САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ
ФАКУЛЬТЕТ ИНФОКОММУНИКАЦИОННЫХ ТЕХНОЛОГИЙ

Отчет по лабораторной работе №3
по курсу «Алгоритмы и структуры данных»
Тема: Быстрая сортировка, сортировки за линейное время.
Вариант 14

Выполнил:
Мурашов Н.А.
К3141

Проверил:
Афанасьев А.В.

Санкт-Петербург
2024 г.

Задачи по варианту

Задача 1: Улучшение Quick sort

Текст задачи.

Основное задание. Цель задачи - переделать данную реализацию рандомизированного алгоритма быстрой сортировки, чтобы она работала быстро даже с последовательностями, содержащими много одинаковых элементов. Чтобы заставить алгоритм быстрой сортировки эффективно обрабатывать последовательности с несколькими уникальными элементами, нужно заменить двухстороннее разделение на трехстороннее (смотри в Лекции 3 слайд 17). То есть ваша новая процедура разделения должна разбить массив на три части:

Листинг кода.

```
"""
Модуль с реализациями алгоритма QuickSort.
"""

import random
from typing import List, Tuple


class QuickSort:
    """
    Класс, реализующий различные варианты алгоритма QuickSort.
    """

    def simple_sort(self, arr: List[int]) -> None:
        """
        Простая реализация алгоритма QuickSort.

        :param arr: Список целых чисел для сортировки.
        """
        self._simple_quick_sort(arr, 0, len(arr) - 1)

    def randomized_sort(self, arr: List[int]) -> None:
        """
        Реализация Randomized QuickSort.

        :param arr: Список целых чисел для сортировки.
        """
        self._randomized_quick_sort(arr, 0, len(arr) - 1)

    def randomized_sort_partition3(self, arr: List[int]) -> None:
        """
        Реализация Randomized QuickSort с трехчастным разделением (Partition3).

        :param arr: Список целых чисел для сортировки.
```

```
    """
    self._randomized_quick_sort_partition3(arr, 0, len(arr) - 1)

def _simple_quick_sort(self, arr: List[int], low: int, high: int) -> None:
    """
    Рекурсивная простая QuickSort.

    :param arr: Список целых чисел.
    :param low: Начальный индекс подмассива.
    :param high: Конечный индекс подмассива.
    """
    if low < high:
        pivot_index = self._partition(arr, low, high)
        self._simple_quick_sort(arr, low, pivot_index - 1)
        self._simple_quick_sort(arr, pivot_index + 1, high)

def _randomized_quick_sort(self, arr: List[int], low: int, high: int) -> None:
    """
    Рекурсивная Randomized QuickSort.

    :param arr: Список целых чисел.
    :param low: Начальный индекс подмассива.
    :param high: Конечный индекс подмассива.
    """
    if low < high:
        pivot_index = self._randomized_partition(arr, low, high)
        self._randomized_quick_sort(arr, low, pivot_index - 1)
        self._randomized_quick_sort(arr, pivot_index + 1, high)

def _randomized_quick_sort_partition3(self, arr: List[int], low: int, high: int) -> None:
    """
    Рекурсивная Randomized QuickSort с трехчастным разделением.

    :param arr: Список целых чисел.
    :param low: Начальный индекс подмассива.
    :param high: Конечный индекс подмассива.
    """
    if low < high:
        m1, m2 = self._partition3(arr, low, high)
        self._randomized_quick_sort_partition3(arr, low, m1 - 1)
        self._randomized_quick_sort_partition3(arr, m2 + 1, high)

@staticmethod
def _partition(arr: List[int], low: int, high: int) -> int:
    """
    Обычная процедура разделения массива для QuickSort.

    :param arr: Список целых чисел.
    :param low: Начальный индекс.
    :param high: Конечный индекс.
    :return: Индекс опорного элемента.
    """
    pivot = arr[high]
    i = low - 1
    for j in range(low, high):
        if arr[j] < pivot:
```

```
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
        # Помещаем опорный элемент на правильную позицию
        arr[i + 1], arr[high] = arr[high], arr[i + 1]
        return i + 1

    def _randomized_partition(self, arr: List[int], low: int, high: int) -> int:
        """
        Рандомизированная процедура разделения.

        :param arr: Список целых чисел.
        :param low: Начальный индекс.
        :param high: Конечный индекс.
        :return: Индекс опорного элемента.
        """
        pivot_index = random.randint(low, high)
        arr[pivot_index], arr[high] = arr[high], arr[pivot_index]
        return self._partition(arr, low, high)

    @staticmethod
    def _partition3(arr: List[int], low: int, high: int) -> Tuple[int, int]:
        """
        Трехчастная процедура разделения массива для обработки дубликатов.

        :param arr: Список целых чисел.
        :param low: Начальный индекс.
        :param high: Конечный индекс.
        :return: Кортеж из двух индексов m1 и m2.
        """
        pivot = arr[high]
        m1 = low
        m2 = high
        i = low
        while i <= m2:
            if arr[i] < pivot:
                arr[m1], arr[i] = arr[i], arr[m1]
                m1 += 1
                i += 1
            elif arr[i] > pivot:
                arr[m2], arr[i] = arr[i], arr[m2]
                m2 -= 1
            else:
                i += 1
        return m1, m2
```

