[Год]

[Название документа]

[ПОДЗАГОЛОВОК ДОКУМЕНТА]

GC0544

Содержание

Введение

Целью данной работы является изучение и анализ алгоритмов, связанных с вычислением чисел Фибоначчи, алгоритмов Хаффмана и алгоритмов сортировки. Эти алгоритмы имеют широкое применение в информатике, от задач оптимизации и кодирования данных до базовых операций обработки информации. Основной задачей является исследование вычислительной сложности данных алгоритмов и сравнительный анализ их эффективности в различных условиях.

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

Изучены основные методы вычисления чисел Фибоначчи, включая рекурсивный подход, использование 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 пузырьковая сортировка, сортировка вставками, сортировка и сортировка слиянием, с упором на их временную и пространственную сложность.

Выполнен сравнительный анализ исследованных алгоритмов, включая оценку их сложности и эффективности в различных 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

Подходя к практике на первом занятии, мы познакомились с понятием вычислительной сложности. Мы рассмотрели различные виды сложностей, изучили их особенности, а также познакомились с основными нотациями, которые используются для их описания.

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:

"Чистка ковра пылесосом" требует времени, которое зависит от его площади. Если площадь ковра увеличивается в два раза, то и время на его уборку возрастает пропорционально, также в два раза. Этот процесс имеет линейную сложность O(n), где n — площадь ковра.

"Поиск имени в телефонной книге", упорядоченной по алфавиту, осуществляется с помощью бинарного поиска. Каждый шаг в этом примере уменьшает количество оставшихся страниц вдвое, что существенно ускоряет процесс. В результате даже в книге с огромным количеством страниц имя можно найти за небольшое количество шагов благодаря логарифмической сложности O(logn).

Рассмотрение входных данных большого размера и оценка порядка роста времени работы алгоритма приводят к понятию асимптотической сложности алгоритма. При этом алгоритм с меньшей асимптотической сложностью является более эффективным для всех входных данных, за исключением лишь, возможно, данных малого размера.

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

Нотация O большое — это математическая нотация, которая описывает ограничивающее поведение функции, когда аргумент стремится к определенному значению или бесконечности. O 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 «O большое» существуют и другие: «o малое», «Омега» и «Тета»

- O большое ($O(n)$) описывает верхнюю границу сложности, то есть наихудший случай.
- o малое ($o(n)$) описывает верхнюю границу, исключая точную оценку, то есть только порядок наихудшего случая.
- Омега ($\Omega(n)$) описывает нижнюю границу сложности, то есть наилучший случай.
- Тета ($\Theta(n)$) описывает точную оценку сложности, то есть оценку сложности с учетом особенностей входных данных.

Исходя из всего перечисленного можно сказать, что нотация O-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 Python, реализованных в виде кода, с использованием различных методов и библиотек. 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:

```
def fib(n):

if n == 0:

return 0

elif n == 1:

return 1

else:

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

В этом коде выполняется вычисление чисел Фибоначчи с помощью рекурсивной функции и замеряет время, необходимое для их вычисления. В функции fib(n) реализован рекурсивный алгоритм: если n=0, функция возвращает 0, если n=1, возвращает 1. Для всех других значений n функция вызывает сама себя для (n−1) и (n−2), складывая результаты. 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) = 5, time: 0.00 ms
fib(10) = 55, time: 0.02 ms
fib(15) = 610, time: 0.18 ms
fib(20) = 6765, time: 1.96 ms
fib(24) = 46368, time: 13.58 ms
```

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:

```
def fib(n):

    if n == 0:

        return 0

    elif n == 1:

        return 1

    prev, curr = 1, 1

    for _ in range(2, n):

        prev, curr = curr, prev + curr

    return curr
```

