Санкт-Петербургский Национальный Исследовательский
УниверситетИнформационных Технологий, Механики и Оптики

Факультет инфокоммуникационных технологий

Лабораторная работа №3

Выполнили:
Стафеев И.А., Лапшина Ю.С., Килебе Нтангу Дьевина
Проверил
Мусаев А.А.

Санкт-Петербург,
2023

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цель работы: изучить несколько существующих алгоритмов сортировки, оценить их сложность, выявить преимущества и недостатки.

Для достижения цели были поставлены следующие задачи:

– изучить быструю сортировку и сортировку расческой, оценить время их выполнения с помощью timeit;

– изучить пирамидальную сортировку, блочную сортировку и сортировку слиянием;

– написать код, реализующий названные алгоритмы сортировки;

– оценить сложность, преимущества и недостатки каждой из изученных сортировок.

1 Быстрая сортировка и сортировка расческой

1.1 Быстрая сортировка

Быстрая сортировка основана на принципе «разделяй и властвуй». Идея данной сортировки заключается в разбиении массива на три группы относительно элемента pivot. Изначально элемент $pivot$ в данной программе взят как средний элемент по индексу в исходном массиве (в других вариациях реализации данного метода сортировки элемент $pivot$ может быть средним, медианой первого, среднего и последнего элементов или же - выбираться рандомно). Первая группа представлена массивом $prev\_nums$, содержащим в себе все элементы меньшие, чем $pivot$. Элементы равные $pivot$ отправляются во вторую группу, которая представлена в программе в качестве массива eq_nums. Третья группа представлена массивом $next\_nums$, содержащим все элементы, превосходящие $pivot$.

К первой и третьей группе элементов применяется всё то же разбиение на три группы, но теперь с новым элементом $pivot$. Это реализовано с помощью рекурсии, т.е к каждой группе применяется один и тот же алгоритм разбиения до тех пор, пока в группе не останется один элемент. В конце концов все группы соединяются в один отсортированный массив. Реализованная программа представлена на рисунке 1:

```
4 usages   dymonyx
def quicksort(array):  # алгоритм быстрой сортировки
    if len(array) < 2:
        return array
    else:
        pivot = array[(0 + len(array) - 1) // 2]
        prev_nums, eq_nums, next_nums = [], [], []
        for numb in array:
            if numb < pivot:
                prev_nums.append(numb)
            elif numb == pivot:
                eq_nums.append(numb)
            else:
                next_nums.append(numb)
        return quicksort(prev_nums) + eq_nums + quicksort(next_nums)
```

Рисунок 1 - Алгоритм быстрой сортировки

Сложность данной сортировки в среднем случае равна $O(n * logn)$, а в худшем $O(n^2)$. Если первая и третья группа на каждом вызове функции сортировки будут примерно равны – будет наилучший случай со сложностью $O(n * logn)$.

Пример работы кода представлен на рисунке 2:

```
Исходный массив [42, 91, 55, 35, 10]
[42] [] []
[42] [] [91]
[42] [55] [91]
[42, 35] [55] [91]
[42, 35, 10] [55] [91]
[] [] [42]
[] [35] [42]
[10] [35] [42]
Отсортированный массив [10, 35, 42, 55, 91]
```

Рисунок 2 - Пример выполнения быстрой сортировки

1.2 Сортировка расчёской

Сортировка расчёской является улучшенной версией сортировки пузырьком. Основная идея сортировки расчёской заключается в сравнении элементов с определённым шагом ($step$). (В начале сортировки он максимален,

а затем постепенно уменьшается). Изначальный шаг можно брать любой, но для лучшей эффективности принято брать его равным длине массива, поделённой на фактор ($factor$), равный 1.2743. На каждой итерации алгоритма шаг уменьшается на фактор (минимальное значение шага равно единице).

Данный метод сортировки реализован через вложенный цикл, в ходе которого сравниваются все пары элементов, расположенные на расстоянии шага друг от друга. (если первый элемент больше второго, то они меняются местами). Также была заведена переменная $swaps$, в которой хранится количество перестановок за время отработки одной итерации с определённым шагом. Она нужна для того, чтобы в случае, когда шаг уже стал равным одному, но перестановки за итерацию всё ещё есть, провести дополнительную итерацию и отсортировать массив до конца. Код реализованного алгоритма представлен на рисунке:

```
def combsort(array): # алгоритм сортировки расчёской
    factor = 1.2743
    step = len(array)
    swaps = 1
    while step > 1 or swaps > 0:
        print(array, "шаг: ", step)
        index = 0
        swaps = 0
        if step > 1:
            step = int(step // factor)
        while index + step < len(array):
            if array[index] > array[index + step]:
                array[index], array[index + step] = array[index + step], array[index]
                swaps += 1
            index += 1
    return array
```

Рисунок 3 - Алгоритм сортировки расческой

Сложность данного алгоритма в худшем случае равна $O(n^2)$.