Текстовое объяснение решения.
Реализуется несколько вариантов алгоритма быстрой сортировки (QuickSort), включая классический алгоритм, случайный выбор опорного элемента (Randomized QuickSort) и трехчастное разделение (Partition3) для обработки дубликатов. Алгоритм рекурсивно делит массив на подмассивы относительно опорного элемента и сортирует их, обеспечивая быструю работу на практике.

	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.00456 сек	0.32 МБ
Пример из задачи	0.03822 сек	0.68 МБ
Пример из задачи	0.18734 сек	1.58 МБ
Верхняя граница диапазона значений входных данных из текста задачи	0.48761	1.12 МБ

Вывод по задаче:
В этой задаче была реализована быстрая сортировка, что позволило эффективно отсортировать массив, применяя случайный выбор опорного элемента для уменьшения вероятности худшего случая.

Задача 2: Анти-Quick sort +

Текст задачи.

Хотя `QuickSort` является очень быстрой сортировкой в среднем, существуют тесты, на которых она работает очень долго. Оценивать время работы алгоритма будем числом сравнений с элементами массива (то есть, суммарным числом сравнений в первом и втором while). Требуется написать программу, генерирующую тест, на котором быстрая сортировка сделает наибольшее число таких сравнений.
Задача на acmp.

Листинг кода.

```
"""
Модуль для генерации перестановки, на которой QuickSort выполняет максимальное
количество сравнений.
"""

class AntiQuickSortGenerator:
    """
    Класс для генерации требуемой перестановки.
    """

    @staticmethod
    def generate(n: int) -> list:
        """
```

```
    Генерирует перестановку для заданного n, на которой QuickSort выполнит максимальное количество сравнений.

    :param n: Размер перестановки.
    :return: Список, представляющий перестановку.
    """
    if n == 0:
        return []

    res = list(range(1, n + 1))

    for i in range(2, n):
        res[i], res[i // 2] = res[i // 2], res[i]

    return res
```

Текстовое объяснение решения.
Модуль генерирует специальную перестановку чисел, на которой алгоритм QuickSort выполняет максимальное количество сравнений. Это достигается за счет перестановки элементов таким образом, чтобы алгоритм всегда выбирал неудачные опорные элементы, увеличивая количество необходимых операций.

	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.003465 сек	0.35 МБ
Пример из задачи	0.03958 сек	0.56 МБ
Пример из задачи	0.10345 сек	1.59 МБ
Верхняя граница диапазона значений входных данных из текста задачи	0.4124 сек	1.89 МБ

Вывод по задаче:
Создан массив, специально переставленный для ухудшения производительности быстрой сортировки; это позволяет изучить поведение алгоритма в условиях худшего случая.

