Содержание

Введение

Целью настоящей работы является изучение и анализ алгоритмов, которые используются для вычисления чисел Фибоначчи, работы алгоритмов Хаффмана и выполнения сортировки данных. Эти алгоритмы имеют широкое применение в информатике, включая задачи оптимизации, кодирование информации и выполнение основных операций обработки данных. Главный акцент сделан на анализе вычислительной сложности алгоритмов и сравнении их производительности в различных условиях.

Для реализации цели были решены следующие задачи:

Изучены основные способы вычисления чисел Фибоначчи, включая рекурсивный метод, мемоизацию, итеративные подходы и алгоритм быстрого возведения матриц в степень.

Проведен разбор алгоритма Хаффмана, который используется для создания оптимальных префиксных кодов. Рассмотрены ключевые этапы: построение дерева Хаффмана и генерация кодов символов. Исследованы популярные алгоритмы сортировки, такие как сортировка пузырьком, вставками, выбором и слиянием, с акцентом на их временную и пространственную сложность.

Выполнено сравнение изученных алгоритмов с учетом их вычислительной сложности и эффективности в различных условиях применения.

В рамках практической части работы были созданы и протестированы программные реализации алгоритмов, что позволило не только изучить теоретические аспекты, но и проверить их в реальных условиях.

Вычислительная сложность

На начальном этапе работы было рассмотрено понятие вычислительной сложности: её основные типы, характеристики и обозначения, которые используются для описания.

Вычислительная сложность определяет, как зависит объем работы алгоритма от размера входных данных. Примеры:

Линейная сложность O(n): Время выполнения растет пропорционально увеличению входных данных. Например, время уборки ковра пылесосом увеличивается вместе с его площадью.

Логарифмическая сложность O(log n): Поиск имени в упорядоченной телефонной книге можно провести бинарным поиском, при котором объем данных сокращается вдвое на каждом шаге.

При анализе алгоритмов с большими объемами данных применяется понятие асимптотической сложности, которая отражает рост времени выполнения алгоритма. Наиболее популярная нотация для описания сложности — «О большое» (Big O Notation).

Основные виды нотаций:

O(n): Верхняя граница сложности, описывающая худший сценарий.

o(n): Более точное указание верхней границы, исключающее точные значения.

Ω(n): Нижняя граница, описывающая лучший случай выполнения.

Θ(n): Точная оценка сложности, отражающая реальное время выполнения.

Использование нотации «О большое» позволяет исключить второстепенные детали реализации алгоритма и сосредоточиться на основных характеристиках роста его ресурсоемкости при увеличении входных данных. Это упрощает сравнение различных алгоритмов и выбор наиболее оптимального решения.

Алгоритмы чисел Фибоначчи

Вычисление чисел из ряда Фибоначчи является одной из классических задач, часто используемых на собеседованиях для проверки знаний кандидата о базовых алгоритмах. [3]

В ходе работы я исследовал несколько способов вычисления чисел Фибоначчи, реализованных на языке JavaScript с использованием TypeScript. Для каждого метода была выполнена реализация кода с применением соответствующих подходов. В каждом случае использовались наиболее подходящие инструменты для достижения целей и проведения анализа 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 данной задачи я реализовал свою функцию для вычисления алгоритма:

```
function fib(n: number): number {
  if (n > 24) {
    throw new Error("Значение должно быть не бо
24.")
  } else if (n <= 1) {
```

```
    return n;
  } else {
    return fib(n - 1) + fib(n - 2);
  }
}
```

В этом коде реализован алгоритм вычисления чисел Фибоначчи с помощью рекурсивной функции на языке JavaScript. Также выполняется замер времени выполнения для каждого значения. Функция fib(n) работает следующим образом: если n = 0, она возвращает 0; если n = 1, возвращается 1. Для всех остальных значений функция вызывает саму себя с аргументами (n−1) и (n−2) и возвращает их сумму. Подробный разбор алгоритма приведен в Приложении 2.

