Отчет о проверке № 9058520

Дата выгрузки: 2024-12-23 09:29:57
Пользователь: okolomarsa@gmail.com, ID: 9058520

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

Информация о документе

№ документа: 9058520
Имя исходного файла: УП.02 - Марупов Р. А. - отчет.pdf
Размер файла: 0.53 МБ
Размер текста: 26765
Слов в тексте: 4161
Число предложений: 380

Информация об отчете

Дата: 2024-12-23 09:29:57 - Последний готовый отчет
Оценка оригинальности: 91%
Заимствования: 9%

91.00% 9%

Источники:

Доля в тексте	Ссылка
21.60%	https://metal-archive.ru/stati/33884-vychislitelnaya-slozhnost.h...
19.30%	https://habr.com/ru/companies/gnivc/articles/683128/
14.60%	https://ru.ruwiki.ru/wiki/%D0%92%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D...
14.50%	https://sky.pro/wiki/python/algoritm-fibonachchi-na-python-posha...
11.00%	https://videouroki.net/razrabotki/primienieniie-chislovogho-riad...
9.90%	https://skillbox.ru/media/code/chisla-fibonachchi-dlya-chego-nuz...
8.10%	https://studassistent.ru/pascal-abc/zapolnit-massiv-chislami-fib...
8.10%	https://ru.wikipedia.org/wiki/%D0%92%D1%8B%D1%87%D0%B8%D1%81%D0%...
7.70%	https://is59-2015.susu.ru/solovyev/2017/11/
7.70%	https://is59-2015.susu.ru/solovyev/author/nachtigall/
6.80%	https://www.cyberforum.ru/cpp-beginners/thread1091421.html

Информация о документе:

Частное учреждение профессионального образования "Высшая школа предпринимательства" (ЧУПО "ВШП") ОТЧЕТ ПО ПРАКТИКЕ по основной образовательной программе среднего профессионального образования по специальности 09.02.07 "Информационные системы и программирование" Вид практики (учебная, производственная, преддипломная): _____ Установленный по КУГ срок прохождения практики: с __.__.20__г. по __.__.20__г. Место прохождения практики (наименование организации): _____ _____ _____ Выполнил