Пример работы алгоритма представлен на рисунке 4:

```
Исходный массив [4, 37, 10, 76, 100, 97, 39, 12, 68, 45]
[4, 37, 10, 76, 100, 97, 39, 12, 68, 45] шаг:  10
[4, 37, 10, 76, 100, 97, 39, 12, 68, 45] шаг:  7
[4, 37, 10, 68, 45, 97, 39, 12, 76, 100] шаг:  5
[4, 37, 10, 39, 12, 76, 68, 45, 97, 100] шаг:  3
[4, 37, 10, 39, 12, 45, 68, 76, 97, 100] шаг:  2
[4, 10, 37, 12, 39, 45, 68, 76, 97, 100] шаг:  1
[4, 10, 12, 37, 39, 45, 68, 76, 97, 100] шаг:  1
Отсортированный массив [4, 10, 12, 37, 39, 45, 68, 76, 97, 100]
```

Рисунок 4 - Пример выполнения сортировки расческой

1.3 Оценка сложности помощью модуля timeit

Оценка была произведена для рандомно сгенерированного массива из 100, 1000, 10000 и 100000 чисел. Код реализации подсчёта времени представлен на рисунке 5:

```
quicksort_time = []
combsort_time = []
for x in [100, 1_000, 10_000, 100_000]:
    array = [random.randint( a: 0,  b: 100) for _ in range(0, x)]
    quicksort_time.append(round((timeit.timeit(lambda: quicksort(array), number=1)), 5))
    combsort_time.append(round((timeit.timeit(lambda: combsort(array), number=1)), 5))
```

Рисунок 5 - Генерация тестовых данных для сортировок

Результат измерений представлен на рисунке:

```
Количество элементов:  [100, 1000, 10000, 100000]
Метод быстрой сортирвоки, время выполнения: [5e-05, 0.00035, 0.00336, 0.02916]
Метод сортировки расчёской, время выполнения: [0.00013, 0.00253, 0.03981, 0.50521]
```

Рисунок 6 - Результат сравнения сортировок

Несложно заметить, что алгоритм быстрой сортировки работает куда быстрее, а растёт медленнее, чем алгоритм сортировки расчёской.

2 Блочная, пирамидальная и сортировка слиянием

2.1 Блочная сортировка

Блочная (или карманная) сортировка основана на предположении о равномерности входных данных. Алгоритмы блочной сортировки разбивает входные данные на k блоков (карманов), каждый из которых впоследствии отдельно сортируется, а результаты конкатенириуются.

Существует несколько вариантов реализации блочной сортировки: с заданным k или без, рекурсивная или нерекурсивная. Здесь будет рассмотрен нерекурсивный вариант без заданного параметра, поскольку в общем случае (нам не сказано, какие могут быть входные данные) нерекурсивная реализация сможет корректно обрабатывать числа, которые не сильно друг от друга отличаются. Код реализации алгоритма представлен на рисунке 7.

```python
def bucket_sort(a: list[float]):
    """Карманная сортировка"""
    min_, max_ = float("inf"), float("-inf")
    for el in a:
        min_ = min(el, min_)
        max_ = max(el, max_)
    range_ = max_ - min_
    buckets = [[] for _ in range(len(a))]
    for element in a:
        index = min(int(element * len(a) / range_), len(a) - 1)
        buckets[index].append(element)
    print(buckets)
    for i in range(len(buckets)):
        buckets[i] = sorted(buckets[i])
    result = []
    [result.extend(bucket) for bucket in buckets]
    return result
```

Рисунок 7 - Алгоритм блочной сортировки

В этой реализации карман, в который помещается каждое число, определяется как это число, умноженное на длину массива и поделенное на разницу максимального и минимального элементов в массиве. Всего карманов n, что равно длине массива входных данных. Понятно, что после распределения данных по карманам часть из них может оказаться пустыми.

Затем происходит сортировка каждого кармана с помощью встроенной в python сортировки, после чего все списки-карманы объединяются в один результирующий список отсортированных чисел. Пример работы кода можно увидеть на рисунке 8, где в отдельной строке выведены все карманы, которые создаются алгоритмом.