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

Результаты моей реализации алгоритма на JavaScript:

Значение	Результат	Время выполнения
10	55	35.656 мс
15	610	143.620 мс
20	6765	763.560 мс
24	46368	1150.3 мс

Результаты тестирования показали, что рекурсивный метод не подходит для работы с большими значениями n, так как приводит к значительному увеличению числа вызовов функции и многократному пересчету одних и тех же значений. Для повышения эффективности алгоритма можно применять оптимизированные подходы, такие как мемоизация или итеративные методы.

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 данной задачи я реализовал свою функцию для вычисления алгоритма:

```
function fib(n: number): numbe
  let temp = 0;
  let prev = 1;
  let current = 1;

  if (n === 1 || n === 2) {
    return 1;
  }

  for (let i = 3; i <= n; i++) {
    temp = current;
    current = prev + current;
    prev = temp;
```

```
  }
  return current;
}
```

В данном примере числа Фибоначчи вычисляются с использованием итеративного подхода. Функция fib(n) предназначена для нахождения n-го числа Фибоначчи и обеспечивает более эффективный расчет 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 последовательности. На каждой итерации переменная *prev* обновляется текущим значением, а *curr* становится суммой текущего и предыдущего чисел. После завершения цикла переменная *curr* содержит *n-е* число Фибоначчи.

Такой метод исключает лишние вычисления, характерные для рекурсии, и обладает линейной временной сложностью. Полный разбор алгоритма приведен в Приложении 3.

Для проверки работы алгоритма были выбраны несколько значений *n*. Для каждого из них вычислено соответствующее число Фибоначчи, а также измерено время выполнения в миллисекундах. Результаты тестов представлены ниже.

Значение	Результат	Время выполнения
5	5	33.82 мс
10	55	32.541 мс
15	610	32.541 мс
20	6765	35.387 мс
30	832040	36.128 мс

Результаты тестирования показывают, что использование цикла для вычисления n-го числа Фибоначчи является быстрым и эффективным решением для любых значений n. Этот подход обеспечивает стабильную работу даже при больших значениях, благодаря линейной сложности O(n). Скорость выполнения настолько

высокая, что для небольших значений n, как в тестах, время выполнения может быть округлено до 0 мс, что подтверждает высокую производительность алгоритма.

Задача №3

Дано целое число $1 \leq n \leq 40$, необходимо написать функцию fib(n) для вычисления nn-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 реализовал следующий алгоритм:

```
function fib(n: number): number[] {
  if (n < 0 || n > 40) {
    throw new Error('Значение должно быть от 1 до
  }

  const fibArray = [0, 1];

  for (let i = 2; i <= n; i++) {
    fibArray[i] = fibArray[i - 1] + fibArray[i - 2];
  }
```

```
  return fibArray;
}
```

В этом коде я вычислял последовательность Фибоначчи до n-го элемента включительно. Сначала создается массив fibArray длиной n+1, где первые два элемента равны 0 и 1, а остальные инициализируются нулями. Затем с помощью цикла, проходящего по индексам от 2 до n, вычисляется каждый элемент последовательности как сумма двух предыдущих. Для полного ознакомления с кодом можно обратиться к Приложению 4. Результаты тестирования моей реализации алгоритма:

Значение	Результат	Время выполнения
5	0, 1, 1, 2, 3, 5	54.263 мс
10	0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55	50.576 мс
15	0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610	44.494 мс

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.

$F(n) = (1 + \sqrt{5} 2) n - (1 - \sqrt{5} 2) n \sqrt{5}$

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.