В этом коде вычисляется числа Фибоначчи с использованием итеративного подхода и измеряет время выполнения для заданных значений. Функция fib(n) возвращает n-е число Фибоначчи. Если n = 0, возвращается 0, а если n = 1, возвращается 1. Для значений n ≥ 2 используется цикл, где две переменные, prev и curr, хранят два последних числа Фибоначчи. В каждой итерации обновляются значения: prev становится текущим числом, а curr — суммой предыдущего и текущего чисел. После завершения цикла в переменной curr оказывается n-е число Фибоначчи. 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) = 5, time: 0.005 ms

fib(10) = 55, time: 0.004 ms

fib(15) = 610, time: 0.002 ms

fib(20) = 6765, time: 0.002 ms

fib(32) = 2178309, time: 0.002 ms
```

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 ≤ n ≤ 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:

```
def fib(n):
    fib_sequence = [0, 1] + [0] * (n - 1)
    for i in range(2, n + 1):
      fib_sequence[i] = fib_sequence[i - 1] + fib_sequence[i - 2]
    return fib_sequence
```

В этом коде я ты вычислял последовательность Фибоначчи до n-го элемента включительно. Сначала создается список fib_sequence 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:

```
def fib(n):

    r = (1 + math.sqrt(5)) / 2

    i = (1 - math.sqrt(5)) / 2

    fibonacci_number = (math.pow(r, n) - math.pow(i, n)) /
math.sqrt(5)
```

```
    return round(fibonacci_number)

n = int(input("Введите число n: "))
```

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 для устранения ошибок, связанных с вычислениями с плавающей точкой. 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=8 и вызывалась функцию для вычисления числа через Бине и вывелся определенный результат:

```
Введите число n: 8

Число Фибоначчи F (8) равно: 21
```

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:

```
def fib_eo(n):

    d = 60

    n = n % d

    a, b = 1, 1

    for _ in range (n - 1):

        a, b = b, (a + b) % 10

    return "even" if b % 2 == 0 else "odd"
```

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

01101101101110110111100001110010101110100000101000011111001
10101011
```

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 она как раз и является для декодирования строки. Эта функция принимает 4 параметра:

symbol_count количество различных символов, используемых в коде (например, 12). В данной функции он не используется, но это параметр для контекста. encoded_size длина закодированной строки в битах (например, 60). В данной функции также не используется.

codes словарь, где ключ - символ, а значение - его Хаффман-код (например, ' ': '1011').

encoded_string строка из битов, которая была закодирована с использованием алгоритма Хаффмана. Данная функция huffman_decode возвращает раскодированную строку, преобразуя её из битового представления.

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_decoding. Эта функция принимает 4 параметра:
symbol_count: количество различных символов, используемых в кодировке (например, 12). В данной функции он не используется, но помогает описать контекст.
encoded_size: длина закодированной строки в битах (например, 60). Также не используется в функции, но предоставляет дополнительную информацию о данных.
codes: словарь, где ключ - символ, а значение — его Хаффман-код.
encoded_string строка, содержащая последовательность битов, которые необходимо декодировать.

Алгоритмы сортировки

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

Сортировка пузырьком — один из самых известных алгоритмов сортировки. Здесь нужно последовательно сравнивать значения соседних элементов и менять числа местами, если предыдущее оказывается больше последующего. Этот алгоритм считается учебным и почти не применяется на практике из-за низкой эффективности: он медленно работает на тестах, в которых маленькие элементы (их называют «черепахами») стоят в конце массива.

Шейкерная сортировка отличается от пузырьковой тем, что она двунаправленная: алгоритм перемещается не строго слева направо, а сначала слева направо, затем справа налево.

Сортировка расчёской — улучшение сортировки пузырьком. Её идея состоит в том, чтобы «устранить» элементы с небольшими значения в конце массива, которые замедляют работу алгоритма. Если при пузырьковой и шейкерной сортировках при переборе массива сравниваются соседние элементы, то при «расчёсывании» сначала берётся достаточно большое расстояние между сравниваемыми значениями, а потом оно сужается вплоть до минимального.

Сортировке вставками массив постепенно перебирается слева направо. При этом каждый последующий элемент размещается так, чтобы он оказался между ближайшими элементами с минимальным и максимальным значением.