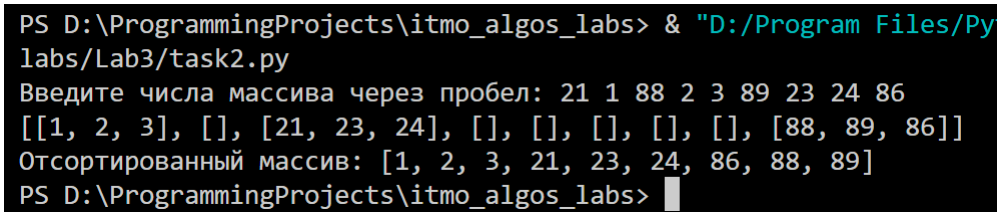

Рисунок 8 - Пример выполнения блочной сортировки

Нетрудно показать, что в худшем варианте (когда числа кластеризованы, т.е. близко расположены друг к другу, из-за чего все попадут в один карман) асимптотическая сложность алгоритма составит $O(n^2)$. В среднем, когда числа распределены равномерно, карманная сортировка выполняется за $O(n)$, так как вычисление минимального и максимального значений линейно, распределение чисел по карманам линейно и, что можно выяснить благодаря подсчетам, суммарная сложность сортировки всех карманов также линейна.

2.2 Пирамидальная сортировка

Пирамидальная сортировка (или сортировка кучей) основана на использовании структуры данных «куча». Основная идея – представить массив входных данных в виде кучи, а потом получать минимальный элемент из нее, пока куча не опустеет, при этом после взятия элемента восстанавливать кучу. Благодаря использованию кучи минимальный элемент можно получать за $O(logn)$, и так как всего элементов в массиве ровно n, то общая сложность алгоритма составит $O(n * logn)$ (и в худшем, и в среднем).

Для удобства построим кучу (то есть частично упорядоченное дерево) из исходного массива: детьми узла $a[i]$ будем считать элементы $a[i * 2 + 1]$ и $a[i * 2 + 2]$. В рекурсивной функции $make\_heap$ создается частично упорядоченно дерево таким образом: если элемент с индексом ix является

максимальным среди него самого и его двух потомков, то происходит выход из рекурсии, иначе родитель и сын с наибольшим значением меняются местами, и происходит вызов функции для индекса максимального среди этих трех узлов.

Код, реализующий алгоритм сортировки кучей представлен на рисунке 9.

```
def make_heap(a: list[float], key, ix):
    """Рекурсивное создание кучи"""
    left_child = 2 * ix + 1
    right_child = 2 * ix + 2
    max_ = ix

    if left_child < key:
        max_ = left_child if a[left_child] > a[max_] else max_
    if right_child < key:
        max_ = right_child if a[right_child] > a[max_] else max_
    if max_ == ix:
        return
    a[ix], a[max_] = a[max_], a[ix]
    make_heap(a, key, max_)


def heap_sort(a: list[float]):
    "Пирамидальная сортировка"

    for i in range(len(a) // 2, -1, -1):
        make_heap(a, len(a), i)

    for i in range(len(a) - 1, -1, -1):
        a[0], a[i] = a[i], a[0]
        make_heap(a, i, 0)
        print(a)
```

Рисунок 9 - Алгоритм пирамидальной сортировки

Изначально выполняется построение кучи, начиная от последнего слоя, элементы которого не являются листьями. Затем в цикле меняются местами нулевой элемент (наибольший) и i-й элемент, после чего массив необходимо восстановить до частично упорядоченного дерева и продолжить таким образом с конца сортировать массив.

Пример выполнения кода можно увидеть на рисунке 10, где кроме ввода исходного массива и вывода отсортированного выведен массив на каждом шаге сортировки. Можно заметить, как он заполняется с конца, а элементы до отсортированной части составляют частично упорядоченное дерево.

```
Введите числа массива через пробел: 14 61 85 24 74 26 17 50 40 45 21 32 59 58 13
[74, 61, 59, 50, 45, 32, 58, 24, 40, 13, 21, 14, 26, 17, 85]
[61, 50, 59, 40, 45, 32, 58, 24, 17, 13, 21, 14, 26, 74, 85]
[59, 50, 58, 40, 45, 32, 26, 24, 17, 13, 21, 14, 61, 74, 85]
[58, 50, 32, 40, 45, 14, 26, 24, 17, 13, 21, 59, 61, 74, 85]
[50, 45, 32, 40, 21, 14, 26, 24, 17, 13, 58, 59, 61, 74, 85]
[45, 40, 32, 24, 21, 14, 26, 13, 17, 50, 58, 59, 61, 74, 85]
[40, 24, 32, 17, 21, 14, 26, 13, 45, 50, 58, 59, 61, 74, 85]
[32, 24, 26, 17, 21, 14, 13, 40, 45, 50, 58, 59, 61, 74, 85]
[26, 24, 14, 17, 21, 13, 32, 40, 45, 50, 58, 59, 61, 74, 85]
[24, 21, 14, 17, 13, 26, 32, 40, 45, 50, 58, 59, 61, 74, 85]
[21, 17, 14, 13, 24, 26, 32, 40, 45, 50, 58, 59, 61, 74, 85]
[17, 13, 14, 21, 24, 26, 32, 40, 45, 50, 58, 59, 61, 74, 85]
[14, 13, 17, 21, 24, 26, 32, 40, 45, 50, 58, 59, 61, 74, 85]
[13, 14, 17, 21, 24, 26, 32, 40, 45, 50, 58, 59, 61, 74, 85]
[13, 14, 17, 21, 24, 26, 32, 40, 45, 50, 58, 59, 61, 74, 85]
Отсортированный массив: [13, 14, 17, 21, 24, 26, 32, 40, 45, 50, 58, 59, 61, 74, 85]
PS D:\ProgrammingProjects\itmo_algos_labs>
```