Моя реализация данного алгоритма на языке JavaScript:

```
function fib(n: number): number {
 if (n < 1 || n > 64) {
   throw new Error("Значение должно быть не менее 1 и не более 6
 }
 const sqrt5 = Math.sqrt(5);
 const phi = (1 + sqrt5) / 2;
 const psi = (1 - sqrt5) / 2;

 const fibNumber = Math.round((Math.pow(phi, n) - Math.pow(psi,
sqrt5);
 return fibNumber;
}
```

В этом коде я вычисляю n-е число Фибоначчи с использованием формулы Бине. Сначала вычисляются два ключевых значения: r (золотое сечение) и i (обратное золотое сечение), которые рассчитываются через корень из 5. Затем число Фибоначчи вычисляется по формуле, где разность r^n и i^n делится на корень из 5. Результат округляется с помощью функции Math.round для минимизации погрешностей, связанных с вычислениями с плавающей точкой. Также добавлена проверка, чтобы значение n находилось в пределах от 1 до 64, и при выходе за эти

пределы выводилась ошибка. Для полного ознакомления с кодом можно обратиться к Приложению 5.

Результаты тестирования моей реализации алгоритма:

Значение	Результат	Время выполнения
5	5	44.404 мс
10	55	43.803 мс
15	610	44.454 мс
20	6765	63.901 мс

Исходя из результатов, можно утверждать, что 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 следует учитывать при использовании.

Задача №5:

Дано целое число $1 \leq n \leq 106$, 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 реализации данного алгоритма я написал свою функцию:

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

  let a = 0;
  let b = 1;

  for (let i = 2; i <= n; i++) {
    const next = (a + b) % 10;
    a = b;
    b = next;
  }

  const lastDigit = n === 1 ? a : b;
  const result = lastDigit % 2 === 0 ? 'even' : 'odd';

