Отчет к первой лоабораторной работе

Этот модуль содержит функции для выполнения сортировки массива с помощью алгоритмов сортировки кучей, быстрой сортировки и сортировки выбором.

Функции

`heapify(arr, n, i)`

Эта функция преобразует подмассив `arr[i:]` в структуру кучи.

Аргументы:

- `arr` (list): Входной массив.
- `n` (int): Размер массива.
- `i` (int): Индекс, с которого начинается преобразование в кучу.

Алгоритм:

1. Определяет корневой элемент, левый и правый дочерние элементы.
2. Сравнивает корневой элемент с его дочерними элементами и находит наибольший элемент.
3. Если наибольший элемент не является корневым, меняет их местами.
4. Рекурсивно применяет функцию к измененному подмассиву.

`heap_sort(arr)`

Эта функция сортирует заданный массив с использованием алгоритма сортировки кучей.

Аргументы:

- `arr` (list): Массив для сортировки.

Алгоритм:

1. Преобразует массив в кучу.
2. Последовательно извлекает максимальный элемент из кучи и помещает его в конец массива.
3. Повторяет процесс для оставшегося массива, уменьшая его размер.

`quick_sort(arr)`

Эта функция сортирует массив с использованием алгоритма быстрой сортировки.

Аргументы:

- `arr` (list): Массив для сортировки.

Возвращает:

- list: Отсортированный массив.

Алгоритм:

1. Выбирает опорный элемент (pivot).
2. Делит массив на три части: элементы меньше опорного, равные ему и больше него.
3. Рекурсивно сортирует части и объединяет их.

`selection_sort(arr)`

Эта функция выполняет сортировку массива с использованием алгоритма сортировки выбором.

Аргументы:

- `arr` (list): Массив для сортировки.

Возвращает:

- list: Отсортированный массив.

Алгоритм:

1. Находит минимальный элемент в неотсортированной части массива.
2. Меняет местами текущий элемент с найденным минимальным.
3. Повторяет процесс для оставшейся части массива.

Пример использования

```
arr = [4, 10, 3, 5, 1]
heap_sort(arr)
print(arr)  # [1, 3, 4, 5, 10]
```

Измерение времени сортировки

Для сравнения времени выполнения различных алгоритмов сортировки используется функция `measure_sorting_time`:

```
def measure_sorting_time(sort_func, data):
    start_time = time.time()
    sort_func(data)
    end_time = time.time()
    return end_time - start_time
```

Загрузка данных и создание подмножеств данных различных размеров выполняются с помощью функций из модуля `data.read`:

```
all_data = read_plants_data('data/plants.csv')
data_lengths = [1000, 5000, 10000, 15000, 20000, 25000, 50000, 75000,
100000]
data_sets = [select_random_subset(all_data, length) for length in
data_lengths]
```

Сравнение времени выполнения

Сравнение времени выполнения алгоритмов сортировки осуществляется путем измерения времени сортировки для каждого подмножества данных:

```
quicksort_times = []
heapsort_times = []
selectionsort_times = []

for data in data_sets:
    quicksort_times.append(measure_sorting_time(quick_sort, data))
    heapsort_times.append(measure_sorting_time(heap_sort, data))
    selectionsort_times.append(measure_sorting_time(selection_sort, data))
```

Построение графика

График времени выполнения алгоритмов сортировки в зависимости от размера данных:

```
plt.plot(data_lengths, quicksort_times, label="Quicksort")
plt.plot(data_lengths, heapsort_times, label="Heapsort")
plt.plot(data_lengths, selectionsort_times, label="Selectionsort")
plt.xlabel("Data Length")
plt.ylabel("Time (seconds)")
plt.title("Sorting Algorithms Comparison")
plt.legend()
plt.savefig('plot.png')
plt.show()
```

Заключение

В данном отчете представлен код для реализации и сравнения трех различных алгоритмов сортировки: сортировка кучей, быстрая сортировка и сортировка выбором