Введение

Иногда данные, которые мы храним или извлекаем в приложении, могут находится в беспорядочном состояние. И иногда возникает необходимость упорядочивания данные прежде чем их можно будет эффективно использовать. За все эти годы учеными было создано множество алгоритмов сортировки для организации данных.

В этой статье мы рассмотрим наиболее популярные алгоритмы сортировки, разберем, как они работают, и напишем их на Python. Мы также сравним, как быстро они сортируют элементы в списке.

Для простоты реализации алгоритмов сортировать числа будем в порядке их возрастания.

Пузырьковая сортировка (Bubble Sort)

Этот самый простой алгоритм сортировки который выполняет итерации по списку, сравнивая элементы попарно и меняя их местами, пока более крупные элементы не перестанут «всплывать» до конца списка, а более мелкие элементы не будут оставаться «снизу».

Объяснение

Начнем со сравнения первых двух элементов списка. Если первый элемент больше второго, мы меняем их местами. Если они уже в нужном порядке, мы оставляем их как есть. Затем мы переходим к следующей паре элементов, сравниваем их значения и меняем местами при необходимости. Этот процесс продолжается до последней пары элементов в списке.

Достигнув конца списка, повторяем этот процесс для каждого элемента снова. Хотя это крайне неэффективно. Что если в массиве нужно сделать только одну замену? Почему мы все еще повторяем, даже если список уже отсортирован? Получается нам нужно пройти список **n^2** раза.

Очевидно, что для оптимизации алгоритма нам нужно остановить его, когда он закончит сортировку.

Откуда нам знать, что мы закончили сортировку? Если бы элементы были отсортированы, то нам не пришлось бы их менять местами. Таким образом, всякий раз, когда мы меняем элементы, мы устанавливаем флаг в

True, чтобы повторить процесс сортировки. Если перестановок не произошло, флаг останется False, и алгоритм остановится.

Реализация

Мы можем реализовать пузырьковую сортировку в Python следующим образом:

```
def bubble_sort(nums):
    # We set swapped to True so the loop looks runs at least
once
    swapped = True
    while swapped:
        swapped = False
        for i in range(len(nums) - 1):
            if nums[i] > nums[i + 1]:
                # Swap the elements
                nums[i], nums[i + 1] = nums[i + 1], nums[i]
                # Set the flag to True so we'll loop again
                swapped = True


# Verify it works
random_list_of_nums = [5, 2, 1, 8, 4]
bubble_sort(random_list_of_nums)
print(random_list_of_nums)
```

Алгоритм работает в цикле **while**, прерываясь только тогда, когда никакие элементы не меняются местами. Вначале мы установили для **swapped** значение **True**, чтобы алгоритм прошел по списку хотя бы один раз.

Сложность

Сложность пузырьковой сортировки в худшем случае (когда список отсортирован в обратном порядке) равна O(**n^2**).

Сортировка выбором (Selection Sort)

Этот алгоритм сегментирует список на две части: отсортированные и несортированные. Он постоянно удаляет наименьший элемент из несортированного сегмента списка и добавляет его в отсортированный сегмент.

Объяснение

На практике нам не нужно создавать новый список для отсортированных элементов, мы будет обрабатывать крайнюю левую часть списка как отсортированный сегмент. Затем мы ищем во всем списке наименьший элемент и меняем его на первый элемент.

Теперь мы знаем, что первый элемент списка отсортирован, мы получаем наименьший элемент из оставшихся элементов и заменяем его вторым элементом. Это повторяется до тех пор, пока последний элемент списка не станет оставшимся элементом для изучения.