студент __-го курса _____ (подпись) _____ (фамилия, имя, отчество) _____ Руководитель от образовательной организации _____ (подпись) _____ (ученая степень, фамилия, имя, отчество) _____ _____ (должность) Руководитель от предприятия _____ (подпись) _____ (ученая степень, фамилия, имя, отчество) _____ _____ (должность) Оценка _____ (прописью) Дата сдачи отчета: __.__.20__г. Тверь, 2024 г. Оглавление Введение 3 Вычислительная сложность 6 Алгоритмы для вычисления ряда Фибоначчи 8 Задача №1 8 Задача №2 10 Задача №3 11 Задача №4 13 Задача №5 14 Задачи по алгоритмам Хаффмана 17 Алгоритмы сортировки 24 Заключение 26 Электронные ресурсы 27 Приложение №1 28 Приложение №2 29 Приложение №3 31 Введение В современном мире, где технологическое развитие не имеет смысла, существует множество подходов к решению одной и той же технической задачи. Каждое из них имеет свои преимущества и недостатки, которые необходимо учитывать при выборе. Тем не менее, независимо от вариантов обилия, предпочтение обычно отдается только одному, принятому решению. Умение выбора такого решения является необходимым навыком, который позволяет не только создавать эффективное программное обеспечение, но и минимизировать временные и ресурсные затраты. Научиться оценивать производительность алгоритмов и находить наиболее рациональные подходы к решению задач - одна из ключевых целей для любого разработчика программного обеспечения. Это особенно актуально в условиях, когда требования к скорости и настройке программ становятся все более строгими. В рамках учебной практики, проходившей с 8 ноября по 27 декабря, я сосредоточился на изучении понятия "вычислительная сложность". Этот параметр играет решающую роль при анализе алгоритмов, так как позволяет количественно оценить, эффективно реализовать свою задачу при увеличении объема данных. Чтобы лучше понять теоретическую основу, я изучил и применил вычислительную сложность на примере таких классических алгоритмов, как: • алгоритм Хаффмана - используется для сжатия данных, • последовательность Фибоначчи - математический объект с широким применением в программировании, • алгоритмы сортировки, которые являются фундаментальными для разработки в любой области. Цель учебной практики: Развить понимание вычислительной сложности и научиться применять эти знания на практике, используя классические алгоритмы. Задачи учебной практики: Для достижения целей я поставил перед собой следующие задачи: 1. Разберемся с понятием "вычислительная сложность" и изучим его теоретическую основу. 2. Исследуются алгоритмы расчета чисел Фибоначчи, включая их особенности и производительность. 3. Освоить метод Хаффмана, его принципы работы и применимость области применения. 4. Рассмотрите алгоритмы сортировки, изучите их особенности и применимость в зависимости от задачи. Выполненные работы: В процессе практики мной были выполнены следующие задания: • Решение задач по вычислению чисел Фибоначчи с использованием различных подходов, таких как рекурсия, итерация и методы оптимизации. • Изучение и реализация алгоритма Хаффмана, включающего построение дерева и кодирования строк. Этим задачам удается закрепить теоретические знания о применении, улучшить навыки написания кода и улучшить понимание сложности алгоритмов. Для решения поставленных задач я использовал язык программирования Python, который выбрал по необходимым причинам: 1. Высокая скорость разработки. Python предоставляет множество встроенных библиотек и инструментов, упрощающих выполнение сложных программ. 2. Уверенное вмешательство. Данный язык является для меня основным вариантом разработки, в котором особое внимание уделяется алгоритмам, а не изучению синтаксиса. 3. Логический и читаемый синтаксис. Python обеспечивает простоту и ясность кода, что делает его удобным для анализа и улучшения программ.[3] Вычислительная сложность В рамках первого учебно-практического занятия мы ознакомились с понятием вычислительная сложность. Были рассмотрены такие аспекты, как история возникновения данного понятия, расчет значения и способы выражения. Вычислительная сложность - понятие в информатике и теории алгоритмов, обозначающее функцию зависимости объёма работы, которая выполняется некоторым алгоритмом, от размера входных данных. Нам объяснили как выражается вычислительная сложность на нескольких примерах: • "почистить ковёр пылесосом" требует время, линейно зависящее от его площади. Если площадь ковра увеличить в N раз, то и время, затраченное на его очистку тоже увеличиться в N раз. • "найти имя в телефонной книге" требует всего лишь времени, логарифмически зависящего от количества записей. Так как телефонная книга отсортирована по алфавиту, достаточно открыть её посередине для уменьшения круга поиска нужного имени в двое. Ещё раз поделив оставшиеся страницы пополам, мы сократим поиск имени в 4 раза, и так далее. [5] Рассмотрение входных данных большого размера и оценка порядка роста времени работы алгоритма приводят к понятию асимптотической сложности алгоритма. Наиболее популярной нотацией для описания вычислительной сложности алгоритмов является "O большое" Нотация O большое - это математическая нотация, которая описывает ограничивающее поведение функции, когда аргумент стремится к определенному значению или бесконечности. Впервые обозначение "O большое" было введено во втором томе книги математика Пауля Бахмана, вышедшем в 1894 году. [2] Существуют следующие наиболее часто встречающиеся классы сложности алгоритмов: • O(1) - константное время, например: определение чётности числа (представленного в двоичном виде); целого • O(log n) - логарифмическое время, например: двоичный поиск; • O(n) - линейное время, например: поиск наименьшего или наибольшего элемента в неотсортированном массиве; • O(n log n) - линейно-логарифмическое время, например: самый быстрый вариант сортировки сравнением, сортировка слиянием; • O(n^2) - квадратичное время, например: пузырьком, сортировка вставками; сортировка n×n матриц, вычисление частичной корреляции; • O(n^3) кубическое время, например: обычное умножение двух • O(n^k) - полиномиальное время, например: алгоритм Кармаркара для линейного программирования, AKC-тест простоты; • O(k^n) - экспоненциальное время, например: решение задачи порядке перемножения матриц с помощью полного перебора; • O(n!) -- факториальное время, например: решение коммивояжёра полным перебором. задачи Алгоритмы для вычисления ряда Фибоначчи В рамках данной учебной практики мы изучили числовой ряд Фибоначчи, а также его математическое и историческое значение. Последовательность чисел, впервые описанная итальянским математиком Леонардо Фибоначчи в 1202 году, представляет собой ряд чисел, в которых каждый следующий элемент Фибоначчи равен сумме двух предыдущих. Числа Фибоначчи находят широкое применение в различных областях, таких как программирование, теория чисел, искусство,