Рисунок 10 - Пример выполнения пирамидальной сортировки

2.3 Сортировка слиянием

Алгоритм сортировки слиянием основан на принципе «разделяй и властвуй»: входные данные делятся на две части по элементу по средним индексом, затем обе части рекурсивно сортируются, а отсортированные части впоследствии конкатериниуются. Алгоритм сортировки слиянием (без кода объединения двух частей) представлен на рисунке 11.

```python
def merge_sort(a: list[float], left, right):
    """Сортировка слиянием"""
    if left >= right or len(a) < 2:
        return
    mid = (left + right) // 2
    merge_sort(a, left, mid)
    merge_sort(a, mid + 1, right)
    merge(a, left, right, mid)
```

Рисунок 11 - Алгоритм сортировки слиянием

Стоит отдельно рассказать про слияние двух частей массива. Для оптимизации создадим два указателя на начала обеих частей массивов (пусть они называются $left\_half$ и $right\_half$, а сам массив – a). Тогда $a[i] = left\_half[p1]$, если $lef\_half[p1] \leq right\_half[p2]$, иначе $a[i] = right\_half[p2]$, после чего соответствующий указатель увеличивается на один. Когда какой-либо из указателей вышел за пределы длины массива, в результирующий массив оставшиеся элементы из второго массива. В результате за линейное время обе части будут объединены в одну. Реализующий слияние код представлен на рисунке 12.

```python
def merge(a: list[float], start: int, end: int, mid: int):
    """Слияние двух частей массива на очередном вызове рекурсии"""
    p1, p2 = 0, 0
    ix = start
    left_half, right_half = a[start:mid+1], a[mid+1:end+1]
    print(left_half, right_half)

    while p1 < len(left_half) and p2 < len(right_half):
        if left_half[p1] > right_half[p2]:
            a[ix] = right_half[p2]
            p2 += 1
        else:
            a[ix] = left_half[p1]
            p1 += 1
        ix += 1

    while p1 < len(left_half):
        a[ix] = left_half[p1]
        ix += 1
        p1 += 1

    while p2 < len(right_half):
        a[ix] = right_half[p2]
        ix += 1
        p2 += 1
```

Рисунок 12 - Алгоритм конкатенации частей массива

Пример выполнения программы представлен на рисунке 13. Здесь, помимо ввода исходного массива и вывода отсортированного, выведены на каждом шаге алгоритма части массивов, которые в конкретный момент

должны соединиться в один. И в худшем случае, и в среднем алгоритм сортировки слиянием выполняется за $O(n * logn)$.

```
PS D:\ProgrammingProjects\itmo_algos_labs> & D:/Progra
labs/Lab3/task2.py
Введите числа массива через пробел: 2 4 1 6 8 3 5 7
[2] [4]
[1] [6]
[2, 4] [1, 6]
[8] [3]
[5] [7]
[3, 8] [5, 7]
[1, 2, 4, 6] [3, 5, 7, 8]
Отсортированный массив: [1, 2, 3, 4, 5, 6, 7, 8]
PS D:\ProgrammingProjects\itmo_algos_labs>
```

Рисунок 13 - Пример выполнения сортировки слиянием

3 Оценка методов сортировки

Основными количественными характеристиками алгоритмов сортировки, которые нас будут интересовать – это время выполнения в худшем случае, время выполнения в среднем и затраты дополнительной памяти. Для удобства эти данные представлены в таблице 1.