Реализация

```
def selection_sort(nums):
    # значение i соответствует тому, сколько значений было
отсортировано
    for i in range(len(nums)):
        # Мы предполагаем, что первый элемент несортированного
сегмента является наименьшим
        lowest_value_index = i
        # Этот цикл перебирает несортированные элементы
        for j in range(i + 1, len(nums)):
            if nums[j] < nums[lowest_value_index]:
                lowest_value_index = j
        # Поменять местами значения самого низкого
несортированного элемента с первым несортированным
        nums[i], nums[lowest_value_index] =
nums[lowest_value_index], nums[i]


# Проверяем, что это работает
random_list_of_nums = [12, 8, 3, 20, 11]
selection_sort(random_list_of_nums)
print(random_list_of_nums)
```

Мы видим, что по мере того как **i** увеличивается, нам нужно проверять все меньше элементов.

Сложность

Сложность сортировки выбором в среднем составляет O(**n^2**).

Сортировка вставками (Insertion Sort)

Как и Сортировка выбором, этот алгоритм сегментирует список на отсортированные и несортированные части. Он перебирает

несортированный сегмент и вставляет просматриваемый элемент в правильную позицию отсортированного списка.

Объяснение

Предполагается, что первый элемент списка отсортирован. Затем мы переходим к следующему элементу, назовем его **x**. Если **x** больше первого элемента, мы оставляем его как есть. Если **x** меньше, мы копируем значение первого элемента во вторую позицию и затем устанавливаем первый элемент в **x**.

Когда мы переходим к другим элементам несортированного сегмента, мы непрерывно перемещаем более крупные элементы в отсортированном сегменте вверх по списку, пока не встретим элемент меньше **x**, или не достигнем конца отсортированного сегмента, а затем поместим **x** в его правильное положение.

Реализация

```
def insertion_sort(nums):
    # Начнем со второго элемента, так как мы предполагаем, что
первый элемент отсортирован
    for i in range(1, len(nums)):
        item_to_insert = nums[i]
        # И сохранить ссылку на индекс предыдущего элемента
        j = i - 1
        # Переместить все элементы отсортированного сегмента
вперед, если они больше, чем элемент для вставки
        while j >= 0 and nums[j] > item_to_insert:
            nums[j + 1] = nums[j]
            j -= 1
        # Вставляем элемент
        nums[j + 1] = item_to_insert


# Проверяем, что это работает
random_list_of_nums = [9, 1, 15, 28, 6]
insertion_sort(random_list_of_nums)
print(random_list_of_nums)
```

Сложность

Сложность сортировки вставками в среднем равна O(**n^2**).

Пирамидальная сортировка (Heap Sort) (англ. *Heapsort*, «Сортировка кучей»)

Этот популярный алгоритм сортировки, как сортировки вставками и выбором, сегментирует список на отсортированные и несортированные части. Он преобразует несортированный сегмент списка в структуру данных типа куча (heap), чтобы мы могли эффективно определить самый большой элемент.

Объяснение

Мы начинаем с преобразования списка в **Max Heap** – бинарное дерево, где самым большим элементом является корневой узел. Затем мы помещаем этот элемент в конец списка. Затем мы восстанавливаем нашу **Max Heap**, которая теперь имеет на одно меньшее значение, помещая новое наибольшее значение перед последним элементом списка.

Мы повторяем этот процесс построения кучи, пока все узлы не будут удалены.