Быстрая сортировка этот алгоритм состоит из трёх шагов. Сначала из массива нужно выбрать один элемент — его обычно называют опорным. Затем другие элементы в массиве перераспределяют так, чтобы элементы меньше опорного оказались до него, а большие или равные — после. А дальше рекурсивно применяют первые два шага к подмассивам справа и слева от опорного значения.

Сортировка слиянием пригодится для таких структур данных, в которых доступ к элементам осуществляется последовательно (например, для потоков).

Здесь массив разбивается на две примерно равные части, и каждая из них сортируется по отдельности. Затем два отсортированных подмассива сливаются в один.

Пирамидальная сортировка при этой сортировке сначала строится пирамида из элементов исходного массива. Пирамида (или двоичная куча) — это способ представления элементов, при котором от каждого узла может отходить не больше двух ответвлений. А значение в родительском узле должно быть больше значений в его двух дочерних узлах.

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 рамках практики были изучены разные виды алгоритмов, связанные с вычислением чисел Фибоначчи, алгоритмами Хаффмана и алгоритмами сортировки. Проведенное исследование позволило получить всестороннее представление об этих алгоритмах, их эффективности и применимости в различных условиях.

Основные выводы по итогам работы:

Анализ различных подходов к вычислению чисел Фибоначчи, от рекурсивного метода до быстрого возведения матриц в степень, показал, что выбор алгоритма существенно зависит от требований к скорости выполнения и доступной памяти. Итеративные алгоритмы и метод матриц демонстрируют наилучшую производительность при работе с большими значениями, обеспечивая оптимальное соотношение скорости и ресурсоемкости.

Алгоритм Хаффмана показал свою эффективность в задачах кодирования данных. Построение дерева Хаффмана и использование префиксных кодов позволяют значительно минимизировать размер закодированной информации, что делает данный алгоритм ключевым инструментом для задач сжатия данных и оптимизации их представления.

Сравнение классических алгоритмов сортировки, таких как пузырьковая сортировка, сортировка вставками, быстрая сортировка и сортировка слиянием, показало, что простейшие методы, например пузырьковая сортировка, применимы только для небольших объемов данных. В то же время более сложные алгоритмы, такие как быстрая сортировка и сортировка слиянием, демонстрируют высокую производительность при работе с большими наборами данных, что делает их предпочтительным выбором в большинстве практических случаев.

В общем в анализе сама эффективность алгоритмов во многом определяется условиями их применения, структурой входных данных и ограничениями на ресурсы, включая время выполнения и объем используемой памяти.

Список источников

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

2. Обозначение «Большое O» [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/boychik2004/algorithms_practicum

Приложение 2. Числа Фибоначчи Задача №1

```
import time
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
def measure_execution_time():
    test_values = [5, 10, 15, 20, 24]
    for n in test_values:
        start_time = time.perf_counter()
        result = fib(n)
        end_time = time.perf_counter()
        elapsed_time = (end_time - start_time) * 1000
        print(f"fib({n}) = {result}, time: {elapsed_time:.2f} ms")

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

Приложение 3. Числа Фибоначчи Задача №2

```
import time
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    prev, curr = 1, 1
    for _ in range(2, n):
        prev, curr = curr, prev + curr
    return curr
def measure_execution_time():
    test_values = [5, 10, 15, 20, 32]
    for n in test_values:
        start_time = time.perf_counter()
        result = fib(n)
        end_time = time.perf_counter()
        elapsed_time = (end_time - start_time) * 1000
        print(f"fib({n}) = {result}, time: {elapsed_time:.3f} ms")
if __name__ == "__main__":
    measure_execution_time()
```

Приложение 4. Числа Фибоначчи Задача №3

```
def fib(n):
    fib_sequence = [0, 1] + [0] * (n - 1)
    for i in range(2, n + 1):
        fib_sequence[i] = fib_sequence[i - 1] + fib_sequence[i - 2]
    return fib_sequence
if __name__ == "__main__":
    n = 8
    result = fib(n)
    print(result)
```

Приложение 5. Числа Фибоначчи Задача №4

