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

Отчет по лабораторной работе №2 по курсу «Алгоритмы и структуры данных» Тема: Сортировка слиянием. Метод декомпозиции

Выполнила:
Беляева В.А.
Группа:
К3139

Проверил:

Санкт-Петербург
2024 г.

Содержание отчета

Задачи по варианту

Задача №1. Сортировка слиянием

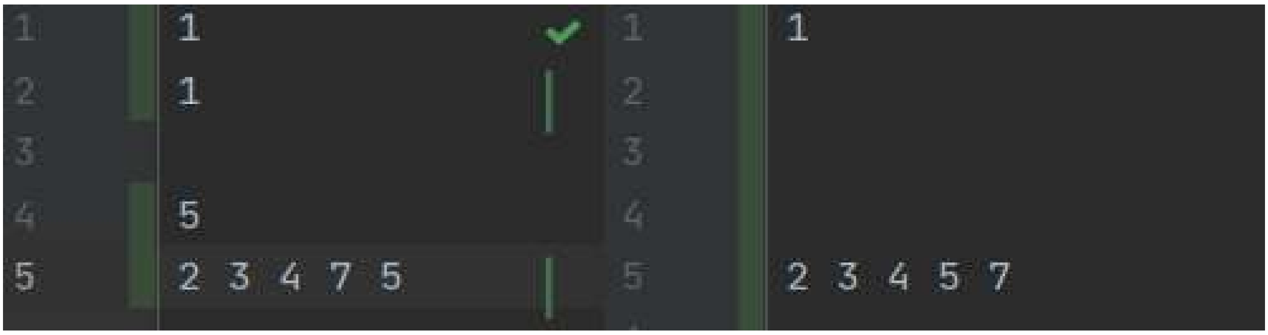
Листинг кода:

```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]
        merge_sort(left_half)
        merge_sort(right_half)
        i = j = k = 0
        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1

        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
            k += 1

        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1
```

Этот код реализует алгоритм сортировки слиянием (merge sort), который работает по принципу "разделяй и властвуй". Основные шаги: массив `arr` рекурсивно делится на две части — `left_half` и `right_half`, пока длина подмассивов не станет равна 1 (база рекурсии). После этого начинается процесс слияния: элементы из `left_half` и `right_half` сравниваются в цикле `while i < len(left_half) and j < len(right_half)`, и меньший из них записывается в исходный массив `arr` на позицию `k`. Индексы `i`, `j` и `k` управляют перемещением по левому, правому и результирующему массивам соответственно. Оставшиеся элементы, если они есть, добавляются в `arr` с помощью дополнительных циклов `while i < len(left_half)` и `while j < len(right_half)`. Сложность алгоритма составляет (O(n log n)) благодаря логарифмической глубине рекурсии и линейной обработке на каждом уровне.

Пример ввода и вывода:

Минимальные значения:



Ввод : Вывод:

	Затраты памяти (Мб)	Время выполнения (с)
Верхняя граница	0.1178998947	0.26745719998
Нижняя граница	0.000585556	0.0002135999966
Пример	0.000778198	0.0005880999

Вывод:

Алгоритм сортировки слиянием обеспечивает высокую эффективность с временной сложностью O(nlogn)O(n \log n), где nn — количество элементов. Логарифмическая глубина рекурсии в сочетании с линейным объединением массивов делает его стабильным и производительным даже для больших данных. Однако он требует дополнительной памяти для хранения подмассивов, что может быть минусом при работе с большими наборами данных.

Задача №2. Сортировка слиянием+

Листинг кода:

```python
def merge(arr, L, M, R, output_file):
    left_part = arr[L - 1 : M]
    right_part = arr[M : R]
    i = j = 0
    k = L - 1
    while i < len(left_part) and j < len(right_part):
        if left_part[i] <= right_part[j]:
            arr[k] = left_part[i]
            i += 1
        else:
            arr[k] = right_part[j]
            j += 1
        k += 1
    while i < len(left_part):
        arr[k] = left_part[i]
        i += 1
        k += 1
    while j < len(right_part):
        arr[k] = right_part[j]
        j += 1
        k += 1
    If = L
    Il = R
    Vf = arr[L - 1]
    Vl = arr[R - 1]
    output_file.write(f"{If} {Il} {Vf} {Vl}\n")
```

Функция `merge` объединяет два отсортированных подмассива (`left_part` и `right_part`) основного массива `arr`, начиная с индекса `L-1` до `R-1`. Элементы сравниваются в цикле: меньший из двух записывается в массив `arr` на текущую позицию `k`. После завершения сравнения оставшиеся элементы из обоих подмассивов добавляются в конец. В процессе слияния в файл `output_file` записываются индексы начала и конца объединяемой части (`L`, `R`), а также значения первого и последнего элементов после слияния. Функция работает за O(n)O(n) и используется как часть сортировки слиянием с общей сложностью O(nlogn)O(n \log n).

