Задание 1.

Для выполнения этого задания выбрал алгоритм из ДЗ-2 (задача 4): найти сумму n элементов следующего ряда: 1, -0.5, 0.25, -0,125…

В ДЗ-2 сумма найдена с использованием рекурсии (этот алгоритм перенесён в файл task01.py).
Дополнительно для сравнения скорости нашёл сумму с использованием цикла (алгоритм реализован в файле task02.py).
В обоих алгоритмах 1000 раз проведено вычисление суммы 990 элементов ряда (больше не смог из-за ограничения рекурсии).

Результаты работы алгоритма с рекурсией:

```
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.365    0.365 <string>:1(<module>)
        1    0.001    0.001    0.365    0.365 task01.py:16(main)
990000/1000    0.364    0.000    0.364    0.000 task01.py:3(sum_series)
        1    0.000    0.000    0.365    0.365 {built-in method builtins.exec}
        1    0.000    0.000    0.000    0.000 {built-in method builtins.print}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Видно, что функция расчёта суммы вызывалась 1000*990 раз, на что потрачено 0,364 сек.

Результаты работы алгоритма с циклом:

```
 ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      1    0.000    0.000    0.058    0.058 <string>:1(<module>)
      1    0.000    0.000    0.058    0.058 task02.py:18(main)
   1000    0.057    0.000    0.057    0.000 task02.py:3(sum_series)
      1    0.001    0.001    0.059    0.059 {built-in method builtins.exec}
      1    0.000    0.000    0.000    0.000 {built-in method builtins.print}
      1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Функция расчёта суммы вызывалась 1000 раз, на что потрачено 0,057 сек.

Можно сделать вывод, что расчёт суммы элементов ряда эффективнее проводить с использованием цикла (быстрее в 6 раз), чем с использованием рекурсии.

Асимптотическая оценка сложности алгоритма:
O(n) = n, т. к. для вычисления суммы n членов ряда нам необходимо сделать вычисления n раз.

Задание 2.

Для выполнения данного задания использовать решето Эратосфена не совсем корректно (т. к. по этому алгоритму мы ищем простые числа в диапазоне, а не какое-то их количество). Тем не менее, в файле task03.py реализован алгоритм нахождения простых чисел «Решето Эратосфена» в диапазоне [0, 100000], получено 9592 простых числа.

```
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.001    0.001    0.028    0.028 <string>:1(<module>)
     1    0.023    0.023    0.027    0.027 task03.py:4(main)
     1    0.003    0.003    0.003    0.003 task03.py:7(<listcomp>)
     1    0.000    0.000    0.028    0.028 {built-in method builtins.exec}
     1    0.000    0.000    0.000    0.000 {built-in method builtins.len}
     2    0.000    0.000    0.000    0.000 {built-in method builtins.print}
  9592    0.001    0.000    0.001    0.000 {method 'append' of 'list' objects}
     1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Выполнение программы занимает 0,028 сек.

Асимптотическая оценка сложности алгоритма:
$O(n) = n \log \log n$

Второй алгоритм нахождения простых чисел (реализован в файле task04.py) заключается в следующем.
1. Создаём список простых чисел (вначале он пуст).
2. Перебираем числа от 2 до того числа, когда длина списка простых чисел не станет равной заданному значению (нашли X простых чисел). Если число не делится ни на одно из уже найденных простых чисел, значит, оно само простое — добавляем его в список простых чисел.
3. Попытка оптимизации: если очередное число (больше 5) делится нацело на 2, 3 или 5 — пропускаем его.

```
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    3.662    3.662 <string>:1(<module>)
     1    3.650    3.650    3.662    3.662 task04.py:4(main)
     1    0.000    0.000    3.662    3.662 {built-in method builtins.exec}
126661    0.010    0.000    0.010    0.000 {built-in method builtins.len}
     2    0.000    0.000    0.000    0.000 {built-in method builtins.print}
  9592    0.001    0.000    0.001    0.000 {method 'append' of 'list' objects}
     1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Выполнение программы заняло 3,662 сек., что в 13 раз медленнее решета Эратосфена.

Если закомментировать проверку очередного числа на делимость на 2, 3 или 5 (вычеркнуть п.3 алгоритма), то получим только чуть-чуть худшие результаты (скриншот ниже).

Объясняю это тем, что числа 2, 3 и 5 находятся в самом начале списка простых чисел, поэтому нет особого выигрыша в том, чтобы вначале проверить очередное число на делимость отдельно, а не в общем списке.

```
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    3.846    3.846 <string>:1(<module>)
     1    3.828    3.828    3.846    3.846 task04.py:4(main)
     1    0.000    0.000    3.846    3.846 {built-in method builtins.exec}
199982    0.015    0.000    0.015    0.000 {built-in method builtins.len}
     2    0.000    0.000    0.000    0.000 {built-in method builtins.print}
  9592    0.002    0.000    0.002    0.000 {method 'append' of 'list' objects}
     1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Выполнение программы в этом случае заняло 3,846 сек.

С асимтотической оценкой сложности алгоритма затрудняюсь.