  return result;
}
```

В этом коде я определяю чётность n-го числа Фибоначчи, используя особенности последовательности по модулю 10. Сначала вычисляется остаток от деления n на 60, потому что последовательность чисел Фибоначчи по модулю 10 повторяется каждые 60 чисел. Затем, начиная с первых двух чисел Фибоначчи, выполняется

цикл для нахождения последней цифры n-го числа. В конце проверяется чётность этой цифры: если она чётная, возвращается "even", если нечётная — "odd". Для полного ознакомления с кодом можно обратиться к Приложению 6.

Результаты тестирования моей реализации алгоритма:

Значение	Результат	Время выполнения
5	odd	25.638 мс
10	odd	47.780 мс
15	even	47.900 мс
20	odd	56.447 мс

Из тестирования алгоритма можно сделать вывод, что он эффективно определяет чётность n-го числа Фибоначчи, не вычисляя само число. Используя свойства чисел Фибоначчи по модулю 10, алгоритм значительно сокращает количество вычислений. Это позволяет достичь временной сложности O(1), что делает решение быстрым и экономичным по времени.

Изучив задачи, связанные с числами Фибоначчи, я выделил несколько ключевых выводов, которые считаю наиболее важными. Во-первых, это эффективность алгоритмов. Я понял, что наиболее эффективными являются методы, использующие формулу Бине и определение чётности по модулю 10, так как они имеют временную сложность O(1) и не зависят от размера входных данных, что позволяет избежать лишних вычислений. Эти методы обеспечивают быстрые и точные результаты, что особенно важно при больших значениях n.

Во-вторых, стоит отметить рекурсивный подход. Хотя он интуитивно понятен, для больших значений n он оказывается слишком неэффективным из-за быстрого роста временной сложности. Этот метод подходит только для небольших значений n и его следует избегать при работе с большими числами.

В-третьих, стоит упомянуть оптимизацию использования памяти. Методы, которые включают хранение всей последовательности чисел Фибоначчи, могут быть полезны, если необходимо не только вычислить одно число, но и

использовать всю последовательность. Однако для вычисления конкретного числа это не самый эффективный способ, так как требует значительных затрат 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': 011016
'a': 1110
'e': 1111
'h': 0111
'm': 010
'n': 1000

'r': 110 's': 1001 't': 1010 'u': 001 0110110110111011011110000111001010111010000010100001111100110101011

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_encode.

Эта функция принимает 2 параметра:

value	строка, которую нужно закодировать.
filePath	строка, путь к файлу, где будут располагаться закодированные символы.

Результаты тестирования моей реализации данного метода на JavaScript:

Значение	Результат	Время выполнения
Hello world	11101111010110000001110010010011	68869 мс.
Lorem ipsum	11001101111011111010000001010011001101	108247 мс.

Хоть по условиям задачи и не требовалось адаптировать свою реализацию алгоритма, я решил сделать более практичную версию, работающую с любыми строками.

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 huffman_decode. Эта функция принимает 2 параметра:

value	Строка, закодированная ранее методом huffman_encode.
filePath	Строка, путь к файле с закодированными символами

Алгоритмы сортировки

В ходе изучения алгоритмов сортировки мы выделили несколько основных и значимых методов:

Сортировка пузырьком — это один из наиболее известных, но неэффективных алгоритмов. Он работает путем последовательного сравнения соседних элементов и их обмена местами, если предшествующий элемент больше следующего. Несмотря на свою популярность, на практике сортировка пузырьком редко применяется из-за своей низкой эффективности, особенно когда в конце массива находятся элементы с малыми значениями.

Шейкерная сортировка является усовершенствованной версией пузырьковой сортировки и выполняется в обоих направлениях. Алгоритм сначала движется слева направо, а затем справа налево, что способствует более быстрому устранению элементов, которые находятся не на своих местах.

Сортировка расчёской представляет собой оптимизированный вариант пузырьковой сортировки. В отличие от пузырьковой и шейкерной сортировок, которые сравнивают только соседние элементы, в сортировке расчёской используется изначально большое расстояние между сравниваемыми элементами, которое постепенно уменьшается до минимального. Это позволяет улучшить скорость сортировки, быстро устраняя «мелкие» элементы, расположенные в конце массива.

Сортировка вставками — это метод, при котором массив просматривается слева направо, и каждый новый элемент вставляется на соответствующее место между

ближайшими элементами с наименьшим и наибольшим значениями. Этот алгоритм довольно эффективен для сортировки небольших массивов.

Быстрая сортировка включает три этапа: выбор опорного элемента, перераспределение остальных элементов массива так, чтобы те, что меньше опорного, находились перед ним, а те, что больше или равны, — после. Затем эта процедура рекурсивно применяется к подмассивам слева и справа от опорного элемента. Быстрая сортировка считается одним из самых быстрых алгоритмов для работы с большими массивами.

Сортировка слиянием — это эффективный метод для сортировки данных, к которым доступ осуществляется последовательно (например, в потоках). Массив делится на две примерно равные части, каждая из которых сортируется отдельно, а затем два отсортированных подмассива объединяются в один отсортированный массив.

Пирамидальная сортировка начинается с создания пирамиды (или двоичной кучи) на базе элементов исходного массива. В этой структуре родительские элементы всегда больше своих дочерних. Затем из пирамиды извлекаются элементы, что позволяет отсортировать массив.

Поразрядная сортировка (Radix sort) делится на две категории: с сортировкой по младшим разрядам (LSD — least significant digit) и по старшим разрядам (MSD — most significant digit). В случае LSD, элементы сортируются по младшим цифрам (сначала те, что заканчиваются на 0, затем на 1 и так далее до 9). После этой сортировки они группируются по следующему разряду с конца, пока не будут обработаны все разряды. В сортировке MSD работа ведется по старшим разрядам.

Заключение

В ходе практической работы были исследованы разные алгоритмы, связанные с вычислением чисел Фибоначчи, алгоритмами сжатия Хаффмана и методами сортировки. Это исследование дало полное представление о данных алгоритмах, их эффективности и областях применения в различных ситуациях. Основные выводы работы заключаются в следующем:

1. **Вычисление чисел Фибоначчи**: Обзор различных методов, включая рекурсивные подходы и более быстрые алгоритмы, такие как матричное возведение в степень, показал, что выбор алгоритма зависит от требований к времени выполнения и доступной памяти. Итеративные методы и матричное вычисление демонстрируют лучшие результаты для больших чисел, обеспечивая оптимальное сочетание скорости и использования ресурсов.

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

3. **Алгоритмы сортировки**: Сравнение традиционных алгоритмов сортировки, таких как пузырьковая сортировка, сортировка вставками, быстрая сортировка и сортировка слиянием, показало, что простые методы, как пузырьковая сортировка, подходят лишь для небольших наборов данных. В то же время более сложные алгоритмы, такие как быстрая сортировка и сортировка слиянием, показывают высокую производительность при обработке больших массивов, что делает их предпочтительными в большинстве практических случаев.

В целом, эффективность алгоритмов во многом зависит от условий применения, структуры входных данных и таких ограничений, как время работы 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/MeldyTheCoder/algorithms_practicum

Приложение №2. Числа Фибоначчи - Задача №1