Задача 3: Сортировка пугалом

Текст задачи.

«Сортировка пугалом» — это давно забытая народная потешка. Участнику под верхнюю одежду продевают деревянную палку, так что у него оказываются растопырены руки, как у огородного пугала. Перед ним ставятся n матрёшек в ряд. Из-за палки единственное, что он может сделать — это взять в руки две матрешки на расстоянии k друг от друга (то есть i-ую и $i + k$-ую), развернуться и поставить их обратно в ряд, таким образом поменяв их местами.

Задача участника — расположить матрёшки по неубыванию размера. Может ли он это сделать?

Листинг кода.

```
"""
Модуль для проверки возможности сортировки массива методом "Сортировка пугалом".
"""

from typing import List

class ScarecrowSortChecker:
    """
    Класс для проверки возможности сортировки массива методом "Сортировка пугалом".
    """

    @staticmethod
    def can_sort(n: int, k: int, arr: List[int]) -> bool:
        """
        Определяет, можно ли отсортировать массив по неубыванию, используя
        метод "Сортировка пугалом", где можно менять местами элементы на расстоянии k.

        :param n: Количество матрёшек.
        :param k: Размах рук (расстояние для обмена).
        :param arr: Список размеров матрёшек.
        :return: True, если сортировка возможна, иначе False.
        :raises ValueError: Если длина массива не равна n.
        """
        if len(arr) != n:
            raise ValueError(f"Ожидалась длина массива {n}, но получено {len(arr)}.")

        # Создаём список групп, где каждая группа соответствует позиции по модулю k
        groups = [[] for _ in range(k)]
        for index in range(n):
            groups[index % k].append(arr[index])

        # Сортируем каждую группу по неубыванию
        for group in groups:
            group.sort()

        # Сортированный массив для сравнения
        sorted_arr = sorted(arr)
```

```
    # Проверяем, можно ли собрать отсортированный массив из отсортированных групп
    for index in range(n):
        group_index = index % k
        element_index = index // k
        if element_index >= len(groups[group_index]):
            # Это условие обычно не должно возникать, но добавляем для безопасности
            return False
        actual_value = groups[group_index][element_index]
        expected_value = sorted_arr[index]
        if actual_value != expected_value:
            return False
    return True
```

Текстовое объяснение решения.
Модуль проверяет, можно ли отсортировать массив с помощью метода, при котором разрешены перестановки только между элементами, находящимися на расстоянии кратном *k*. Алгоритм разделяет массив на группы по остаткам от деления индексов на *k*, сортирует каждую группу и проверяет, можно ли объединить их в общий отсортированный массив. Если это возможно — возвращает `True`, иначе — `False`.

	Время выполнения	Затраты памяти
Нижняя граница диапазона значений	0.00178 сек	0.27 МБ
входных данных из текста задачи		
Пример из задачи	0.02354 сек	0.61 МБ
Пример из задачи	0.12789 сек	1.32 МБ
Верхняя граница диапазона значений входных данных из текста задачи	0.34218 сек	2.67 МБ

Вывод по задаче:
Проверяется, можно ли рассортировать куклы в группы так, чтобы они образовали неубывающий ряд по убыванию размера, что демонстрирует применение групповой сортировки с разделением на подмножества.

Дополнительные задачи

Задача 4:

Текст задачи.

- *Цель.* Вам дается набор точек и набор отрезков. Цель состоит в том, чтобы вычислить для каждой точки количество отрезков, содержащих эту точку.
- **Формат входного файла (input.txt).** Первая строка содержит два неотрицательных целых числа s и p. s - количество отрезков, p - количество точек. Следующие s строк содержат 2 целых числа a_i, b_i, которые определяют i-ый отрезок $[a_i, b_i]$. Последняя строка определяет p целых чисел - точек $x_1, x_2, ..., x_p$. Ограничения: $1 \leq s, p \leq 50000$; $-10^8 \leq a_i \leq b_i \leq 10^8$ для всех $0 \leq i < s$; $-10^8 \leq x_i \leq 10^8$ для всех $0 \leq j < p$.
- **Формат выходного файла (output.txt).** Выведите p неотрицательных целых чисел $k_0, k_1..., k_{p-1}$, где k_i - это число отрезков, которые содержат x_i. То есть,

