexecutiontime(n){
    const start = performance.now();
    const result = fib(n);
    const end = performance.now();
    console.log(`fib(${n}) = ${result}, выполнено за ${(end - start).toFixed(3)} милисекунды`);
}

//Тесты
const   testValues = [5, 10, 15, 20, 24];
testValues.forEach(n => measuringtheexecutiontime(n));
```

Приложение 3. Числа Фибоначчи Задача №2

```
// Сама функция
function fib(n) {
    if (n === 1 || n === 2) {
        return 1;
    }
    let prev = 1, curr = 1;
    for (let i = 3; i <= n; i++) {
        let temp = curr;
        curr = prev + curr;
        prev = temp;
    }
    return curr;
}

// Замеры времени выполнения для нескольких ззначений n
const testValues = [5, 10, 15, 20, 30];
testValues.forEach(n => {
     const startTime = performance.now();
    const result = fib(n);
    const elapsedTime = performance.now() - startTime;
    console.log(`n = ${n}, Фибоначчи = ${result}, Время =
${elapsedTime.toFixed(4)} мс`);
});
```

Приложение 4. Числа Фибоначчи Задача №3

```
// Функция для вычисления n-го числа Фибоначчи
function fib(n) {
    if (n < 0 || n > 40) {
        throw new Error('n должно быть в диапазоне от 1 до 40');
    }

    // Массив для хранения чисел Фибоначчи
    const fibArray = [0, 1];

    // Заполнение массива до n-го числа
    for (let i = 2; i <= n; i++) {
        fibArray[i] = fibArray[i - 1] + fibArray[i - 2];
    }

    // Вывод массива
    console.log(fibArray);
    return fibArray;
}

// Пример вызова функции
fib(8); // Ожидаемый вывод: [0, 1, 1, 2, 3, 5, 8, 13, 21]
```

Приложение 5. Числа Фибоначчи Задача №4

```
// Функция для вычисления n-го числа Фибоначчи с использованием
формулы Бине
function fib(n) {
    if (n < 1 || n > 64) {
        throw new Error('n должно быть в диапазоне от 1 до 64');
    }

    const sqrt5 = Math.sqrt(5);
    const phi = (1 + sqrt5) / 2;
    const psi = (1 - sqrt5) / 2;

    // Вычисление числа Фибоначчи по формуле Бине
    const fibNumber = Math.round((Math.pow(phi, n) - Math.pow(psi, n))
/ sqrt5);

    console.log(fibNumber);
    return fibNumber;
}

// Пример вызова функции
fib(32); // Ожидаемый вывод: 2178309
```

Приложение 6. Числа Фибоначчи Задача №5

```
// Функция для определения четности n-го числа Фибоначчи
function fib_eo(n) {
    if (n < 1 || n > 1_000_000) {
        throw new Error('n должно быть в диапазоне от 1 до 10^6');
    }

    // Начальные значения для последней цифры чисел Фибоначчи
    let a = 0; // F(0) % 10
    let b = 1; // F(1) % 10

    // Вычисление последней цифры числа Фибоначчи по модулю 10
    for (let i = 2; i <= n; i++) {
        const next = (a + b) % 10;
        a = b;
        b = next;
    }

    // Последняя цифра числа Фибоначчи
    const lastDigit = n === 1 ? a : b;

    // Определение четности
    const result = lastDigit % 2 === 0 ? 'even' : 'odd';

    console.log(result);
```

```
    return result;
}


// Пример вызова функции
fib_eo(841645); // Ожидаемый вывод: odd
```

Приложение 7. Задачи по алгоритмам Хаффмана №1

```
// Импортируем модуль для работы с приоритетной очередью
class PriorityQueue {
    constructor() {
        this.queue = [];
    }

    enqueue(node) {
        this.queue.push(node);
        this.queue.sort((a, b) => a.freq - b.freq);
    }

    dequeue() {
        return this.queue.shift();
    }

    size() {
        return this.queue.length;
    }
}

// Узел дерева Хаффмана
class HuffmanNode {
    constructor(char, freq) {
        this.char = char;
        this.freq = freq;
        this.left = null;
        this.right = null;
    }
}

// Функция для построения дерева Хаффмана
function buildHuffmanTree(frequencies) {
    const pq = new PriorityQueue();

    // Создаем узлы для каждого символа и добавляем их в очередь
    for (const [char, freq] of Object.entries(frequencies)) {
        pq.enqueue(new HuffmanNode(char, freq));
    }

    // Строим дерево
```

```
    while (pq.size() > 1) {
        const left = pq.dequeue();
        const right = pq.dequeue();

        const newNode = new HuffmanNode(null, left.freq + right.freq);
        newNode.left = left;
        newNode.right = right;

        pq.enqueue(newNode);
    }

    return pq.dequeue();
}

// Функция для создания кодов Хаффмана
function generateHuffmanCodes(node, prefix = "", codes = {}) {
    if (!node) return;

    if (node.char !== null) {
        codes[node.char] = prefix;
    }

    generateHuffmanCodes(node.left, prefix + "0", codes);
    generateHuffmanCodes(node.right, prefix + "1", codes);

    return codes;
}

// Функция для кодирования строки по алгоритму Хаффмана
function huffman_encode(input) {
    // Подсчет частот символов
    const frequencies = {};
    for (const char of input) {
        frequencies[char] = (frequencies[char] || 0) + 1;
    }

    // Построение дерева Хаффмана
    const huffmanTree = buildHuffmanTree(frequencies);

    // Генерация кодов Хаффмана
    const huffmanCodes = generateHuffmanCodes(huffmanTree);

    // Кодирование строки
    let encodedString = "";
    for (const char of input) {
        encodedString += huffmanCodes[char];
    }

    // Вывод результатов
    console.log(Object.keys(frequencies).length,
encodedString.length);
    for (const [char, code] of Object.entries(huffmanCodes)) {
        console.log(`'${char}': ${code}`);
```

```
    }
    console.log(encodedString);

    return { huffmanCodes, encodedString };
}

// Пример вызова функции
huffman_encode("Errare humanum est.");
```

Приложение 8. Задачи по алгоритмам Хаффмана №2

```
// Функция для декодирования строки по алгоритму Хаффмана
function huffman_decode(symbolCount, encodedSize, codes,
encodedString) {
    // Инвертируем словарь кодов для поиска символов по кодам
    const reversedCodes = Object.fromEntries(
        Object.entries(codes).map(([char, code]) => [code, char])
    );

    let decodedString = "";
    let buffer = "";

    // Проходим по закодированной строке, восстанавливая исходные
символы
    for (const bit of encodedString) {
        buffer += bit;
        if (reversedCodes[buffer]) {
            decodedString += reversedCodes[buffer];
            buffer = "";
        }
    }

    // Вывод восстановленной строки
    console.log(decodedString);
    return decodedString;
}

// Пример вызова функции
const symbolCount = 12;
const encodedSize = 60;
const codes = {
    ' ': '1011',
    '.': '1110',
    'D': '1000',
    'c': '000',
    'd': '001',
    'e': '1001',
    'i': '010',
    'm': '1100',
```

```
    'n': '1010',
    'o': '1111',
    's': '011',
    'u': '1101'
};
const encodedString =
"1000111100010011010001111110110010100110000101100110101111110";

huffman_decode(symbolCount, encodedSize, codes, encodedString); //
Ожидаемый вывод: Docendo discimus.
```