```
import math

def fib(n):

    r = (1 + math.sqrt(5)) / 2

    i = (1 - math.sqrt(5)) / 2

    fibonacci_number = (math.pow(r, n) - math.pow(i, n)) / 
math.sqrt(5)


    return round(fibonacci_number)

n = int(input("Введите число n: "))


if __name__ == "__main__":

   print(f"Число Фибоначчи F({n}) равно: {fib(n)}")
```

Приложение 6. Числа Фибоначчи Задача №5

```
def fib_eo(n):
    d = 60
    n = n % d
    a, b = 1, 1
    for _ in range(n - 1):
        a, b = b, (a + b) % 10
    return "even" if b % 2 == 0 else "odd"
if __name__ == "__main__":
    n = int(input("Введите число n: "))
    print(fib_eo(n))
```

Приложение 7. Задачи по алгоритмам Хаффмана №1

```
def huffman_decode(symbol_count, encoded_size, codes, 
encoded_string):
    code_to_symbol = {code: symbol for symbol, code in 
codes.items()}
    decoded_string = ""
    buffer = ""
    for bit in encoded_string:
        buffer += bit
        if buffer in code_to_symbol:
            decoded_string += code_to_symbol[buffer]
            buffer = ""
    return decoded_string


if __name__ == "__main__":
    symbol_count = 12
    encoded_size = 60
    codes = {
        ' ': '1011',
        '.': '1110',
        'D': '1000',
        'c': '000',
        'd': '001',
        'e': '1001',
        'i': '010',
        'm': '1100',
        'n': '1010',
        'o': '1111',
        's': '011',
        'u': '1101'
    }
    encoded_string = 
"100011110001001101000111111011001010011000010110011010111110"
```

```
    decoded_string = huffman_decode(symbol_count, encoded_size, codes, encoded_string)

    print(decoded_string)
```

Приложение 8. Задачи по алгоритмам Хаффмана №2

```
def huffman_decode(symbol_count, encoded_size, codes,
encoded_string):

    code_to_symbol = {code: symbol for symbol, code in
codes.items()}

    decoded_string = ""

    buffer = ""

    for bit in encoded_string:

        buffer += bit

        if buffer in code_to_symbol:

            decoded_string += code_to_symbol[buffer]

            buffer = ""

    return decoded_string


symbol_count = 12

encoded_size = 60

codes = {

    ' ': '1011',

    '.': '1110',

    'D': '1000',

    'c': '000',

    'd': '001',

    'e': '1001',

    'i': '010',

    'm': '1100',

    'n': '1010',

    'o': '1111',

    's': '011',

    'u': '1101'

}

encoded_string =
"100011110001001101000111111011001010011000010110011010111110"
```

```
decoded_string = huffman_decode(symbol_count, encoded_size, codes,
encoded_string)