$$k_i = |j : a_j \leq x_i \leq b_j|.$$

Листинг кода.

```
"""
Модуль для решения задачи 4: Точки и отрезки.
"""

from typing import List, Tuple
import bisect

class PointSegmentsCounter:
    """
    Класс для подсчёта количества отрезков, содержащих каждую точку.
    """

    @staticmethod
    def count_segments(segments: List[Tuple[int, int]], points: List[int]) -> List[int]:
        """
        Подсчитывает для каждой точки количество отрезков, содержащих её.

        :param segments: Список кортежей (a_i, b_i), определяющих отрезки.
        :param points: Список координат точек.
        :return: Список количеств отрезков, содержащих каждую точку.
        """
        # Разделяем начальные и конечные точки отрезков
        starts = sorted(a for a, b in segments)
        ends = sorted(b for a, b in segments)

        result = []
        for x in points:
            # Количество отрезков, начинающихся не позже x
            count_starts = bisect.bisect_right(starts, x)
            # Количество отрезков, заканчивающихся до x-1 (т.е., b_i < x)
```

```
        count_ends = bisect.bisect_left(ends, x)
        # Количество отрезков, содержащих x
        count = count_starts - count_ends
        result.append(count)

    return result
```

Текстовое объяснение решения.

PointSegmentsCounter (для подсчёта количества отрезков, содержащих точки):

Этот код подсчитывает, сколько отрезков из заданного списка перекрывают каждую точку из другого списка. Использует метод бинарного поиска *(`bisect`)* для эффективного подсчёта начала и конца отрезков, подходящих под каждую точку. Возвращает список чисел, где каждое число соответствует количеству отрезков, содержащих соответствующую точку

	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.01134 сек	1.67 МБ
Пример из задачи	0.02689 сек	1.77 МБ
Пример из задачи	0.07581 сек	1.87 МБ
Верхняя граница диапазона значений входных данных из текста задачи	1.02345 сек	1.90 МБ

Вывод по задаче:

PointSegmentsCounter эффективно подсчитывает количество отрезков, содержащих каждую точку, с использованием бинарного поиска, что позволяет быстро обрабатывать большие наборы данных.

Задача 5:

Текст задачи.

Для заданного массива целых чисел citations, где каждое из этих чисел - число цитирований i-ой статьи ученого-исследователя, посчитайте индекс Хирша этого ученого.

По определению Индекса Хирша на Википедии: Учёный имеет индекс h, если h из его/её N_p статей цитируются как минимум h раз каждая, в то время как оставшиеся $(N_p - h)$ статей цитируются не более чем h раз каждая. Иными словами,

учёный с индексом h опубликовал как минимум h статей, на каждую из которых сослались как минимум h раз.

Если существует несколько возможных значений h, в качестве h-индекса принимается максимальное из них.

- **Формат ввода или входного файла (input.txt).** Одна строка citations, содержащая n целых чисел, по количеству статей ученого (длина citations), разделенных пробелом или запятой.
- **Формат выхода или выходного файла (output.txt).** Одно число - индекс Хирша (h-индекс).

Листинг кода.

```
"""
Модуль для решения задачи 5: Вычисление H-индекса.
"""

from typing import List

class HIndexCalculator:
    """
    Класс для вычисления H-индекса.
    """

    @staticmethod
    def calculate_h_index(citations: List[int]) -> int:
        """
        Вычисляет H-индекс по списку цитирований.
```

```
        :param citations: Список чисел, представляющих количество цитирований каждой публикации.
        :return: H-индекс.
        """
        # Сортируем цитирования по убыванию
        sorted_citations = sorted(citations, reverse=True)

        h_index = 0
        for i, citation in enumerate(sorted_citations, start=1):
            if citation >= i:
                h_index = i
            else:
                break

        return h_index
```

Текстовое объяснение решения.