архитектура, биология, и их изучение является необходимым этапом для освоения базовых физических алгоритмов. На занятиях нам были продемонстрированы примеры классических алгоритмов для вычисления чисел Фибоначчи. Мы реализовали их с использованием языка программирования Python и проанализировали разницу в производительности.[3] Задача №1 Первая задача, связанная с вычислением чисел Фибоначчи, звучит следующим образом: Дано целое число 1≤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. Для решения этой задачи я написал такой алгоритм: ... def fib(n): """Рекурсивная функция для вычисления n-го числа Фибоначчи""" if n <= 0: return 0 elif n == 1: return 1 else: return fib(n - 1) + fib(n - 2) ... 1. Имя функции и описание: Функция называется fib, и её цель - вычислить n-е число Фибоначчи, где n - входной параметр функции. 2. Параметр функции: Функция принимает один параметр - n, который представляет собой индекс числа в последовательности Фибоначчи. 3. Условия: • Первое условие: if n <= 0:. Если значение n меньше или равно нулю, функция возвращает 0. Это базовый случай, так как F(0)=0F(0). • Второе условие: elif n == 1:. Если 1n=1, функция возвращает 1. Это второй базовый случай, так как F(1)=1F(1). 4. Рекурсивный случай: Если 1n>1, то функция возвращает результат сложения двух рекурсивных вызовов: fib(n - 1) + fib(n - 2). Это и есть основная идея алгоритма Фибоначчи, где каждое число в последовательности равно сумме двух предыдущих. 5. Работа функции: • Функция рекурсивно вызывает себя для n−1n и 2n−2 до тех пор, пока не достигает базовых случаев, где n≤1n. Тогда она начинает возвращать значения, которые постепенно складываются в итоговый результат. Задача №2 Вторая задача, связанная с вычислением чисел Фибоначчи, звучит следующим образом: Дано целое число 1≤n≤32, необходимо написать функцию fib(n) для вычисления n-го числа Фибоначчи с использованием цикла. Функция fib(n) должна производить расчет от 1 до , на каждой последующей итерации используя значение числа(чисел), необходимых для расчета, полученных на предыдущей итерации. В результате выполнения, функция должна вывести на экран вычисленное число Фибоначчи, например fib(3) должна вывести число 2, а fib(7) - соответственно 13. Для решения этой задачи я написал такой алгоритм: ... def fib(n): """Функция для вычисления n-го числа Фибоначчи с использованием цикла""" if n <= 0: return 0 elif n == 1: return 1 prev , curr = 0 , 1 for _ in range (2 , n + 1): prev , curr = curr , prev + curr return curr ... Функция fib(n) вычисляет n-е число Фибоначчи с использованием циклического подхода. Вместо рекурсии, здесь применяется итерация для вычисления значений последовательности Фибоначчи. Сначала функция проверяет условия для базовых случаев: если n меньше или равно 0, возвращается 0, а если n равно 1, возвращается 1. Если n больше 111, то функция использует цикл для вычисления следующих чисел. Начальные значения prev и curr устанавливаются на 000 и 111 соответственно. Затем цикл начинается с 222-го элемента и продолжается до n-го. В каждом шаге цикла переменные обновляются: prev становится старым значением curr, а curr становится суммой prev и curr. После завершения цикла функция возвращает значение curr, которое представляет n-е число Фибоначчи. Этот итеративный подход