print(decoded_string)
```

Отчет о проверке № 9064626

Дата выгрузки: 2024-12-25 16:08:59
Пользователь: maximstroev7777@mail.ru, ID: 9064626

Отчет предоставлен сервисом «Антиплагиат»
на сайте antiplagius.ru/

Информация о документе

№ документа: 9064626
Имя исходного файла: УП.02 - Строев М- отчет.pdf
Размер файла: 0.44 МБ
Размер текста: 23954
Слов в тексте: 3862
Число предложений: 254

Информация об отчете

Дата: 2024-12-25 16:08:59 - Последний готовый отчет
Оценка оригинальности: 88%
Заимствования: 12%

88.60% 11.4%

Источники:

Доля в тексте	Ссылка
79.80%	https://full-arts.ru/blog/algoritmy-kotorye-dolzhen-znat-kazhdyj...
61.80%	https://multiurok.ru/files/vidy-sortirovok.html
17.00%	https://education.yandex.ru/journal/osnovnye-vidy-sortirovok-i-p...
14.70%	https://habr.com/ru/articles/782608/
12.80%	https://tproger.ru/articles/algoritmy-sortirovki-na-java-s-prime...
12.80%	https://www.yuripetrov.ru/edu/python/ch_06_01.html
12.70%	https://proglib.io/p/sort-gif
12.70%	https://practice.keyfire.ru/blog/kak-zakodirovat-stroku-s-pomosh...
11.90%	https://spravochnick.ru/programmirovanie/algoritmy_i_analiz_sloz...
11.40%	https://bankami.ru/chto-znachit-zakaz-v-centre-sortirovki
10.10%	https://pcnews.ru/blogs/izucenie_izvestnye_algoritmy_sortirovok-...
8.80%	https://blog.underpowered.net/algorithms/algoritmy-teoriya-prakt...

Информация о документе:

[Год] [Название документа] [ПОДЗАГОЛОВОК ДОКУМЕНТА] GC0544 Содержание Введение 3 Вычислительная сложность 4 Алгоритмы для чисел Фибоначчи 6 Задачи по алгоритмам Хаффмана 15 Алгоритмы сортировки 19 Заключение 21 Список источников 23 Приложение 1. QR код 24 Приложение 2. Числа Фибоначчи Задача №1 25 Приложение 3. Числа Фибоначчи Задача №2 26 Приложение 4. Числа Фибоначчи Задача №3 27 Приложение 5. Числа Фибоначчи Задача №4 28 Приложение 6. Числа

Фибоначчи Задача №5 29 Приложение 7. Задачи по алгоритмам Хаффмана №1 30 Приложение 8. Задачи по алгоритмам Хаффмана №2 32 Введение Целью данной работы является изучение и анализ алгоритмов, связанных с вычислением чисел Фибоначчи, алгоритмов Хаффмана и алгоритмов сортировки. Эти алгоритмы имеют широкое применение в информатике, от задач оптимизации и кодирования данных до базовых операций обработки информации. Основной задачей является исследование вычислительной сложности данных алгоритмов и сравнительный анализ их эффективности в различных условиях. Для достижения поставленной цели в процессе работы были выполнены следующие задачи: Изучены основные методы вычисления чисел Фибоначчи, включая рекурсивный подход, использование 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 пузырьковая сортировка, сортировка вставками, сортировка и сортировка слиянием, с упором на их временную и пространственную сложность. Выполнен сравнительный анализ исследованных алгоритмов, включая оценку их сложности и эффективности в различных 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 Подходя к практике на первом занятии, мы познакомились с понятием вычислительной сложности. Мы рассмотрели различные виды сложностей, изучили их особенности, а также познакомились с основными нотациями, которые используются для их описания. 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: "Чистка ковра пылесосом" требует времени, которое зависит от его площади. Если площадь ковра увеличивается в два раза, то и время на его уборку возрастает пропорционально, также в два раза. Этот процесс имеет линейную сложность O(n), где n - площадь ковра. "Поиск имени в телефонной книге", упорядоченной по алфавиту, осуществляется с помощью бинарного поиска. Каждый шаг в этом примере уменьшает количество оставшихся страниц вдвое, что существенно ускоряет процесс. В результате даже в книге с огромным количеством страниц имя можно найти за небольшое количество шагов благодаря логарифмической сложности O(logn). Рассмотрение входных данных большого размера и оценка порядка роста времени работы алгоритма приводят к понятию асимптотической сложности алгоритма. При этом алгоритм с меньшей асимптотической сложностью является более эффективным для всех входных данных, за исключением лишь, возможно, данных малого размера. Наиболее популярной нотацией для описания вычислительной сложности алгоритмов является "O большое" (Big O Notation или просто Big O). Нотация O большое - это математическая нотация, которая