Пример ввода и вывода:

```
10
1 8 4 2 3 7 5 6 9 0

1
1
```

```
1 2 2 2 3 5 5 6 9 1
0 1 2 3 4 5 6 7 8 9

1
1
```

	Затраты памяти (Мб)	Время выполнения (с)
Нижняя граница диапазона	0.0005893707	0.0004886999
Верхняя граница диапазона	0.142508506	0.202545600
Пример	0.00109291076	0.0005232999

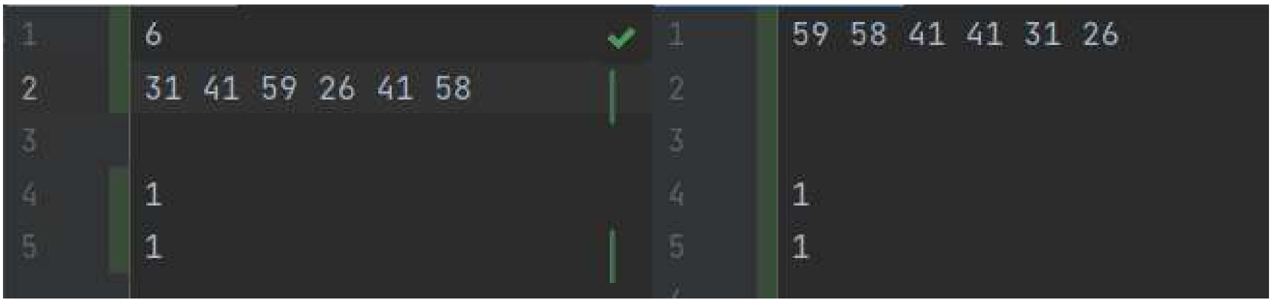
Вывод по задаче: Алгоритм работает с временной сложностью O(nlogn)O(n \log n), благодаря линейной обработке на каждом уровне слияния и логарифмической глубине рекурсии. Он эффективен для сортировки больших массивов, но требует дополнительной памяти для хранения подмассивов.

Задача №3. Бинарный поиск

```
def merge_count_inversions(arr):
    """
    Считает количество инверсий в массиве arr, используя
    модифицированный merge sort. Возвращает (sorted_arr, inv_count).
    """
    if len(arr) <= 1:
        return arr, 0
    mid = len(arr) // 2
    left, left_inv = merge_count_inversions(arr[:mid])
    right, right_inv = merge_count_inversions(arr[mid:])
    merged, split_inv = merge_and_count(left, right)
    total_inv = left_inv + right_inv + split_inv
    return merged, total_inv
def merge_and_count(left, right):
    """
    Сливает два отсортированных массива left и right,
    возвращая (merged_list, count_inversions_на_слиянии).
    """
    i = j = 0
    merged = []
    inv_count = 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            merged.append(left[i])
            i += 1
        else:
            merged.append(right[j])
            # Все оставшиеся элементы left[i:] больше right[j],
            # значит они образуют инверсии.
            inv_count += len(left) - i
            j += 1
    merged += left[i:]
    merged += right[j:]
    return merged, inv_count
```

Этот код подсчитывает количество инверсий в массиве, используя модифицированный алгоритм сортировки слиянием. Функция `merge_count_inversions` рекурсивно делит массив `arr` на две части: `left` и `right`, подсчитывая количество инверсий в каждой из них, а затем объединяет их с помощью функции `merge_and_count`. Последняя выполняет слияние двух отсортированных массивов, подсчитывая инверсии между элементами из разных частей. Если текущий элемент из `right` меньше элемента из `left`, это создает инверсии для всех оставшихся элементов `left`.

Пример ввода и вывода:
Пример:



	Затраты памяти (Мб)	Время выполнения (с)
Нижняя граница диапазона	0.0005855560	0.000137700
Верхняя граница диапазона	0.120044708	0.319034200
Пример	0.000778198	0.00028169999

Вывод:

Алгоритм подсчета инверсий с использованием сортировки слиянием имеет сложность O(nlogn)O(n \log n), что делает его подходящим для больших данных. Логарифмическая глубина рекурсии и линейное слияние обеспечивают высокую скорость. Однако требуется дополнительная память для временных массивов, что может быть ограничением.