более эффективен, чем рекурсия, так как имеет линейную сложность O(n) и требует меньше памяти. Задача №3 Третья задача, связанная с вычислением чисел Фибоначчи, звучит следующим образом: Дано целое число 1≤n≤40 , необходимо написать функцию fib(n) для вычисления n-го числа Фибоначчи. Функция fib(n) должна в процессе выполнения записывать вычисленные значения в массив таким образом что индекс записанного числа в массиве должен соответствовать порядковому номеру числа Фибоначчи. При этом уже вычисленные значения должны браться из массива, а вновь вычисляемые должны записываться в массив только в случае если они еще не были вычислены. В результате выполнения, функция должна вывести на экран массив, содержащий все вычисленные числа Фибоначчи вплоть до заданного, включая его например fib(8) должна вывести массив: [0, 1, 1, 2, 3, 5, 8, 13, 21]. Для решения этой задачи я написал такой алгоритм: ... def fib(n): """Функция для вычисления чисел Фибоначчи с записью в массив""" if n < 0: return [] # Инициализация массива fib_sequence = [0] * (n + 1) # Задание начальных условий if n >= 1: fib_sequence[1] = 1 # Вычисление чисел Фибоначчи for i in range(2, n + 1): fib_sequence[i] = fib_sequence[i - 1] + fib_sequence[i - 2] return fib_sequence ... Функция fib(n) вычисляет последовательность чисел Фибоначчи от F(0) до F(n) и возвращает их в виде массива. Если n<0n < 0n<0, возвращается пустой массив, так как последовательность для отрицательных индексов не определена. Для вычислений создаётся массив длиной n+1n + 1n+1, все элементы которого изначально равны нулю. Начальные условия задаются: F(0)=0 (уже задано в массиве) и F(1)=1, если n≥1n. Далее используется цикл, который проходит от 2 до n, на каждом шаге вычисляя текущее число Фибоначчи как сумму двух предыдущих чисел, сохранённых в массиве. Результаты записываются в массив, а по завершении работы возвращается массив, содержащий всю последовательность. Например, вызов fib(5) вернёт массив [0, 1, 1, 2, 3, 5]. Такой подход позволяет быстро вычислить последовательность с линейной сложностью O(n) и сохранить все числа для дальнейшего использования.[6] Задача №4 Четвёртая задача, связанная с вычислением чисел Фибоначчи, звучит следующим образом: Дано целое число 1 ≤ ≤ 64 1≤n≤64 , необходимо написать функцию fib(n) для вычисления n-го 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 такой алгоритм: ... def fib(n): """Функция для вычисления n-го числа Фибоначчи с использованием формулы Бине""" if n < 0: raise ValueError ("n должно быть больше или равно 0") # Формула Бине sqrt_5 = math.sqrt (5) phi = (1 + sqrt_5) / 2 # Золотое сечение psi = (1 - sqrt_5) / 2 # Вычисление числа Фибоначчи с округлением fib_n = (phi ** n - psi ** n) / sqrt_5 return round (fib_n) ... Функция fib(n) вычисляет n-е число Фибоначчи с использованием формулы Бине. Если n < 0, функция вызывает исключение ValueError , так как последовательность Фибоначчи определена только для неотрицательных чисел. В основе расчётов лежит формула Бине, которая использует золотое сечение (ф) и его сопряжённое значение (ф). Сначала вычисляются ф = (1 + √5)/2, а затем по формуле (ф^n - ф^n)/√5 определяется n-е число Фибоначчи. Полученное значение округляется с помощью функции round() для устранения возможных ошибок, связанных с