описывает ограничивающее поведение функции, когда аргумент стремится к определенному значению или бесконечности. O 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 "O большое" существуют и другие: "о малое", "Омега" и "Тета" • O большое (O(n)) описывает верхнюю границу сложности, то есть наихудший случай. • о малое (o(n)) описывает верхнюю границу, исключая точную оценку, то есть только порядок наихудшего случая. • Омега (Ω(n)) описывает нижнюю границу сложности, то есть наилучший случай. • Тета (Θ(n)) описывает точную оценку сложности, то есть оценку сложности с учетом особенностей входных данных. Исходя из всего перечисленного можно сказать, что нотация O-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 Python, реализованных в виде кода, с использованием различных методов и библиотек. В каждом случае были выбраны оптимальные инструменты для достижения целей и оценки производительности. Задача №1 Дано целое число 1 ≤ n ≤ 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: def fib(n): if n == 0: return 0 elif n == 1: return 1 else: return fib (n - 1) + fib(n - 2) В этом коде выполняется вычисление чисел Фибоначчи с помощью рекурсивной функции и замеряет время, необходимое для их вычисления. В функции fib(n) реализован рекурсивный алгоритм: если n=0, функция возвращает 0, если n=1, возвращает 1. Для всех других значений n функция вызывает сама себя для (n−1) и (n−2), складывая результаты. 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) = 5, time: 0.00 ms fib(10) = 55, time: 0.02 ms fib(15) = 610, time: 0.18 ms fib(20) = 6765, time: 1.96 ms fib(24) = 46368, time: 13.58 ms 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 ≤ n ≤ 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: def fib(n): if n == 0: return 0 elif n == 1: return 1 prev, curr = 1, 1 for _ in range(2, n): prev, curr = curr, prev + curr return curr В этом коде вычисляется числа Фибоначчи с использованием итеративного подхода и измеряет время выполнения для заданных значений. Функция fib(n) возвращает n-е число Фибоначчи. Если n = 0, возвращается 0, а если n = 1, возвращается 1. Для значений n ≥ 2 используется цикл, где две переменные, prev и curr, хранят два последних числа Фибоначчи. В каждой итерации обновляются значения: prev становится текущим числом, а curr - суммой предыдущего и текущего чисел. После завершения цикла в переменной curr оказывается n-е число Фибоначчи. 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) = 5, time: 0.005 ms fib(10) = 55, time: 0.004 ms fib(15) = 610, time: 0.002 ms fib(20) = 6765, time: 0.002 ms fib(32) = 2178309, time: 0.002 ms 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 ≤ n ≤ 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 написал алгоритм: def fib(n): fib_sequence = [0, 1] + [0] * (n - 1) for i in range(2, n + 1): fib_sequence[i] = fib_sequence[i - 1] + fib_sequence[i - 2] return fib_sequence В этом коде я ты вычислял последовательность Фибоначчи до n-го элемента включительно. Сначала создается список fib_sequence 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≤ n ≤64, необходимо написать функцию fib(n) для вычисления nго числа Фибоначчи. Функция fib(n) должна производить вычисление по формуле Бине. □ □ □ 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: def fib(n): r = (1 + math.sqrt(5)) / 2 i = (1 - math.sqrt(5)) / 2 fibonacci_number = (math.pow(r, n) - math.pow(i, n)) / math.sqrt(5) return round(fibonacci_number) n = int(input("Введите число n: ")) 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 для устранения ошибок, связанных с вычислениями с плавающей точкой. 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=8 и вызывалась функцию для вычисления числа через Бине и вывелся определенный результат: Введите число n: 8 Число Фибоначчи F (8) равно: 21 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≤□≤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 ≤ a, b ≤ 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: def fib_eo(n): d = 60 n = n % d a, b = 1, 1 for _ in range (n - 1): a, b = b, (a + b) % 10 return "even" if b % 2 == 0 else "odd" 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 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_decode она как раз и является для декодирования строки. Эта функция принимает 4 параметра: symbol_count количество различных символов, используемых в коде (например, 12). В данной функции он не используется, но это параметр для контекста. encoded_size длина закодированной строки в битах (например, 60). В данной функции также не используется. codes словарь, где ключ - символ, а значение - его Хаффман-код (например, ' ': '1011'). encoded_string строка из битов, которая была закодирована с использованием алгоритма Хаффмана. Данная функция huffman_decode возвращает раскодированную строку, преобразуя её из битового представления. 