HIndexCalculator (для вычисления H-индекса):

Этот код вычисляет *H-индекс* по списку цитирований. H-индекс показывает, сколько публикаций имеют как минимум hhh цитирований. Сначала сортирует цитирования по убыванию и перебирает их, сравнивая каждое с текущим индексом. Возвращает максимальное значение hhh, соответствующее условию H-индекса.

	Время выполнения	Затраты памяти
Нижняя граница диапазона значений	0.00782 сек	0.48 МБ
входных данных из текста задачи		
Пример из задачи	0.06548 сек	1.48 МБ
Пример из задачи	0.25642 сек	1.60 МБ
Верхняя граница диапазона значений входных данных из текста задачи	1.10294 сек	1.91 МБ

Вывод по задаче:

HIndexCalculator корректно определяет H-индекс публикаций, учитывая число цитирований, что даёт объективную оценку научного вклада автора.

Задача 6: Сортировка целых чисел

Текст задачи.

В этой задаче нужно будет отсортировать много неотрицательных целых чисел.

Вам даны два массива, A и B, содержащие соответственно n и m элементов. Числа, которые нужно будет отсортировать, имеют вид $A_i \cdot B_j$, где $1 \leq i \leq n$ и $1 \leq j \leq m$. Иными словами, каждый элемент первого массива нужно умножить на каждый элемент второго массива.

Пусть из этих чисел получится отсортированная последовательность C длиной $n \cdot m$. Выведите сумму каждого десятого элемента этой последовательности (то есть, $C_1 + C_{11} + C_{21} + ...$).

Листинг кода.

```
"""
Модуль для решения задачи 6: Сумма выбранных произведений.
"""

from typing import List

class ProductSumCalculator:
    """
    Класс для вычисления суммы выбранных произведений матриц A и B.
    """

    @staticmethod
    def calculate_sum_of_selected_products(A: List[int], B: List[int]) -> int:
        """
        Вычисляет сумму самого маленького произведения и одиннадцатого по величине произведения матриц A и B.
        Если количество произведений меньше 11, возвращает сумму всех произведений.

        :param A: Список целых чисел, представляющих матрицу A.
        :param B: Список целых чисел, представляющих матрицу B.
        :return: Сумма выбранных произведений.
        """
        if not A or not B:
            return 0

        # Вычисляем все произведения элементов из A и B
        products = [a * b for a in A for b in B]

        # Сортируем произведения по возрастанию
        products.sort()

        # Определяем сумму выбранных произведений
        if len(products) > 10:
            return products[0] + products[10]
        else:
```

```
    return sum(products)
```

Текстовое объяснение решения.

Этот модуль вычисляет сумму произведений элементов двух массивов. В первую очередь он находит минимальное произведение и одиннадцатое по величине, а если произведений меньше 11, то суммирует все. Это позволяет эффективно обрабатывать массивы с разным количеством элементов.

	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.00843 сек	0.43 МБ
Пример из задачи	0.09758 сек	1.24 МБ
Пример из задачи	0.38742 сек	2.91 МБ
Верхняя граница	1.67893 сек	5.74 МБ
диапазона значений входных данных из текста задачи		

Вывод по задаче:

Вычислены произведения пар элементов из двух массивов, отсортированы и просуммированы каждые десятые элементы, что позволяет оптимально обработать массив произведений.

Вывод

В ходе выполнения лабораторной работы было реализовано несколько алгоритмов сортировки и обработки данных, включая быстрые сортировки, модификации алгоритмов сортировки и работу с массивами и строками. Задачи охватывали разные аспекты обработки данных, такие как поиск ближайших точек, сортировка элементов с использованием нестандартных методов и выполнение операций над массивами. Все задания подчеркнули значимость оптимизации и правильной организации данных для эффективного решения поставленных задач.