вычислениями с плавающей точкой, и возвращается результат. Этот метод эффективен и позволяет вычислять числа Фибоначчи за O(1) времени, но может быть менее точным для очень больших n, из-за ограничений точности чисел с плавающей точкой. Задача №5 Пятая задача, связанная с вычислением чисел Фибоначчи, звучит следующим образом: Дано целое число 1≤n≤10 6 , необходимо написать функцию fib_eo(n) для определения четности n-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 ≤ , ≤ 9 0≤a,b≤9 - последние цифры чисел F n и + 1 F n+1 соответственно, то (+) mod 10 (a+b)mod10 - последняя цифра числа + 2 F n+2 . В результате выполнения, функция должна вывести на экран четное ли число или нет (even или odd соответственно), например fib_eo(841645) должна вывести odd, т.к. последняя цифра данного числа - 5. Для решения этой задачи я написал такой алгоритм: ... def fib_eo(n): """Функция для определения четности n-го числа Фибоначчи""" if n < 0: raise ValueError ("n должно быть больше или равно 0") # Последние цифры двух первых чисел Фибоначчи a , b = 0 , 1 # Циклическое вычисление последней цифры for _ in range (n): # итерация от 0 до n-1 a , b = b , (a + b) % 10 # Последняя цифра n-го числа last_digit = a # Определение четности return "even" if last_digit % 2 == 0 else "odd" ... Функция fib_eo(n) определяет, является ли n-е число Фибоначчи чётным или нечётным. Если n<0, вызывается ошибка. Для вычисления последней цифры числа Фибоначчи используется итеративный подход: переменные a и b инициализируются как первые два числа Фибоначчи (0 и 1), а затем обновляются по циклу с учётом остатка от деления на 10. После завершения цикла последняя цифра числа Фибоначчи сохраняется в переменной last_digit, на основе которой определяется его чётность. Функция возвращает строку "even" для чётных чисел и "odd" для нечётных. Этот алгоритм обладает такой же высокой скоростью работы, как и другие алгоритмы с использованием циклов, но при этом отличается экономным использованием памяти, так как в переменных сохраняются лишь значения младших разрядов обрабатываемых чисел.[8] Задачи по алгоритмам Хаффмана На этой лекции нам рассказали о Дэвиде Альберте Хоффмане, его достижениях в области науки и технологиях, а также подробно объяснили алгоритм, который он разработал. Задача №1 Первая задача, связанная с кодирование строки по алгоритму Хаффмана, звучит следующим образом: По данной строке, состоящей из строчных букв латинского алфавита: Errare humanum est. постройте оптимальный беспрефиксный код на основании классического алгоритма кодирования Хаффмана. В результате выполнения, функция huffman_encode() должна вывести на экран в первой строке - количество уникальных букв, встречающихся в строке и размер получившейся закодированной строки в битах. В следующих строках запишите коды символов в формате "'symbol': code". В последней строке выведите саму закодированную строку. Пример вывода для данного текста: 12 67 ' ': 000 '.': 1011 'E': 0110 'a': 1110 'e': 1111 'h': 0111 'm': 010 'n': 1000 'r': 110 's': 1001 't': 1010 'u': 001 01101101101110110111100001110010101110100000101000011111100110 101011 Для решения данной задачи мной был разработан следующий алгоритм (см. Приложение №2). Этот код реализует алгоритм Хаффмана для сжатия текста. Сначала класс Node описывает узлы дерева Хаффмана, где каждый узел хранит символ, его частоту, а также ссылки на левого и правого потомков. Функция build_huffman_tree строит дерево Хаффмана, используя частоты символов из текста и минимальную кучу для упорядочивания узлов. Функция build_codes обходит дерево, формируя префиксные коды для каждого символа, используя "0" для левого пути и "1" для правого. Основная функция huffman_encode принимает строку, строит дерево Хаффмана, генерирует коды символов, а затем кодирует строку с помощью этих кодов. На выходе отображаются количество символов, длина закодированной строки