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_decoding. Эта функция принимает 4 параметра: symbol_count: количество различных символов, используемых в кодировке (например, 12). В данной функции он не используется, но помогает описать контекст. encoded_size: длина закодированной строки в битах (например, 60). Также не используется в функции, но предоставляет дополнительную информацию о данных. codes: словарь, где ключ - символ, а значение - его Хаффман-код. encoded_string строка, содержащая последовательность битов, которые необходимо декодировать. Алгоритмы сортировки На практике, рассматривая тему алгоритмов сортировки, мы выделили основные и наиболее важные из них: Сортировка пузырьком - один из самых известных алгоритмов сортировки. Здесь нужно последовательно сравнивать значения соседних элементов и менять числа местами, если предыдущее оказывается больше последующего. Этот алгоритм считается учебным и почти не применяется на практике из-за низкой эффективности: он медленно работает на тестах, в которых маленькие элементы (их называют "черепахами") стоят в конце массива. Шейкерная сортировка отличается от пузырьковой тем, что она двунаправленная: алгоритм перемещается не строго слева направо, а сначала слева направо, затем справа налево. Сортировка расчёской - улучшение сортировки пузырьком. Её идея состоит в том, чтобы "устранить" элементы с небольшими значения в конце массива, которые замедляют работу алгоритма. Если при пузырьковой и шейкерной сортировках при переборе массива сравниваются соседние элементы, то при "расчёсывании" сначала берётся достаточно большое расстояние между сравниваемыми значениями, а потом оно сужается вплоть до минимального. Сортировке вставками массив постепенно перебирается слева направо. При этом каждый последующий элемент размещается так, чтобы он оказался между ближайшими элементами с минимальным и максимальным значением. Быстрая сортировка этот алгоритм состоит из трёх шагов. Сначала из массива нужно выбрать один элемент - его обычно называют опорным. Затем другие элементы в массиве перераспределяют так, чтобы элементы меньше опорного оказались до него, а большие или равные - после. А дальше рекурсивно применяют первые два шага к подмассивам справа и слева от опорного значения. Сортировка слиянием пригодится для таких структур данных, в которых доступ к элементам осуществляется последовательно (например, для потоков). Здесь массив разбивается на две примерно равные части, и каждая из них сортируется по отдельности. Затем два отсортированных подмассива сливаются в один. Пирамидальная сортировка при этой сортировке сначала строится пирамида из элементов исходного массива. Пирамида (или двоичная куча) - это способ представления элементов, при котором от каждого узла может отходить не больше двух ответвлений. А значение в родительском узле должно быть больше значений в его двух дочерних узлах. 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 рамках практики были изучены разные виды алгоритмов, связанные с вычислением чисел Фибоначчи, алгоритмами Хаффмана и алгоритмами сортировки. Проведенное исследование позволило получить всестороннее представление об этих алгоритмах, их эффективности и применимости в различных условиях. Основные выводы по итогам работы: Анализ различных подходов к вычислению чисел Фибоначчи, от рекурсивного метода до быстрого возведения матриц в степень, показал, что выбор алгоритма существенно зависит от требований к скорости выполнения и доступной памяти. Итеративные алгоритмы и метод матриц демонстрируют наилучшую производительность при работе с большими значениями, обеспечивая оптимальное соотношение скорости и ресурсоемкости. Алгоритм Хаффмана показал свою эффективность в задачах кодирования данных. Построение дерева Хаффмана и использование префиксных кодов