Реализация

Мы создаем вспомогательную функцию **heapify** для реализации этого алгоритма:

```
def heapify(nums, heap_size, root_index):
    # Предположим, что индекс самого большого элемента является
корневым индексом
    largest = root_index
    left_child = (2 * root_index) + 1
    right_child = (2 * root_index) + 2

    # Если левый потомок корня является допустимым индексом, а
элемент больше
    # чем текущий самый большой элемент, то обновляем самый
большой элемент
    if left_child < heap_size and nums[left_child] >
nums[largest]:
        largest = left_child

    # Делайте то же самое для right_child
    if right_child < heap_size and nums[right_child] >
nums[largest]:
        largest = right_child
```

```
    # Если самый большой элемент больше не является корневым
элементом, меняем их местами
    if largest != root_index:
        nums[root_index], nums[largest] = nums[largest],
nums[root_index]
        # Heapify the new root element to ensure it's the
largest
        heapify(nums, heap_size, largest)


def heap_sort(nums):
    n = len(nums)

    # Создаем Max Heap из списка
    # Второй аргумент означает, что мы останавливаемся на
элементе перед -1, то есть на первом элементе списка.
    # Третий аргумент означает, что мы повторяем в обратном
направлении, уменьшая количество i на 1
    for i in range(n, -1, -1):
        heapify(nums, n, i)

    # Перемещаем корень max hea в конец
    for i in range(n - 1, 0, -1):
        nums[i], nums[0] = nums[0], nums[i]
        heapify(nums, i, 0)


# Проверяем, что все работает
random_list_of_nums = [35, 12, 43, 8, 51]
heap_sort(random_list_of_nums)
print(random_list_of_nums)
```

Сложность

В среднем сложность сортировки кучи составляет O(**nlog (n)**), что уже значительно быстрее, чем в предыдущих алгоритмах.

Сортировка слиянием (Merge Sort)

Этот алгоритм «разделяй и властвуй» разбивает список пополам и продолжает разбивать список на пары, пока в нем не будут только одиночные элементы.

Соседние элементы становятся отсортированными парами, затем отсортированные пары объединяются и сортируются с другими парами.

Этот процесс продолжается до тех пор, пока мы не получим отсортированный список со всеми элементами несортированного списка.

Объяснение

Мы рекурсивно разделяем список пополам, пока не получим списки с одиночным размером. Затем мы объединяем каждую половину, которая была разделена, и при этом сортируя их.

Сортировка осуществляется путем сравнения наименьших элементов каждой половины. Первый элемент каждого списка сравнивается с первым. Если первая половина начинается с меньшего значения, то мы добавляем ее в отсортированный список. Затем мы сравниваем второе наименьшее значение первой половины с первым наименьшим значением второй половины.

Каждый раз, когда мы выбираем меньшее значение в начале половины, мы перемещаем индекс, элемент которого нужно сравнить на единицу.

Реализация

```
def merge(left_list, right_list):
    sorted_list = []
    left_list_index = right_list_index = 0

    # Мы будет часто используем длины списков, поэтому удобно
сразу создавать переменные для этого
    left_list_length, right_list_length = len(left_list),
len(right_list)

    for _ in range(left_list_length + right_list_length):
        if left_list_index < left_list_length and
right_list_index < right_list_length:
            # Мы проверяем, какое значение с начала каждого
списка меньше
            # Если элемент в начале левого списка меньше,
добавляем его в отсортированный список
            if left_list[left_list_index] <=
right_list[right_list_index]:
                sorted_list.append(left_list[left_list_index])
                left_list_index += 1
            # Если элемент в начале правого списка меньше,
добавляем его в отсортированный список
            else:
                sorted_list.append(right_list[right_list_index])
                right_list_index += 1
```

```
        # Если мы достигли конца левого списка, добавляем
элементы из правого списка
        elif left_list_index == left_list_length:
            sorted_list.append(right_list[right_list_index])
            right_list_index += 1
        # Если мы достигли конца правого списка, добавляем
элементы из левого списка
        elif right_list_index == right_list_length:
            sorted_list.append(left_list[left_list_index])
            left_list_index += 1

    return sorted_list


def merge_sort(nums):
    # Если список представляет собой один элемент, возвращаем
его
    if len(nums) <= 1:
        return nums

    # Используем деление с округлением по наименьшему целому для
получения средней точки, индексы должны быть целыми числами
    mid = len(nums) // 2

    # Сортируем и объединяем каждую половину
    left_list = merge_sort(nums[:mid])
    right_list = merge_sort(nums[mid:])

    # Объединить отсортированные списки в новый
    return merge(left_list, right_list)


# Проверяем, что все работает
random_list_of_nums = [120, 45, 68, 250, 176]
random_list_of_nums = merge_sort(random_list_of_nums)
print(random_list_of_nums)
```

Обратите внимание, что функция **merge_sort()**, в отличие от предыдущих алгоритмов сортировки, возвращает новый отсортированный список, а не сортирует существующий список.