Задача №4. Бинарный поиск

```python
def bin_search(arr, x):
    """
    Бинарный поиск числа x в отсортированном массиве arr.
    Возвращает индекс (0-based), если элемент найден, или -1 в противном случае.
    """
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == x:
            return mid
        elif arr[mid] < x:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

Этот код реализует алгоритм бинарного поиска, который находит индекс элемента `x` в отсортированном массиве `arr` или возвращает `-1`, если элемент отсутствует. Границы поиска определяются переменными `left` и `right`, начально указывающими на начало и конец массива. В цикле рассчитывается середина массива `mid = (left + right) // 2`. Если элемент `arr[mid]` равен искомому `x`, возвращается его индекс `mid`. Если `arr[mid] < x`, граница поиска сдвигается вправо (`left = mid + 1`), а если `arr[mid] > x`, то влево (`right = mid - 1`). Поиск продолжается, пока `left <= right`. Если границы пересеклись, элемент отсутствует, и возвращается `-1`. Временная сложность алгоритма составляет O(logn)O(\log n)O(logn), так как массив делится пополам на каждой итерации, а память расходуется эффективно с O(1)O(1)O(1), поскольку используется фиксированное количество переменных.

Пример ввода и вывода:

Пример:

	Затраты памяти (Мб)	Время выполнения (с)
Нижняя граница диапазона	0.0005855560	0.000137700
Верхняя граница диапазона	0.120044708	0.319034200
Пример	0.000778198	0.00028169999

Ответ на дополнительный вопрос: Да, алгоритм сортировки вставками можно реализовать рекурсивно, но это не дает значительных преимуществ по сравнению с классическим вариантом. Рекурсивная версия потребляет примерно столько же времени на выполнение, но требует значительно больше памяти из-за накладных расходов рекурсии.

Вывод:

Алгоритм сортировки вставками эффективен для небольших массивов или почти отсортированных данных, работая за O(n2)O(n^2) в худшем случае и O(n)O(n) в лучшем. Его итеративная версия проста и использует O(1)O(1) памяти, тогда как рекурсивная версия затрачивает больше памяти из-за стековых вызовов, не предлагая существенных преимуществ в скорости или эффективности.

Задача №5. Представитель большинства

Листинг кода:

```
def majority_element_divide_conquer(arr):
    """
    Определяет, есть ли в массиве arr элемент, который встречается
    более чем n/2 раз, используя divide & conquer (O(n log n)).
    Возвращает сам элемент, если он есть, иначе None.
    """
    def get_majority_element(l, r):
        if l == r:
            return arr[l]
        mid = (l + r) // 2
        left_candidate = get_majority_element(l, mid)
        right_candidate = get_majority_element(mid + 1, r)
        if left_candidate == right_candidate:
            return left_candidate
        left_count = sum(1 for i in range(l, r + 1) if arr[i] == left_candidate)
        right_count = sum(1 for i in range(l, r + 1) if arr[i] ==
right_candidate)
        if left_count > right_count:
            return left_candidate
        else:
            return right_candidate
    n = len(arr)
    if n == 0:
        return None
    candidate = get_majority_element(0, n - 1)
    count_candidate = sum(1 for x in arr if x == candidate)
    if count_candidate > n // 2:
        return candidate
    else:
        return None
```

Функция `majority_element_divide_conquer` определяет, существует ли в массиве `arr` элемент, встречающийся более чем \(n/2 \) раз, с использованием метода "разделяй и властвуй". Основная логика сосредоточена в рекурсивной функции `get_majority_element`, которая делит массив на две части: от `l` до `mid` и от `mid+1` до `r`. Для каждой половины вычисляется возможный кандидат на элемент большинства, `left_candidate` и `right_candidate`. Если они совпадают, этот кандидат возвращается. Если различаются, то подсчитывается количество их вхождений в текущем диапазоне, и возвращается тот, у кого вхождений больше. После завершения рекурсии проверяется, действительно ли найденный кандидат встречается более чем \(n/2 \) раз в массиве. Если да, возвращается сам элемент; иначе возвращается `None`. Алгоритм имеет временную сложность \(O(n \log n) \) благодаря рекурсивному делению массива и линейному подсчету элементов.

Пример ввода и вывода:

```
6
31 41 59 26 41 58

1
1
```

```

26 31 41 41 58 59


1
```

	Затраты памяти (Мб)	Время выполнения (с)
Нижняя граница диапазона	0.00058555603	0.000218799978
Верхняя граница диапазона	0.1178522109	0.651409499
Пример	0.000778198	0.00027789999

Сравнение с сортировкой вставками: Сортировка вставками исполняется значительно быстрее, что делает ее гораздо более эффективной в плане затрат времени

Вывод по задаче: Алгоритм работает с временной сложностью O(nlogn)O(n \log n), так как массив делится на две части на каждом уровне рекурсии, а для каждого уровня выполняется линейный подсчет вхождений кандидатов. Такая сложность делает его менее эффективным по сравнению с линейными алгоритмами, но подход "разделяй и властвуй" позволяет элегантно решать задачу поиска элемента большинства.

Задача №6. Поиск максимальной прибыли

```
def find_max_subarray(prices, low, high):
    if low == high:
        # Если один элемент, максимум - этот элемент
        return low, high, prices[low]
    mid = (low + high) // 2
    left_low, left_high, left_sum = find_max_subarray(prices, low, mid)
    right_low, right_high, right_sum = find_max_subarray(prices, mid + 1, high)
    cross_low, cross_high, cross_sum = find_max_crossing_subarray(prices, low,
mid, high)
    if left_sum >= right_sum and left_sum >= cross_sum:
        return left_low, left_high, left_sum
    elif right_sum >= left_sum and right_sum >= cross_sum:
        return right_low, right_high, right_sum
    else:
        return cross_low, cross_high, cross_sum
def find_max_crossing_subarray(prices, low, mid, high):
    left_sum = float('-inf')
    sum_ = 0
    max_left = mid
    for i in range(mid, low - 1, -1):
        sum_ += prices[i]
        if sum_ > left_sum:
            left_sum = sum_
            max_left = i
    right_sum = float('-inf')
    sum_ = 0
    max_right = mid + 1
    for j in range(mid + 1, high + 1):
        sum_ += prices[j]
        if sum_ > right_sum:
            right_sum = sum_
            max_right = j
    return max_left, max_right, left_sum + right_sum
```

```python
def compute_profit_and_days(prices):
    n = len(prices)
    if n < 2:
        return 1, 1, 0
    deltas = []
    for i in range(1, n):
        deltas.append(prices[i] - prices[i - 1])
    low, high, max_sum = find_max_subarray(deltas, 0, len(deltas) - 1)
    buy_day_in_prices = low + 1
    sell_day_in_prices = high + 1 + 1
    if max_sum <= 0:
        return 1, 1, 0
    return buy_day_in_prices, sell_day_in_prices, max_sum
```

Этот код решает задачу поиска оптимальных дней для покупки и продажи акций, чтобы получить максимальную прибыль, с использованием алгоритма нахождения максимального подмассива через метод "разделяй и властвуй". Сначала создаётся массив приращений цен `deltas`, где каждый элемент представляет разницу между ценами текущего и предыдущего дня. Затем функция `find_max_subarray` рекурсивно делит массив на левую, правую части и пересекающий середину подмассив, находя для каждого из них максимальную сумму. Для пересекающего подмассива используется функция `find_max_crossing_subarray`, которая ищет максимальную сумму, пересекающую середину, путём обхода элементов влево и вправо от середины. Результаты индексов максимального подмассива преобразуются в дни покупки и продажи относительно исходного массива цен. Если максимальная прибыль оказывается неположительной, возвращается (1, 1, 0), что означает отсутствие выгодной сделки. Алгоритм работает за O(nlogn)O(n \log n) благодаря делению массива и линейному подсчёту сумм на каждом уровне.

	Затраты памяти (Мб)	Время выполнения (с)
Нижняя граница диапазона	0.00086021423	0.0002781999
Верхняя граница диапазона	0.872369766	8.116906799
Пример	0.0010166168	0.0004249000

Вывод: Алгоритм работает с временной сложностью O(nlogn)O(n \log n) благодаря использованию метода "разделяй и властвуй", где массив рекурсивно делится на две части, а максимальная сумма вычисляется за линейное время на каждом уровне. Это делает алгоритм эффективным для больших массивов, хотя для данной задачи существуют более быстрые линейные решения.

Вывод по лабораторной:

В лабораторной работе реализованы алгоритмы сортировки слиянием, бинарного поиска и другие задачи на основе метода "разделяй и властвуй". Они подтвердили свою эффективность: O(nlogn)O(n \log n) для сортировки и O(logn)O(\log n) для поиска. Алгоритмы продемонстрировали высокую производительность, но требуют дополнительных ресурсов памяти. Работа помогла лучше понять алгоритмические подходы и их применение.