Таблица 1 - Сравнение алгоритмов сортировки

	Выполнение в среднем	Выполнение в худшем случае	Затраты дополнительной памяти
Быстрая сортировка	$O(n * logn)$	$O(n^2)$	$O(logn)$
Сортировка расческой	$O(n * logn)$	$O(n^2)$	$O(1)$
Блочная сортировка	$O(n + k)$	$O(n^2)$	$O(n + k)$
Пирамидальная сортировка	$O(n * logn)$	$O(n * logn)$	$O(1)$
Сортировка слиянием	$O(n * logn)$	$O(n * logn)$	$O(n)$

Как можно заметить из таблицы, наиболее быстрыми из представленных сортировок являются блочная, пирамидальная и сортировка слиянием, поскольку они имеют наилучшие показатели выполнения в среднем и худшем случаях, однако есть некоторые нюансы относительно каждого из алгоритмов, которые нужно упомянуть отдельно (см. таблицу 2).

Алгоритм	Достоинства	Недостатки
Быстрая сортировка	– Один из самых быстродействующих – Алгоритм очень короткий – Хорошо сочетается с механизмами кэширования и вирт уальной памяти. – Работает на связных списках и других структурах с последовательным доступом,	– Сильно деградирует по скорости $O(n^2)$. в худшем или близком к нему случае, что может случиться при неудачных входных данных.

	допускающих эффективный проход как от начала к концу, так и от конца к началу.	– Прямая реализация в виде функции с двумя рекурсивными вызовами может привести к ошибке переполнения стека. – Неустойчива (меняет элементы с одинаковыми значениями местами).
Сортировка расческой	– Легко реализуема и проста для понимания. – Не использует дополнительной памяти, так как реализована не через рекурсию.	– Неэффективна в сравнении с некоторыми другими сортировками
Блочная сортировка	– Легко реализуема и проста для понимания. – Имеет высокую скорость выполнения	– Если данные распределены не равномерно, сортировка будет выполняться долго, а иногда может и не выполниться.
Пирамидальная сортировка	– Имеет высокую скорость выполнения в худшем случае – Сортирует данные на месте, поэтому не требует дополнительной памяти	– Не является устойчивой – На почти отсортированных данных работает не быстрее, чем на хаотических – Не поддерживается распараллеливание – Из-за необходимости получать произвольный элемент не может использовать связанные списки и другие структуры последовательного доступа

Сортировка слиянием	– Устойчива – Поддерживает кэширование данных и распараллеливание – Не имеет «трудных» входных данных	– На почти отсортированных данных работает не быстрее, чем на хаотических – Требует дополнительной памяти по размеру массива входных данных

Исходя из таблиц, можно сделать вывод, что все алгоритмы сортировки имеют как преимущества, так и недостатки, поэтому выбор определенного алгоритма сортировки зависит в первую очередь от поставленной задачи.

ЗАКЛЮЧЕНИЕ

Результатом выполнения работы стало повышение навыков по использованию различных методов сортировки. В ходе работы были написаны алгоритмы быстрой, блочной, пирамидальной сортировки, сортировки расческой и сортировки слиянием, эти алгоритмы далее были проанализированы на наличие достоинств и недостатков. Таким образом, полученные навыки позволят более эффективно применять сортировку данных в практической деятельности в зависимости от типа задачи.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Ахо А.В., Хопкрофт Д.Э., Ульман Дж.Д. Алгоритмы и структуры данных [Электронный ресурс] – URL: https://vk.com/wall-114485185_260 (дата обращения 14.10.2023)
2. Сортировка кучей (пирамидальная сортировка) :: Heap Sort [Электронный ресурс] – URL: https://youtu.be/DU1uG5310x0?si=MYjYHSERk19DA1YD (дата обращения: 14.10.2023)
3. Bucket Sort [Электронный ресурс] – URL: https://en.wikipedia.org/wiki/Bucket_sort (дата обращения: 14.10.2023)
4. Merge sort algorithm [Электронный ресурс] – URL: https://youtu.be/TzeBrDU-JaY?si=meoarehE2nzzVoZU (дата обращения: 14.10.2023)
5. Грокаем алгоритмы [Электронный ресурс] – URL: https://verstkag.github.io/books/Грокаем%20алгоритмы.pdf (дата обращения: 16.10.2023)
6. Comb Sort | GeeksforGeeks [Электронный ресурс] – URL: https://www.youtube.com/watch?v=n51GFZHXlYY (дата обращения: 16.10.2023)
7. Быстрая сортировка [Электронный ресурс] – URL: https://ru.wikipedia.org/wiki/Быстрая_сортировка (дата обращения: 16.10.2023)