```
import { TFibMethod } from "../types"



function fib(n: number): number {
  if (n > 24) {
    throw new Error("Значение должно быть не больше 24.")
  } else if (n <= 1) {
    return n;
  } else {
    return fib(n - 1) + fib(n - 2);
  }
}



export default {
  label: "Фибоначчи | Рекурсивный метод",
  fn: fib,
  type: 'fib',
} as TFibMethod
```

Приложение №3. Числа Фибоначчи - Задание №2

```
import { TFibMethod } from "../types";



function fib(n: number): number {
  let temp = 0;
  let prev = 1;
  let current = 1;

  if (n > 32) {
    throw new Error("Значение должно быть не более 32.")
  }

  if (n === 1 || n === 2) {
    return 1;
  }

  for (let i = 3; i <= n; i++) {
    temp = current;
    current = prev + current;
    prev = temp;
  }
  return current;
}

export default {
  label: 'Фибоначчи | Метод с циклами',
  fn: fib,
  type: 'fib',
} as TFibMethod
```

Приложение №4. Числа Фибоначчи - Задание №3

```
import { TFibMethod } from "../types";


function fib(n: number): number[] {
  if (n < 0 || n > 40) {
    throw new Error('Значение должно быть от 1 до 40');
  }


  const fibArray = [0, 1];


  for (let i = 2; i <= n; i++) {
    fibArray[i] = fibArray[i - 1] + fibArray[i - 2];
  }


  return fibArray;
}


export default {
  label: "Фибоначе | Метод с массивами",
  fn: fib,
  type: 'fib',
} as TFibMethod
```

Приложение №5. Числа Фибоначчи - Задание №4

```
import { TFibMethod } from "../types";


function fib(n: number): number {
 if (n < 1 || n > 64) {
  throw new Error("Значение должно быть не менее 1 и не более 64");
 }
 const sqrt5 = Math.sqrt(5);
 const phi = (1 + sqrt5) / 2;
 const psi = (1 - sqrt5) / 2;


 const fibNumber = Math.round((Math.pow(phi, n) - Math.pow(psi, n)) / sqrt5);
 return fibNumber;
}


export default {
  label: "Фибоначчи | Метод Бине",
  fn: fib,
  type: 'fib',
} as TFibMethod
```

Приложение №6. Числа Фибоначчи - Задание №5

```
import { TFibMethod } from "../types";


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


  let a = 0;
  let b = 1;


  for (let i = 2; i <= n; i++) {
    const next = (a + b) % 10;
    a = b;
    b = next;
  }


  const lastDigit = n === 1 ? a : b;
  const result = lastDigit % 2 === 0 ? 'even' : 'odd';


  return result;
}


export default {
  label: "Фибоначчи | Проверка четности числа",
  fn: fib_eo,
  type: 'fib',
} as TFibMethod
```

Приложение №7. Задача по алгоритму Хаффмана №1