Поэтому для сортировки слиянием требуется пространство в памяти для создания нового списка того же размера, что и входной список.

Сложность

В среднем сложность сортировки слиянием составляет O(**nlog (n)**).

Быстрая сортировка (Quick Sort)

Это то же алгоритм «разделяй и властвуй» и его наиболее часто используют их описанных в этой статье. При правильной настройке он чрезвычайно эффективен и не требует дополнительного пространства памяти как сортировка слиянием. Мы разделяем список вокруг элемента точка опоры, сортируя значения вокруг этой точки.

Объяснение

Быстрая сортировка начинается с разбиения списка – выбора одного значения списка, которое будет в его отсортированном месте. Это значение называется **опорным**. Все элементы, меньшие, чем этот элемент, перемещаются влево. Все более крупные элементы перемещены вправо.

Зная, что опорный элемент находится на своем правильном месте, мы рекурсивно сортируем значения вокруг этого элемента, пока не будет отсортирован весь список.

Реализация

```
# Есть разные способы реализовать быструю сортировки
# мы выбрали схема Tony Hoare
def partition(nums, low, high):
    # Мы выбираем средний элемент, в качестве опорного.
Некоторые реализации выбирают
    # первый элемент или последний элемент или вообще случайный
элемент.
    pivot = nums[(low + high) // 2]
    i = low - 1
    j = high + 1
    while True:
        i += 1
        while nums[i] < pivot:
            i += 1

        j -= 1
        while nums[j] > pivot:
            j -= 1

        if i >= j:
            return j

        # Если элемент в i (слева от оси) больше, чем
        # элемент в J (справа от оси), то поменять их местами
        nums[i], nums[j] = nums[j], nums[i]


def quick_sort(nums):
    # Создаем вспомогательную рекурсивную функцию
```

```
    def _quick_sort(items, low, high):
        if low < high:
            # Это индекс после опорного элемента, по которому
наши списки разделены
            split_index = partition(items, low, high)
            _quick_sort(items, low, split_index)
            _quick_sort(items, split_index + 1, high)

    _quick_sort(nums, 0, len(nums) - 1)


# Проверяем, что все работает
random_list_of_nums = [22, 5, 1, 18, 99]
quick_sort(random_list_of_nums)
print(random_list_of_nums)
```

Сложность

В среднем сложность быстрой сортировки составляет O(**nlog (n)**).

Примечание. Алгоритм быстрой сортировки будет работать медленно, если опорный элемент будет наименьшим или наибольшим элементом списка. Быстрая сортировка обычно работает быстрее с более сбалансированными значениями. В отличие от сортировки кучи и сортировки слиянием, обе из которых имеют худшие времена O(**nlog (n)**), быстрая сортировка имеет худшее время O(**n^2**).

Задание

Чтобы понять, как быстро работают рассмотренные алгоритмы, необходимо сгенерировать список из N чисел со значениями от 0 до 1000. Затем необходимо определить время, необходимое для завершения каждого алгоритма. Повторить это 10 раз, чтобы можно было более надежно установить производительность сортировки.

Таким образом, для каждого вида сортировки (в том числе и для встроенной сортировки) провести сортировку массивов при N = 100, 1000, 3000, 5000, 7000, 10000, 20000, 50000 элементов. Для каждого вида сортировки провести не менее 10 измерений времени сортировки. Для каждого вида сортировки построить график зависимости времени сортировки от размера массива (оценить зависимость нотации большого O от N). Сравнить полученные графики с теоретическими и сделать выводы.

Весь код сделать в одном файле, графики строить с помощью библиотеки matplotlib. График должен содержать названия линий (метод сортировки) и диаграмм с легендой.