позволяют значительно минимизировать размер закодированной информации, что делает данный алгоритм ключевым инструментом для задач сжатия данных и оптимизации их представления. Сравнение классических алгоритмов сортировки, таких как пузырьковая сортировка, сортировка вставками, быстрая сортировка и сортировка слиянием, показало, что простейшие методы, например пузырьковая сортировка, применимы только для небольших объемов данных. В то же время более сложные алгоритмы, такие как быстрая сортировка и сортировка слиянием, демонстрируют высокую производительность при работе с большими наборами данных, что делает их предпочтительным выбором в большинстве практических случаев. В общем в анализе сама эффективность алгоритмов во многом определяется условиями их применения, структурой входных данных и ограничениями на ресурсы, включая время выполнения и объем используемой 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/boychik2004/algorithms_practicum Приложение 2 Числа Фибоначчи Задача №1 import time def fib(n): if n == 0: return 0 elif n == 1: return 1 else: return fib(n - 1) + fib(n - 2) def measure_execution_time(): test_values = [5, 10, 15, 20, 24] for n in test_values: start_time = time.perf_counter() result = fib(n) end_time = time.perf_counter() elapsed_time = (end_time - start_time) * 1000 print(f"fib({n}) = {result}, time: {elapsed_time:.2f} ms") if __name__ == "__main__": measure_execution_time() Приложение 3 Числа Фибоначчи Задача №2 import time def fib(n): if n == 0: return 0 elif n == 1: return 1 prev, curr = 1, 1 for _ in range(2, n): prev, curr = curr, prev + curr return curr def measure_execution_time(): test_values = [5, 10, 15, 20, 32] for n in test_values: start_time = time.perf_counter() result = fib(n) end_time = time.perf_counter() elapsed_time = (end_time - start_time) * 1000 print(f"fib({n}) = {result}, time: {elapsed_time:.3f} ms") if __name__ == "__main__": measure_execution_time() Приложение 4 Числа Фибоначчи Задача №3 def fib(n): fib_sequence = [0, 1] + [0] * (n - 1) for i in range(2, n + 1): fib_sequence[i] = fib_sequence[i - 1] + fib_sequence[i - 2] return fib_sequence if __name__ == "__main__": n = 8 result = fib(n) print(result) 5 Числа Фибоначчи Задача №4 import math def fib(n): r = (1 + math.sqrt(5)) / 2 i = (1 - math.sqrt(5)) / 2 fibonacci_number = (math.pow(r, n) - math.pow(i, n)) / math.sqrt(5) return round(fibonacci_number) n = int(input("Введите число n: ")) if __name__ == "__main__": print(f"Число Фибоначчи F({n}) равно: {fib(n)}") 6 Числа Фибоначчи Задача №5 def fib_eo(n): d = 60 n = n % d a, b = 1, 1 for _ in range(n - 1): a, b = b, (a + b) % 10 return "even" if b % 2 == 0 else "odd" if __name__ == "__main__": n = int(input("Введите число n: ")) print(fib_eo(n)) 7 Задачи по алгоритмам Хаффмана №1 def huffman_decode(symbol_count, encoded_size, codes, encoded_string): code_to_symbol = {code: symbol for symbol, code in codes.items()} decoded_string = "" buffer = "" for bit in encoded_string: buffer += bit if buffer in code_to_symbol: decoded_string += code_to_symbol[buffer] buffer = "" return decoded_string if __name__ == "__main__": symbol_count = 12 encoded_size = 60 codes = { ' ': '1011', '.': '1110', 'D': '1000', 'c': '000', 'd': '001', 'e': '1001', 'i': '010', 'm': '1100', 'n': '1010', 'o': '1111', 's': '011', 'u': '1101' } encoded_string = "10001111000100110100011111101100101001100001011001101011110" decoded_string = huffman_decode(symbol_count, encoded_size, codes, encoded_string) print(decoded_string) Приложение 8. Задачи по алгоритмам Хаффмана №2 def huffman_decode(symbol_count, encoded_size, codes, encoded_string): code_to_symbol = {code: symbol for symbol, code in codes.items()} decoded_string = "" buffer = "" for bit in encoded_string: buffer += bit if buffer in code_to_symbol: decoded_string += code_to_symbol[buffer] buffer = "" return decoded_string symbol_count = 12 encoded_size = 60 codes = { ' ': '1011', '.': '1110', 'D': '1000', 'c': '000', 'd': '001', 'e': '1001', 'i': '010', 'm': '1100', 'n': '1010', 'o': '1111', 's': '011', 'u': '1101' } encoded_string = "10001111000100110100011111101100101001100001011001101011110" decoded_string = huffman_decode(symbol_count, encoded_size, codes, encoded_string) print(decoded_string) 2 2 . . 2 2 . 2 Приложение . Приложение . 2 2 Приложение . 2 2 2 2