```
import { terminal } from "terminal-kit";
import { CodesType, THuffmanMethod } from "../types";
import {writeFileSync} from 'fs';


type HuffmanNode = {
  char: string | null;
  freq: number;
  left?: HuffmanNode;
  right?: HuffmanNode;
};


function buildHuffmanTree(input: string): HuffmanNode {
  const freqMap: { [key: string]: number } = {};


  for (const char of input) {
    freqMap[char] = (freqMap[char] || 0) + 1;
  }


  const nodes: HuffmanNode[] = Object.entries(freqMap).map(([char, freq]) => ({
    char,
    freq,
  }));


  // Построение дерева Хаффмана
  while (nodes.length > 1) {
    nodes.sort((a, b) => a.freq - b.freq);


    const left = nodes.shift()!;
    const right = nodes.shift()!;
    nodes.push({ char: null, freq: left.freq + right.freq, left, right });
  }


  return nodes[0];
}


function generateCodes(node: HuffmanNode, prefix = '', codes: CodesType = {}): CodesType {
  if (node.char !== null) {
    codes[node.char] = prefix;
  } else {
    if (node.left) generateCodes(node.left, prefix + '0', codes);
    if (node.right) generateCodes(node.right, prefix + '1', codes);
  }
  return codes;
}


function huffman_encode(input: string, filePath: string): string {
  const tree = buildHuffmanTree(input);
  const codes = generateCodes(tree);


  writeFileSync(filePath, JSON.stringify(codes))
  terminal
    .cyan.bold(`\n[!] Таблица кодов записана в `)
    .white(filePath)
    .cyan.bold('.')


  let encoded = '';
  for (const char of input) {
```

```
    encoded += codes[char];
  }

  const encodedLength = (new TextEncoder()).encode(encoded).length;

  terminal.yellow.bold(`\n${Object.values(codes).length} ${encodedLength}\n`)
  terminal.grey.bold(Object.entries(codes).map(
    ([symbol, code]) => `'${symbol}': ${code}`).join('\n'),
  )
  terminal.green.bold(encoded);

  return encoded;
}

export default {
  label: "Хаффман | Кодирование",
  fn: huffman_encode,
  type: 'huffman',
} as THuffmanMethod;
```

Приложение №8. Задача по алгоритму Хаффмана №2

```
import { readFileSync, existsSync } from "fs";
import { CodesType, THuffmanMethod } from "../types";
import { terminal } from "terminal-kit";


function huffman_decode(encodedString: string, filePath: string): string {
  if (!existsSync(filePath)) {
    throw new Error(`Запрашиваемый Вами файл "${filePath}" не найден.`)
  }


  const fileData = readFileSync(filePath, {encoding: 'utf-8'});
  if (!fileData) {
    throw new Error(
      `
        Для того, чтобы использовать данный метод,
        сначала закодируйте любую строку, чтобы информация в файле "${filePath}" изменилась.
      `
    )
  }


  const codes: CodesType = JSON.parse(fileData);


  terminal.yellow.bold("\nНайденная таблица кодов:\n");
  terminal.table(
    [
      ['Символ', 'Код'],
      ...Object.entries(codes)
    ],
    {
      fit: true,
      contentHasMarkup: true,
      firstRowTextAttr: {
        bgColor: 'grey',
      },
      width: 30,
    }


  )


  const reversedCodes = Object.fromEntries(
    Object.entries(codes).map(([char, code]) => [code, char])
  );


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


  for (const bit of encodedString) {
    buffer += bit;
    if (reversedCodes[buffer]) {
      decodedString += reversedCodes[buffer];
      buffer = "";
    }
  }


  return decodedString;
}
```

```ts
export default {
  label: "Хаффман | Декодирование",
  fn: huffman_decode,
  type: 'huffman',
} as THuffmanMethod
```